091M4041H - Algorithm Design and Analysis

Assignment 2

2019 年 8 月 30 日

目录

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?

1.1 Optimal Substructure and DP Equation

设 {1,2,...,n} 为这条这条街上的 n 个房子的序号，v_i 表示第 i 个房子所拥有的金额。

利用 DP 求解这一问题时，我们可以把问题看成两部分，即每一次是否选择这条街上的最后一个房子 n 抢劫。如果不抢劫房子 n，那么问题改变为在房子 {1,2,...,n-1} 中选择房子抢劫；如果抢劫房子 n，获得房子 n 的金额 v_n，根据不能抢劫相邻房子的规则，房子 n-1 不能被抢劫，于是接下来只能从房子 {1,2,...,n-2} 中选择。每次寻找两个子问题中最大收益的子问题，直到所有的房子都已经或不能被抢劫，DP Equation 描述如公式 (1) 所示。

如果这条街是一个环，同样首先考虑将问题看成两部分，只是对于第一步房子 n 的选择有所不同: 如果不抢劫房子 n，继续在 {1,2,...,n-1} 中选择；如果抢劫房子 n，获得房子 n 的金额 v_n，因为房子 1 和 n-1 与 n 相邻所以排除出选择列表，剩下子问题为在 {2,...,n-2} 中选择。在选完房子 n 后，将最大收益的子问题按照公式 (1) 继续迭代寻找，同样每次寻找两个子问题中最大收益的子问题，直到所有的房子都已经或不能被抢劫，环形街的 DP Equation 描述如公式 (2) 所示。

$$OPT(\{1,2,...,n\}) = max \begin{cases} OPT(\{1,2,...,n-1\}) & don't\ choose\ n \\ OPT(\{1,2,...,n-2\}) + v_n & choose\ n \end{cases} \tag{1}$$

$$OPT(\{1,2,...,n\}) = max \begin{cases} OPT(\{1,2,...,n-1\}) & don't\ choose\ n \\ OPT(\{2,...,n-2\}) + v_n & choose\ n \end{cases} \tag{2}$$

then substitute $OPT(\{1,2,...,n-1\})$ or $OPT(\{2,...,n-2\})$ to Equation (1)

1.2 Algorithm Description

设 $OPT(1,n)$ 表示从 1 到 n 的房子中进行选择所获得的最大收益，首先初始化 $OPT(1,0)$,$OPT(1,-1)$,$OPT(2,1)$, $OPT(2,0)$,$OPT(2,-1)$ 的值为 0，表示无房子可选择时的收益为 0。当所有房子在一条直线的街上时，i 从 1 到 n 遍历，对于第 i 个房子判断是否选择第 i 个，寻找两个子问题中较大收益的金额作为当前 $OPT(1,i)$ 的最优解，直到解得 $OPT(1,n)$ 即为最优解。

若房子形成一个环，则需要计算 $OPT(2,i)$，用以判断环形街上的 $OPT(1,n)$ 是否选择最后一个房子 n，选择 $OPT(1,n-1)$ 与 $v_n + OPT(2,n-1)$ 两个子问题中收益最大的子问题，求得 $OPT(1,n)$ 即为环形街上收益最大解。

Algorithm 1 Obtain the maximum money from the houses on the street

```
function MoneyRobbing(n)
    OPT(1,0) = OPT(1,-1) = OPT(2,1) = OPT(2,0) = OPT(2,-1) = 0;
    for i = 1 to n do
        OPT(1,i) = max{OPT(1,i-1), v_i + OPT(1,i-2)};
    end for
    if the street is a circle then
        for i = 2 to n-2 do
            OPT(2,i) = max{OPT(2,i-1), v_i + OPT(2,i-2)};
        end for
        OPT(1,n) = max{OPT(1,n-1), v_n + OPT(2,n-2)};
    end if
end function
```

1.3 Proof of Algorithm Correctness

采用数学归纳法证明。

对于 n=1，可得到:

$$OPT(1,1) = max\{OPT(1,0), v_1 + OPT(1,-1)\} = max\{0, v_1\} = v_1 \tag{3}$$

假设 n=k-1 和 k-2，可得到

$$OPT(1,k-1) = max\{OPT(1,k-2), v_{k-1} + OPT(1,k-3)\} \tag{4}$$

$$OPT(1,k-2) = max\{OPT(1,k-3), v_{k-2} + OPT(1,k-4)\}$$

对于 n=k, 则有

$$OPT(1,k) = max\{OPT(1,k-1), v_k + OPT(1,k-2)\} \tag{5}$$

$$= max\{OPT(1,k-2), v_{k-1} + OPT(1,k-3)\} + max\{OPT(1,k-3), v_{k-2} + OPT(1,k-4)\}$$

因此，当街是一条直线时，算法是正确的。

当街是一个环时，对于 n=1，可得到:

$$OPT(1,1)_{circle} = max\{OPT(1,0), v_1 + OPT(2,-1)\} = max\{0, v_1\} = v_1 \tag{6}$$

假设 n=k-1 和 k-2，可得到

$$OPT(1,k-1)_{circle} = max\{OPT(1,k-2), v_{k-1} + OPT(2,k-3)\} \tag{7}$$

$$OPT(1,k-1)_{line} = max\{OPT(1,k-2), v_{k-1} + OPT(1,k-3)\}$$

$$OPT(2,k-2) = max\{OPT(2,k-3), v_{k-2} + OPT(2,k-4)\}$$

对于 n=k, 则有

$$OPT(1,k)_{circle} = max\{OPT(1,k-1)_{line}, v_k + OPT(2,k-2)\} \tag{8}$$

$$= max\{OPT(1,k-2), v_{k-1} + OPT(1,k-3)\} + max\{OPT(2,k-3), v_{k-2} + OPT(2,k-4)\}$$

因此，当街是一个环时，算法是正确的。

综上所述，算法是正确的。

1.4 Complexity Analysis

因为问题中房子个数为 (1,n)，OPT 数组中最优解的个数为 n，每次分解成两个子问题，因此 OPT 数组需遍历两遍。因此算法的时间复杂度为:

$$T(n) = 2 * n = O(n) \tag{9}$$

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.

2.1 Optimal Substructure and DP Equation

设 OPT(root) 表示以 root 为根的二叉树的最大节点和，根据不能选择相邻节点的规则，可以把问题看成两个子问题，即选择 root 和不选 root 两个部分。如果选择 root 节点，那么接下来只能从以 $root \to left \to left, root \to left \to right, root \to right \to left, root \to right \to right$ 为根的子树中选择节点；如果不选择 root 节点，那么接下来可以从 $root \to left, root \to right$ 为根的子树中选择节点。DP Equation 的描述如公式 (10) 所示。

$$OPT(root) = max \begin{cases} OPT(root \to left \to left) + OPT(root \to left \to right) & \\ +OPT(root \to right \to left) + OPT(root \to right \to right) + v_{root} & choose\ root \\ OPT(root \to left) + OPT(root \to right) & don't\ choose\ root \end{cases} \tag{10}$$

2.2 Algorithm Description

从二叉树的叶子节点出发，初始化当节点为空时 OPT(NULL)=0，计算以当前节点为根的树的最大节点权之和 OPT(node)，通过之前的描述将问题分成两部分，选择当前节点时计算当前节点权值加上其所有孙子子树的 OPT 之和，不选择当前节点时计算当前节点的左右子树的 OPT 之和，判断二者较大者将其赋值给当前节点的 OPT 值。选完所有当前节点后，更新当前节点集为所有当前节点的父节点，以此循坏，直到所有当前节点的父节点全为空时，最后回溯寻找 OPT 最优解的节点选择加入选择集。

Algorithm 2 Find the nodes with the maximum sum of weight in a binary tree

1: **function** NodeSelection($BTree\ root$)
2: $set_{selection} = \{\}$;
3: $set_{current} = \{all\ leaf\ nodes\ in\ the\ binary\ tree\}$;
4: $OPT(NULL) = 0$;
5: **for** $every\ node\ in\ set_{current}$ **do**
6: $substruct_1 = OPT(node \to left \to left) + OPT(node \to left \to right) + OPT(node \to right \to left) + OPT(node \to right \to right) + v_{node}$;
7: $substruct_2 = OPT(node \to left) + OPT(node \to right)$;
8: $OPT(node) = max\{substruct_1, substruct_2\}$;
9: $Pop\ node\ from\ set_{current}$;
10: $Push\ node \to parent\ to\ set_{current}$;
11: **end for**
12: $Research(node)$
13: **end function**

Algorithm 3 Research the selected nodes

function RESEARCH(node)
    $substruct_1 = OPT(node \to left \to left) + OPT(node \to left \to right) + OPT(node \to right \to left) + OPT(node \to right \to right) + v_{node}$;
    if $OPT(node) == substruct_1$ **then**
        $Push\ node\ to\ set_{selection}$;
        $Research(node \to left \to left)$;
        $Research(node \to left \to right)$;
        $Research(node \to right \to left)$;
        $Research(node \to right \to right)$;
    else
        $Research(node \to left)$;
        $Research(node \to left)$;
    end if
end function

2.3 Proof of Algorithm Correctness

采用数学归纳法证明。

当仅有一个节点输入时:

$$OPT(node) = max\{OPT(node \to left \to left) + OPT(node \to left \to right) \tag{11}$$

$$+OPT(node \to right \to left) + OPT(node \to right \to right) + v_{node}, OPT(node \to left) + OPT(node \to right)\} = max\{v_{node}, 0\} = v_{node}$$

假设二叉树的 $root \to left$,$root \to right$,$root \to left \to left, root \to left \to right, root \to right \to left, root \to right \to right$ 节点已知，则:

$$OPT(root \to left) = max\{OPT(root \to left \to left \to left) + OPT(root \to left \to left \to right) \tag{12}$$

$$+OPT(root \to left \to right \to left) + OPT(root \to left \to right \to right) + v_{root \to left}, OPT(root \to left \to left) + OPT(root \to left \to right)\}$$

$$OPT(root \to right) = max\{OPT(root \to right \to left \to left) + OPT(root \to right \to left \to right) \tag{13}$$

$$+OPT(root \to right \to right \to left) + OPT(root \to right \to right \to right) + v_{root \to right}, OPT(root \to right \to left) + OPT(root \to right \to right)\}$$

同理可得 $OPT(root \to left \to left), OPT(root \to left \to right), OPT(root \to right \to left), OPT(root \to right \to right)$

则计算以 root 为根的二叉树的 OPT 值为:

$$OPT(root) = max\{OPT(root \to left \to left) + OPT(root \to left \to right) \tag{14}$$

$$+OPT(root \to right \to left)+OPT(root \to right \to right)+v_{root}, OPT(root \to left)+OPT(root \to right)\}$$

综上所述，算法是正确的。

2.4 Complexity Analysis

假设二叉树的节点个数为 n，OPT 数组中最优解的个数为 n，每次分解成两个子问题，第一个子问题需要查找 4 个孙子节点的 OPT，第二个子问题需要查找 2 个子节点的 OPT，因此每次遍历 6 个节点，算法的时间复杂度为:

$$T(n) = (4 + 2) * n = O(n) \tag{15}$$

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.

3.1 Optimal Substructure and DP Equation

设 $Num(\{e_1, e_2, ..., e_n\})$ 为一串编码后的数字的解码方式总个数，编码数字的个数为 n，利用 DP 方法，考虑每增加一个数字后，最后一个数字是否可以和前一个数字组成一对数字进行编码，因此问题可以分成两个子问题: 当前一个数字 $e_{n-1} \leq 2$ 时，则可以和最后一个数字组成一对进行编码，则其总的解码方式为不加这一对数字的解码和 $Num(\{e_1, e_2, ..., e_{n-2}\})$ 与加上第 n-1 个数之后的解码和 $Num(\{e_1, e_2, ..., e_{n-1}\})$; 当前一个数字 $e_{n-1} > 2$ 时，则说明其不能与数字 e_n 组成一对进行编码，则其总的解码方式即为前 n-1 个数的解码方式总和。此问题的 DP Equation 如公式 (16) 所示。

$$Num(\{e_1, e_2, ..., e_n\}) = \begin{cases} Num(\{e_1, e_2, ..., e_{n-2}\}) + Num(\{e_1, e_2, ..., e_{n-1}\}) & 0 < e_{n-1} \leq 2 \\ Num(\{e_1, e_2, ..., e_{n-1}\}) & e_{n-1} > 2 \end{cases} \tag{16}$$

3.2 Algorithm Description

首先初始化 1 个数时，解码方式为 1，当解码多个数字时，将解码总数分解成两部分: 如果前一个数小于等于 2，则包含前一个数和不包含前一个数的编码方式之和即为加上最后一个数的总的编码方式; 如果前一个数大于 2，则最后一个数不能与前一个数组成一对，其编码方式之和即为前 i-1 个数的编码总和。循环 i 直到 n，求得 $Num(\{e_1, e_2, ..., e_i\})$。

3.3 Proof of Algorithm Correctness

采用数学归纳法证明。

当 n=1 时，则有:

$$Num(\{e_1\}) = 1 \tag{17}$$

假设 n=k-1 和 k-2 时，可得: 当 $e_{k-2} \leq 2$ 时，

$$Num(\{e_1, e_2, ...e_{k-1}\}) = Num(\{e_1, e_2, ..., e_{k-3}\}) + Num(\{e_1, e_2, ..., e_{k-2}\}) \tag{18}$$

当 $e_{k-2} > 2$ 时，

$$Num(\{e_1, e_2, ..., e_{k-1}\}) = Num(\{e_1, e_2, ..., e_{k-2}\}) \tag{19}$$

Algorithm 4 Determine the total number of ways to decode an encoded digits

function DECODING($\{e_1, e_2, ..., e_n\}, n$)
  $Num(\{e_1\}) = 1$;
  for i from 2 to n **do**
    if $i == 2$ **then**
      $Num(\{e_1, e_2, ..., e_{i-2}\}) = 0$
    end if
    if $e_{i-1} \leq 2$ **then**
      $Num(\{e_1, e_2, ..., e_i\}) = Num(\{e_1, e_2, ..., e_{i-2}\}) + Num(\{e_1, e_2, ..., e_{i-1}\})$;
    else
      $Num(\{e_1, e_2, ..., e_i\}) = Num(\{e_1, e_2, ..., e_{i-1}\})$;
    end if
  end for
end function

同理可得到 $Num(\{e_1, e_2, ...e_{k-2}\})$ 的值。

当 n=k 时，则有: 当 $e_{k-1} \leq 2$ 时，

$$Num(\{e_1, e_2, ...e_k\}) = Num(\{e_1, e_2, ..., e_{k-2}\}) + Num(\{e_1, e_2, ..., e_{k-1}\}) \tag{20}$$

当 $e_{k-1} > 2$ 时，

$$Num(\{e_1, e_2, ..., e_k\}) = Num(\{e_1, e_2, ..., e_{k-1}\}) \tag{21}$$

综上所述，算法是正确的。

3.4 Complexity Analysis

设编码数字共 n 个，因此 Num 数组共 n 个元素存放 $Num(\{e_1, e_2, ..., e_n\})$ 的值，每次求解 $Num(\{e_1, e_2, ..., e_n\})$ 时，每次将问题分成两个子问题，第一个子问题需要寻找前两个 Num 值，第二个子问题则需要寻找前一个 Num 值。每次循环只挑选一个子问题，因此算法的时间复杂度为:

$$T(n) \leq 2 * n = O(n) \tag{22}$$

4 Longest Consecutive Subsequence

You are given a sequence of integers L and an integer k, your task is to find the longest consecutive subsequence the sum of which is the multiple of k.

4.1 Optimal Substructure and DP Equation

首先问题可以看成两部分，如果前 n 项数字和 S_n 模 k 的余数为 0，则说明前 n 项即是连续的子串; 如果前 n 项和模 k 的余数不为 0，取每一个余数第一次出现在 L 中的位置 $N_1[S_n\%k]$，考虑同余定理，如果前 n 项和模 k 的余数与第一次余数相同，则余数第一次出现的位置到 n 之间的和模 k 的余数一定为 0，则为 k 的倍数，每次取每一个余数不加第 n 项前的子串长度 $N_{n-1}[S_n\%k]$ 与加上第 n 项后第 n 项到第 1 次出现该余数位置之间的长度 $n - N_1[S_n\%k]$ 的较大者，最终统计所有小于 k 的余数下长度的最大值即为解。问题的 DP Equation 如下所示:

$$N_{\{0,1,...,n\}}[\{0,1,...,k-1\}] = max \begin{cases} n & S_n\%k = 0 \\ max\{N_{n-1}[S_n\%k], n - N_1[S_n\%k]\} & S_n\%k \neq 0 \end{cases} \tag{23}$$

$$S_n = L_1 + L_2 + ... + L_n \tag{24}$$

4.2 Algorithm Description

首先 DP 算法维护两个数字 N_1 和 N_i，N_1 表示余数第一次出现时的位置，N_i 表示该余数下当前子串最大长度。首先为 N_1 初始化赋值为 0，然后从 1 到 n 循环，每加上一个数字，计算其模 k 的余数是否为第一次出现，第一次出现则更新 N_1 否则更新最大长度数组 N_i，更新方式依照问题分成两部分: 当所有和是 k 的倍数是，余数 0 的最大长度即为 i，记录数组长度; 当模 k 为其他余数时，将当前长度减去第一次余数出现时的长度，并与当前余数下的原 N 作比较，较大者更新当前 N，记录其数组位置。最后统计所有余数下的长度的最大值，返回其数组位置。本题算法在下页展示。

4.3 Proof of Algorithm Correctness

当 n=1 时，则有

$$N_1 = 0(S_n\%k \neq 0)$$
$$N_1 = 1(S_n\%k == 0)$$

假设 n=k-1，可得

$$N_{k-1} = max\{N_{k-1}[0], N_{k-1}[1], ...N_{k-1}[k'-1]\}$$

当 n=k 时，则

$$N_k = max\{N_k[0], N_k[1], ...N_k[k'-1]\}$$

而对于每一个 $N_k[i](i < k'-1)$(其中 k' 为倍数)，每次由 $max\{N_{k-1}[i], k - N_1[i]\}$ 可得，又其中 $N_1[i]$ 已知，$N_{k-1}[i]$ 由 n=k-1 假设可得，则 N_k 可计算。

综上所述，算法是正确的。

4.4 Complexity Analysis

算法分成两个子问题，每次寻找两个数组即第一次余数出现位置 N_1 和前一次余数下的最大长度记录 N_{i-1}，因此算法的时间复杂度为

$$T(n) = 2 * n = O(n) \tag{25}$$

Algorithm 5 Find the longest consecutive subsequence the sum of which is the multiple of k.

function LONGESTSUBSEQUENCE($L(L_1, L_2, ..., L_n), n, k$)
  for i from 0 to k-1 **do**
    $N_1[i] = 0$;
  end for
  if $n == 1$ **then**
    if $L_1\%k == 0$ **then**
      return 1;
    end if
    return 0;
  end if
  for i $from$ 1 to n **do**
    $S_n = S_n + L_i$;
    if $N_1[S_n\%k] == 0$ **then**
      $N_1[S_n\%k] == i$;
    else
      if $S_n\%k == 0$ **then**
        $N_i[0] == i$;
      else
        $N_i[S_n\%k] == max\{N_{i-1}[S_n\%k], i - N_1[Sn\%k]\}$;
      end if
    end if
  end for
  $return$ $max\{N_n[0], N_n[1], ...N_n[k-1]\}$;
end function

5 Maximum profit of transactions

You have an array for which the i-th element is the price of a given stock on day i.

Design an algorithm and implement it to find the maximum profit. You may complete at most two transactions.

Note: You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again), and one transaction includes buying and selling.

5.1 Optimal Substructure and DP Equation

设 $buy_1[n], buy_2[n], sell_1[n], sell_2[n]$ 分别为第 n 天第一次买入、第 n 天第一次卖出、第 n 天第二次买入、第 n 天第二次卖出的收益，$price[n]$ 表示第 n 天的股票交易额，为使最终 $sell_2[n]$ 的收益最大，此问题的 DP Equation 如下所示:

$$
\begin{aligned}
sell_2[n] &= max\{sell_2[n-1], price[n] + buy_2[n-1]\} \\
buy_2[n] &= max\{buy_2[n-1], sell_1[n-1] - price[n]\} \\
sell_1[n] &= max\{sell_1[n-1], price[n] + buy_1[n-1]\} \\
buy_1[n] &= max\{buy_1[n-1], -price[n]\}
\end{aligned}
$$

5.2 Algorithm Description

算法首先初始化 $buy_1[1], buy_2[1], sell_1[1], sell_2[1]$ 的值，循环从 2 到 n 开始，每次将前 i-1 天的收益与与第 i 天的收益相比取最大者，直到循环结束到第 n 天，返回前 n 天的 $sell_2[n]$ 即为两次交易的最大收益。

Algorithm 6 Find the maximum profit transaction way.

```
1:  function MaximumProfit(price(1, 2, ..., n), n)
2:      buy_1[1] = buy_2[1] = −price[1];
3:      sell_1[1] = sell_2[1] = 0;
4:      for i from 2 to n do
5:          sell_2[i] = max{sell_2[i − 1], price[i] + buy_2[i − 1]}
6:          buy_2[i] = max{buy_2[i − 1], sell_1[i − 1] − price[i]}
7:          sell_1[i] = max{sell_1[i − 1], price[i] + buy_1[i − 1]}
8:          buy_1[i] = max{buy_1[i − 1], −price[i]}
9:      end for
10:     return sell_2[n];
11: end function
```

5.3 Proof of Algorithm Correctness

采用数学归纳法证明。

当 n=1 时，则有

$$sell_2[1] = 0$$

假设 n=k-1，则可得到

$$
\begin{aligned}
sell_2[k-1] &= max\{sell_2[k-2], price[k-1] + buy_2[k-2]\} \\
buy_2[k-1] &= max\{buy_2[k-2], sell_1[k-2] - price[k-1]\} \\
sell_1[k-1] &= max\{sell_1[k-2], price[k-1] + buy_1[k-2]\} \\
buy_1[k-1] &= max\{buy_1[k-2], -price[k-1]\}
\end{aligned}
$$

当 n=k 时，则

$$
\begin{aligned}
sell_2[k] &= max\{sell_2[k-1], price[k] + buy_2[k-1]\} \\
buy_2[k] &= max\{buy_2[k-1], sell_1[k-1] - price[k]\} \\
sell_1[k] &= max\{sell_1[k-1], price[k] + buy_1[k-1]\} \\
buy_1[k] &= max\{buy_1[k-1], -price[k]\}
\end{aligned}
$$

又 price[k] 及 price[k-1] 均已知，则 $sell_2[k]$ 可得。

综上所述，算法是正确的。

5.4 Complexity Analysis

算法中每个数组长度为 n，算法使 i 从 2 循环到 n，每次查找 $buy_1, buy_2, sell_1, , sell_2, price$ 五个数组值。因此，该算法的时间复杂度:

$$T(n) = 5 * n = O(n) \tag{26}$$