一、实验题目与要求

1. 0/1 Knapsack Problem. There are 5 items that have a value and weight list below, the knapsack can contain at most 100 Lbs. Solve the problem using back-tracking algorithm and try to draw the tree generated.

Value($)	20	30	65	40	60
Weight(Lbs)	10	20	30	40	50
Value/Weight	2	1.5	2.1	1	1.2

2. Solve the 8-Queen problem using back-tracking algorithm.

二、算法思想

1. Knapsack Problem

回溯法需要模拟每一种实际情况，并在实际情况不满足要求时回退，继续尝试其他情况。 对于 0-1 背包问题，所尝试的情况即为“放”与“不放”。

然而，对于有 n 个物品的背包问题，如果模拟所有的情况，则需要枚举 2^n 种，从而导致算法复杂度急剧上升。因此需要定义边界函数：边界函数用于计算在当前情况下（可能已经拿 取了部分物品，使用了一部分背包空间），若继续拿取物品，能取得的最大价值。如果该情况下能取得的最大价值已经小于目前计算得到的最大价值（目前计算得到的最大价值不一定是 最优解），则该情况不可能是最优解，从而无需继续后面的迭代，剪掉了一部分活节点。

边界函数使用类似于部分背包问题的算法来计算，因此要求提供的物品按照价值/重量（单位重量的价值）从大到小排序。利用贪心算法的思想，每次都选择单位质量的价值最大的物品， 如果不足以放入整个物品，则放入部分物品。这得到的将是该情况下该背包所能填充的最大价值。而对于 0-1 背包问题，最终解一定不会超过该值。

2. 8-Queen

n皇后问题对皇后的布置有着较多的限制，且本身不是最优解问题，而是可行解问题。使用回溯算法解决此问题时，逐行测试每一个皇后的位置。如果当前位置不满足要求，则测试下 一位置，直到本行的所有位置都测试完成后，回退到上一行继续下一个位置。若所有行都测试 完毕（均到达下标 n），则所有可行解都寻找结束。

三、算法步骤与核心代码

1. Knapsack Problem

1. Knapsack Problem

该算法实现于0-1背包问题.py中。

首先记录条件：

```
val=[20,30,65,40,60]
wei=[10,20,30,40,50]
```

接下来我用dp数组来记录已经计算出的结果。一级下标指的是背包目前（放了一定东西后）还能承受的重量，二级下标指的是目前val数组遍历到的下标。这样的好处是可以少进行不必要的重复计算，节省了时间（但会浪费一定的空间）。

这里有个小细节：dp数组初始化全部为-1。这样的目的是便于判断是否已经计算过（因为若计算过的话最小的值也是0），不用再另开一个对应的book数组记录，节省了空间：

```
dp=[[-1 for i in range(6)]for i in range(101)]
```

然后，我用con代表背包当前的容量，t代表目前遍历到的下标值，那么根据背包能否装下当前的物品分为两种情况，其中能装下时取装它和不装它最后结果的较大值，核心代码如下：

```
    if con<wei[t]:dp[con][t]=rec(con,t+1)
    else:dp[con][t]=max(val[t]+rec(con-wei[t],t+1),rec(con,t+1))
```

其中rec是递归函数，包含了上一段核心代码及一些特殊情况和边界情况的判断：

```
def rec(con,t):
    if dp[con][t]!=-1:return dp[con][t]
    if t==5:return 0
    if con<wei[t]:dp[con][t]=rec(con,t+1)
    else:dp[con][t]=max(val[t]+rec(con-wei[t],t+1),rec(con,t+1))
    return dp[con][t]
```

2. n-Queens

该算法实现于八皇后.py中。

此问题核心代码就在于回溯操作函数，核心代码如下：

```
def backtrack(row):
    if row == n:
        board = generateBoard()
        ans.append(board)
    else:
        for i in range(n):
            if i in columns or row - i in diagonal1 or row + i in diagonal2:
                continue
            queens[row] = i
            columns.add(i)
            diagonal1.add(row - i)
            diagonal2.add(row + i)
            backtrack(row + 1)
            columns.remove(i)
            diagonal1.remove(row - i)
            diagonal2.remove(row + i)
```

在回溯的基础上，若满足条件需要输出（输出时用Q代表皇后，'.'用来占位）：

```
def generateBoard():
    board = []
    for i in range(n):
        row[queens[i]] = "Q"
        board.append("".join(row))
        row[queens[i]] = "."
    return board
```

四、总结

在本次实验中，我学会了使用不同的算法解决背包问题，并学会了解决 n 皇后问题的算法思路，主要练习了回溯算法的技巧。。