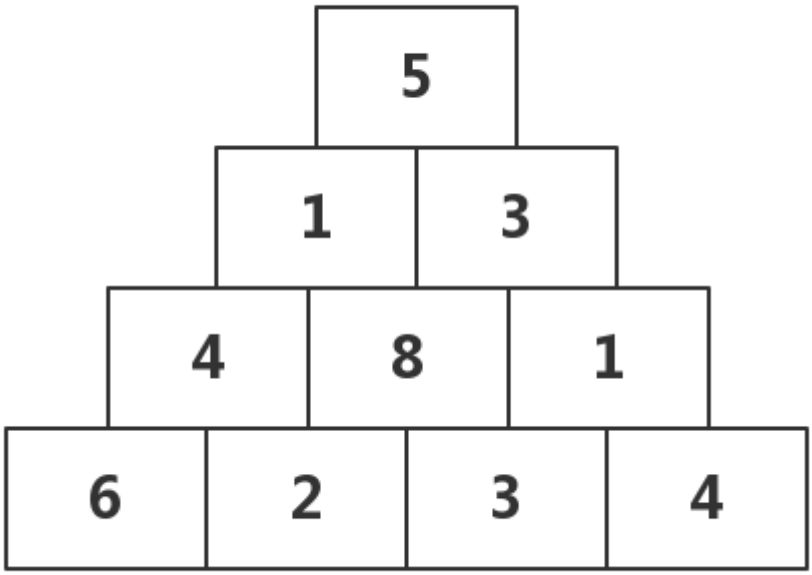
The Miner problem

本报告在参考了刘同学课堂上的讲授的基础上完成，报告内容，包括文中的图片完全由本人原创。

Problem Description

在一座金字塔每块石头上都镶有一块钻石从金字塔的顶端向下收集钻石尽可能收集价值高的钻石，每层每次只能从一块砖斜向左下或斜向右下走到另一块砖上。设计算法求出获取的钻石最大价值及路径。

一个样例如下：



Problem Analysis

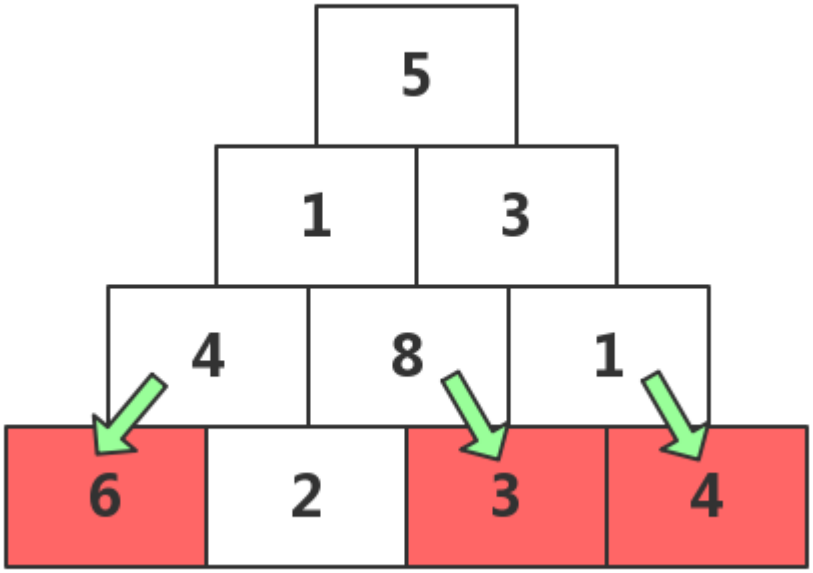
从金字塔顶端到底端的路径，每一层有2种选择，因此总路径数会是 2^{n-1} 之多。如果使用最粗糙的做法，对于一个 n 层的金字塔，我们需要遍历每条路径（长度为 n), 因此时间复杂度会达到令人难以接受的 $O(n2^n)$。

我们分析这样高的时间复杂度的来源。可以注意到，这些路径具有相当多的公共部分，如果能设计一种聪明的算法，我们将不必重复计算这些公共部分，但很遗憾，在上面的做法中，计算每一条路径的价值时我们都进行了一次完整的计算。这是非常大的开销，也是我们之后希望避免的。

这就进入了**动态规划**发挥作用的时候。先做一些约定以方便之后的探讨：我们对金字塔的层数是从上往下计算的，也就是说，最顶层是第一层，而出口层是第 n 层。

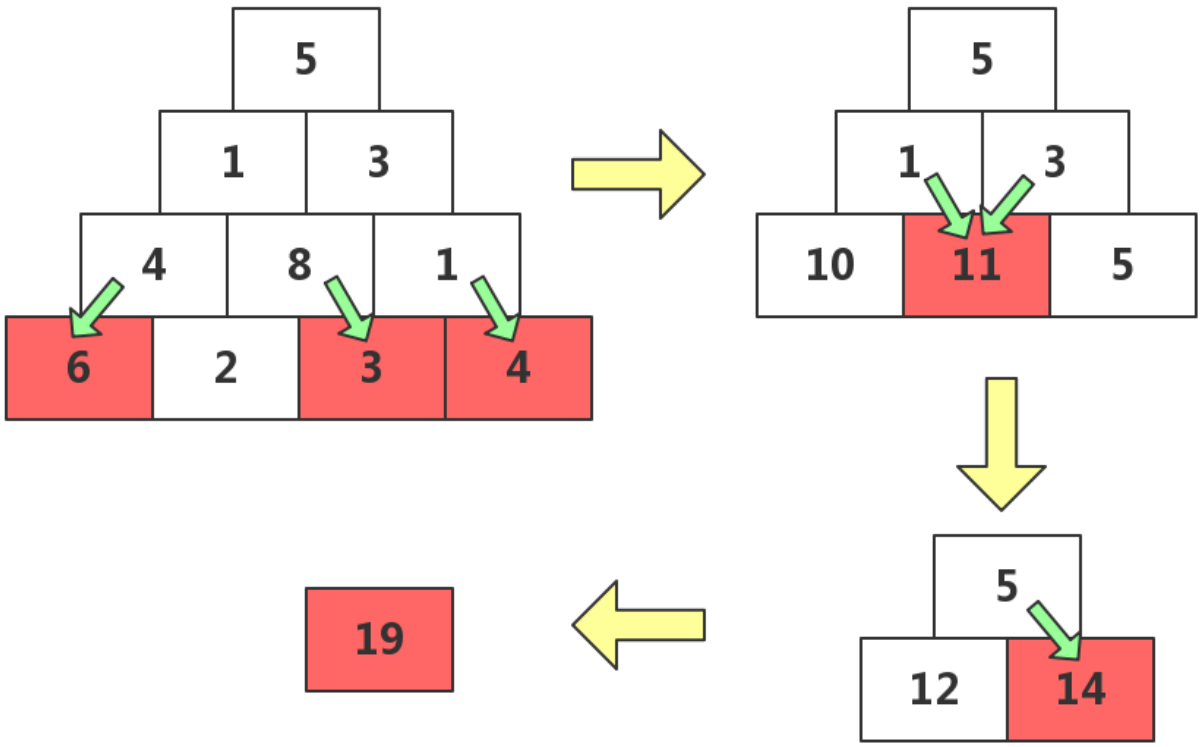
动态规划的关键是找到最优子结构，即**原问题的最优解可以通过找到它的子问题的最优解来解决**。在我们这个例子里，如果从上往下寻找是无法找到这样的性质的：在第i层你获得了一个最优解，但是对于 $i+1$ 层的问题这个解往下找到出口却并不一定是最优解，因为还得取决于第 $i+1$ 层钻石的分布。

那么我们换个思路，从下往上求解问题。 拿上面这个图作为例子，当你到达第三层，也就是还差一层就到达出口的时候，你要如何选择路径？如果你在$[3,1]$（即 4 这个位置），那么选择出口1显然是更优的结果；而如果你在$[3,2]$,那么选择出口3,会是更好的选择；同理在$[3,3]$，我们应该选择出口 4。**这意味着当你到达最后一层时，原本需要检查的6种情况，被筛选成了3种。**如下图：

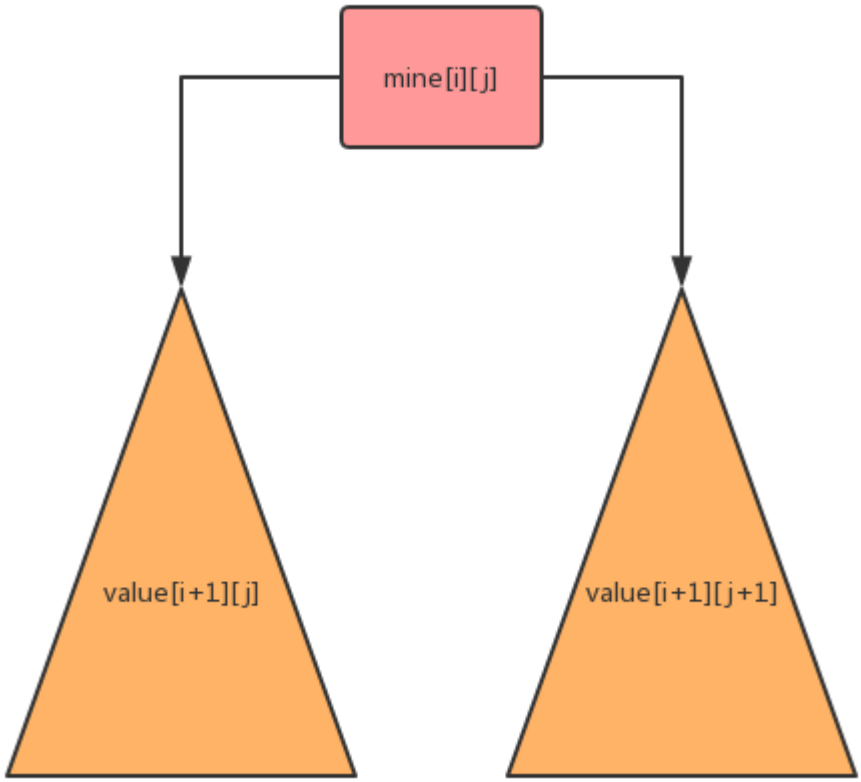


继续往上。当你到达第二层，

即将进入第三层时，你要如何选择路径？根据我们前面的分析，选择$[3,1]$之后能增加的价值是 10, 选择 $[3,2]$ 能增加 11, 选择 $[3,3]$ 能增加 5: 这就相当于我们去掉了最后一层，并且把第三层里的数字$4,8,1$ 分别更新为了 $10,11,5$ 。这样，**我们就能用刚才同样的方法来选择路径了。** 如此重复，更新，直到到达第一层，我们将获得唯一的最大价值。如下图：



因此我们知道其中的递推关系。 我们用`mine`来表示这个金字塔，则`mine[i][j]`表示第`i`层第`j`格中的钻石价值，用`value[i][j]`表示以当前格子为起点能获得的最大价值。则其图示如下：



可知：$value[i][j] = mine[i][j] + max(value[i+1][j], value[i+1][j+1])$

这样我们求得了动态规划问题中求解最优解的递推式，整个问题的最优解即为$value[1][1]$。

Python implementation

User input

```
def input_data():
    '''
    load the data via keyboard input.
    '''
    input_n = input('input the height of triangle:')
    n = int(input_n)

    mine = []
    for i in range(n):
        layer = []
        print('--------input the layer ' + str(i+1) + '-----------')
        input_x = input('The mine:').split()
        #print(input_x)
        assert(len(input_x) == i+1)
```

```
        x = list(map(int, input_x))
        layer.extend(x)
        mine.append(layer)

    return mine
```

Print Path

这里只打印其中一条路径。对于一个6层的金字塔，打印输出结果格式：

```
[a,b,c,d,e]
```

代表第二层到第六层的选择分别为a,b,c,d,e。

```
def print_path(path):
    '''
    Here we only print one possible path but not every.
    '''
    n = len(path)
    opt = [path[0][0]]
    choice = opt[0]
    for i in range(1, n-1):
        choice = path[i][choice]
        opt.append(choice)

    return opt
```

Solving the Problem

```
def miner_solve(mine):
    '''
    parameters:

    mine -- the value of each cell

    returns:

    max_value -- the max value you can obtain
    '''

    n = len(mine)

    # ATTENTION: we have to copy like these below
    # if you use value = list(mine)
    # the list mine will change as value changes
    value = []
    path = []
    for i in range(n):
        value.append(list(mine[i]))
        path.append(list(mine[i]))
```

```
    for i in reversed(range(n - 1)):
        m = len(value[i])
        for j in range(m):
            value[i][j] = mine[i][j] + max(value[i+1][j], value[i+1][j+1])
            if value[i+1][j] >= value[i+1][j+1]:
                path[i][j] = j
            else:
                path[i][j] = j+1


    opt = print_path(path)

    return value[0][0], opt
```

Example 1

这里的例子仅为参考，并非正式的测试。

第一个例子即上面我们所举的例子。

```
mine = [[5],
       [1,3],
       [4, 8 ,1],
       [6, 2, 3, 4]]
```

```
mine = input_data()
```

```
input the height of triangle:4
--------input the layer 1-----------
The mine:5
--------input the layer 2-----------
The mine:1 3
--------input the layer 3-----------
The mine:4 8 1
--------input the layer 4-----------
The mine:6 2 3 4
```

```
max_value, opt = miner_solve(mine)
```

```
max_value
```

```
19
```

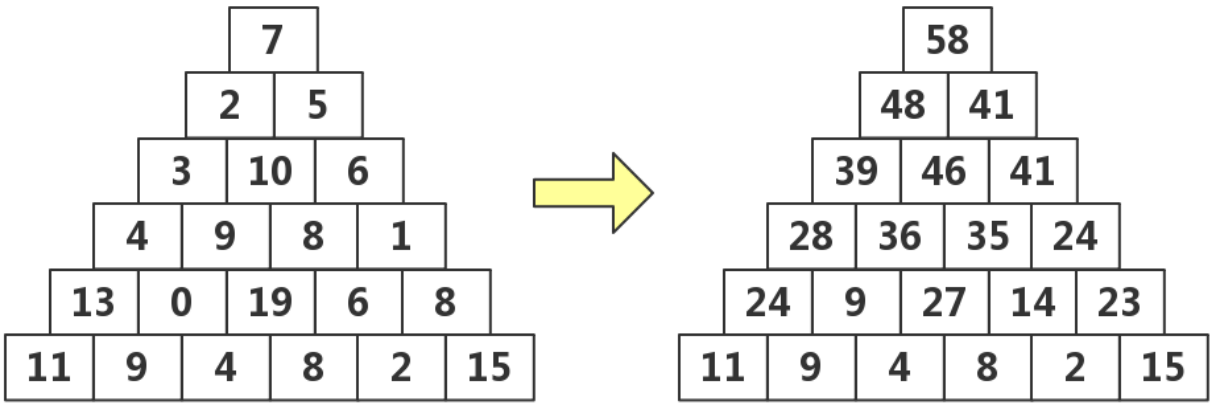
```
opt
```

```
[1, 1, 2]
```

矿工问题的测试很难通过验证型程序来进行测试。因此这里我们仅构造数个不同规模的问题进行手动验证。

Example 2

如下图所示，右侧为手工求解结果：




```
mine = [[7],
        [2,5],
        [3, 10 ,6],
        [4, 9, 8, 1],
        [13, 0, 19, 6, 8],
        [11, 9, 4, 8, 2, 15]]
```

```
max_value, opt = miner_solve(mine)
```

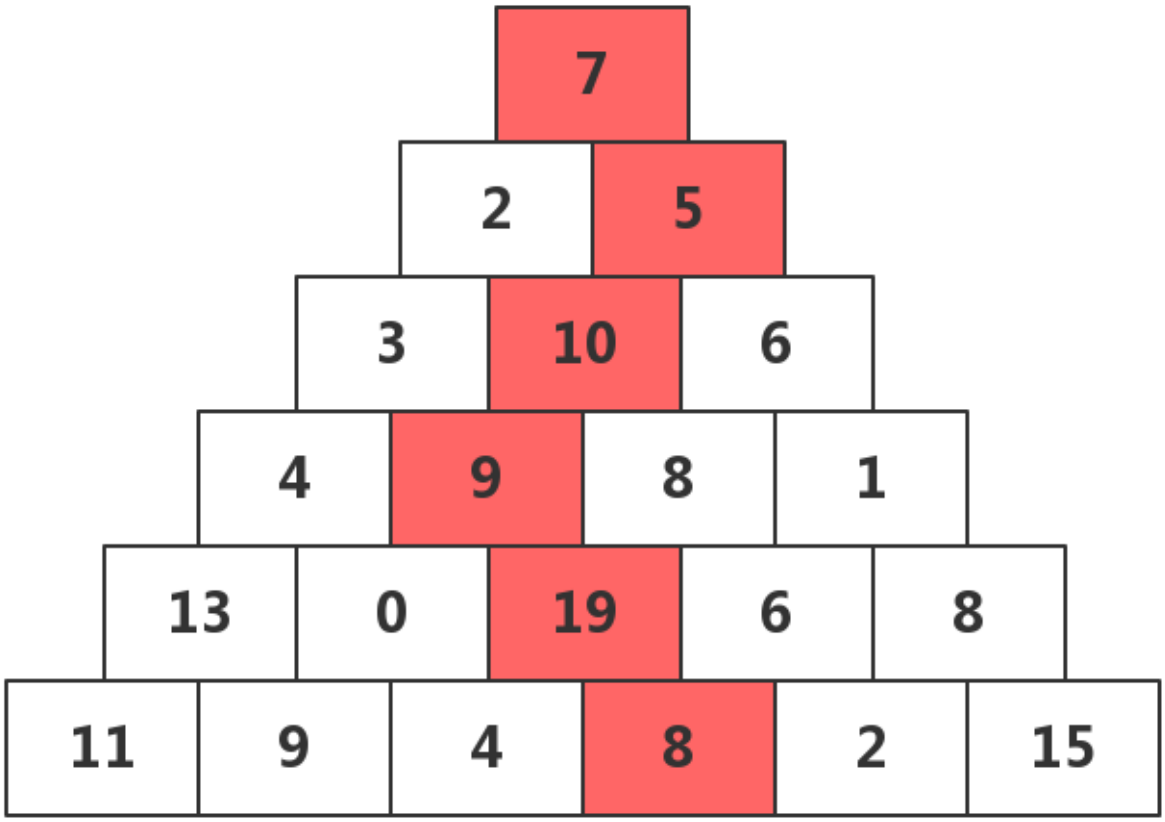
```
max_value
```

```
58
```

```
opt
```

```
[1, 1, 1, 2, 3]
```

把我们的结果可视化，得到：



结果正确。

Other Examples

我们依次进行了6，15，20,40层的测试，结果均正确。限于篇幅，这里不再罗列。

Correctness Test

Test code

```
def max_mine(mine, value, i, j):
    n = len(mine)
    if(i < n -1):
        return mine[i][j] + max(max_mine(mine, value, i+1, j), max_mine(mine, value, i+1, j+1))
    else:
        return mine[i][j]
```

```
def max_gold(mine):

    value = []
    n = len(mine)
    for i in range(n):
        value.append(list(mine[i]))

    return max_mine(mine,value,0,0)
```

```
mine = [[7],
       [2,5],
       [3, 10 ,6],
       [4, 9, 8, 1],
       [13, 0, 19, 6, 8],
       [11, 9, 4, 8, 2, 15]]
```

```
max_gold(mine)
```

```
58
```

```
mine2 = [[5],[1,3],[4,8,1],[6,2,3,4]]
max_gold(mine2)
```

```
19
```

Test data

```
import numpy as np
```

```
def generate_data(n):
    data = []
    for i in range(n):
        dat = list(np.random.randint(50, size = i+1))
        data.append(dat)

    return data
```

```
generate_data(10)
```

```
[[24],
 [18, 37],
 [49, 14, 19],
 [17, 41, 20, 2],
 [11, 18, 22, 35, 5],
 [33, 49, 12, 30, 32, 0],
 [19, 19, 36, 18, 5, 35, 20],
 [36, 6, 22, 30, 40, 17, 10, 42],
 [27, 16, 12, 37, 30, 4, 48, 9, 10],
 [44, 44, 33, 42, 29, 13, 33, 22, 35, 4]]
```

Test

```
test_size = [4, 6, 10, 15, 20,25]

for size in test_size:
    data = generate_data(size)
    max_value_1, path = miner_solve(data)
    max_value_2 = max_gold(data)

    if max_value_1 == max_value_2:
        print('-----on size:'+ str(size) + ' correct-------------')
    else:
        print('-----on size:'+ str(size) + ' incorrect-------------')
```

```
-----on size:4 correct-------------
-----on size:6 correct-------------
-----on size:10 correct-------------
-----on size:15 correct-------------
-----on size:20 correct-------------
-----on size:25 correct-------------
```

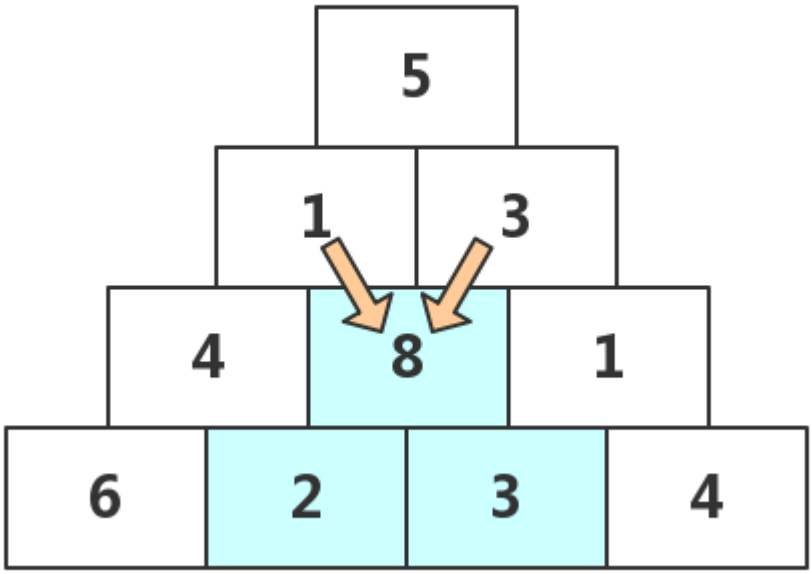
因此我们验证了代码的正确性。

Time Complexity Analysis

很容易分析这个问题的时间复杂度。

代码中包含两个 `for` 循环，因此时间复杂度为 $O(n^2)$。

考虑与分治法的对比。我们知道**动态规划**是自底向上的，而**分治法**是自顶向下的；因此**动态规划能避免大量的重复计算。** 这个问题中分治法的递推式与动态规划相同，因此时间复杂度也相同，但是其中会包含重复计算。举例如下：



计算 1 和 3 处的 `value` 值时，蓝色部分要重复计算。而这样的重复计算在动态规划中是成功避免了的。

上面我们的验证算法实际上就是分治法。 经过分析, 这里采用分治实际上**遍历了每一条路径**，因此算法复杂度为 $O(2^n)$。这说明在把原问题转为子问题时，分治法并没有提供简化问题的方法；而在动态规划中，由于避免了重复计算，算法复杂度被大大优化。