Lab2

1. 问题描述和思考：

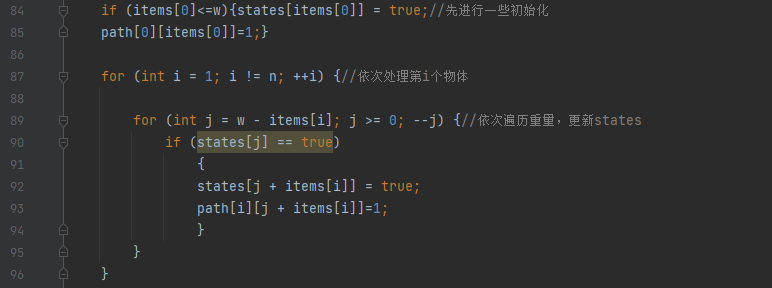
根据题目描述，感觉和背包01问题很相似，并和背包01问题进行对比：一共有N件物品，第i（i从1开始）件物品的重量为w[i]，价值为v[i]。在总重量不超过背包承载上限W的情况下，能够装入背包的最大价值是多少？

从一系列正整数的集合中挑出一些数，使得这些数的和最接近而又不超过某个给定的目标值？因此，这里重量和价值都是代表data.dat里的第二列(除第一行的)，背包上限为data.dat里的第一行第二列，求最接近这个值的数组组合。

由于发现data2的求和是1gb字节的的数，即使使用空间优化过的背包01版本，1gb数量字节的数组一个int4字节，也要4gb，这里我选用bool，缩小4倍，1gb。

1. 代码主要逻辑描述：

考虑到传统的二维数组记录有优化空间，本质就是二维数组的第一维度每次循环都加一，可以直接用一维数组进行元素覆盖，之前的数据不再需要，下一循环的数只需要上一循环即可，可以将空间复杂度除以n。代码中即为states[]的一维数组。其中states[i]表示在一个特定的循环中重量为i的情况是否可以到达，真为true。而外部循环每次对第i个物体进行处理，对全部重量进行逐一遍历，如果对于重量w,w-items[i](物品i的重量)为真(states[w-items[i]]=1，那么加上这个物体显然也为真，states[w]=1;直到处理完最后一个物体终止，然后从后向前遍历数组，找到最大的为真（可以到达的weight）即可。



之后是记录哪些物体被选中，一开始我的想法是直接对一维的数组states进行包装成一个自定义类，让它可以额外记录这个节点是由那些前继加和而成，但是这个对于data2的情况也会崩溃，原因和使用int类型一致，总weight过大。

因此我使用二维bool数组记录，path[i][index]表示表示到达 index 这个重量的时候是否使用了第 i 个数，那么如果使用了，就可以向前回溯，path[i-1][index-item[i]]，如果没有，继续循环判断第i-1个数即可。

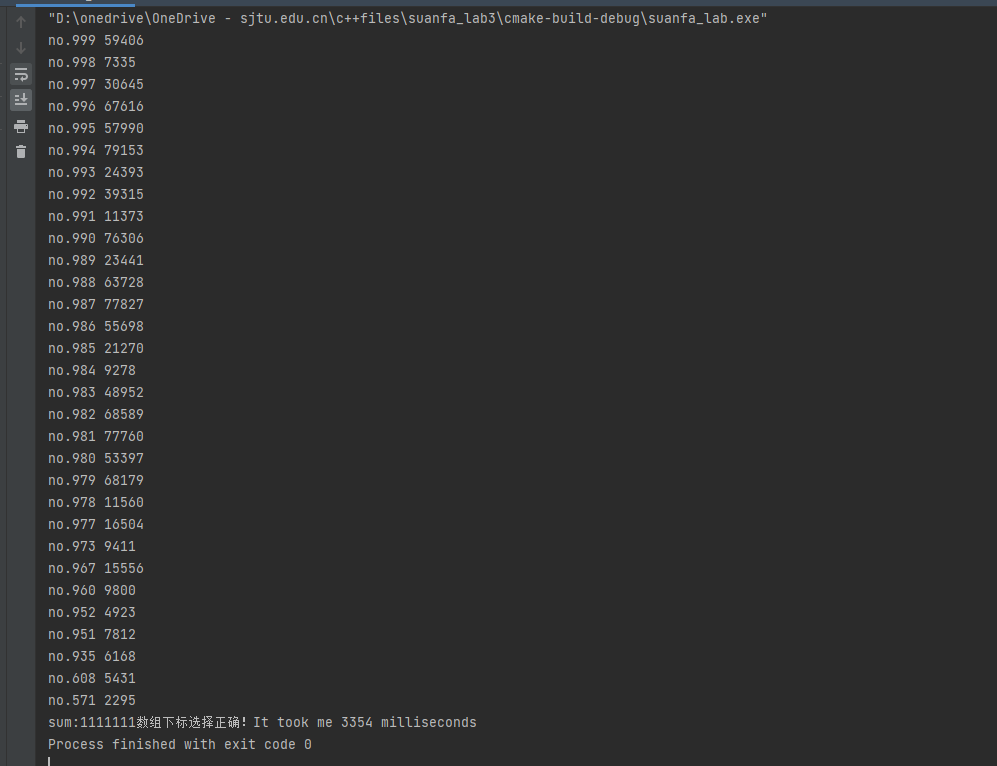


3 结果：

Data1:

Setting:



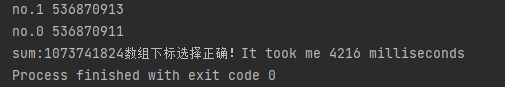


这里对选出的数据进行求和验证，这里我尝试了不验证，减少的时间比每次随机运行不一样的时间差还小，所以这里对时间比较没有影响。

Data2:

Setting:



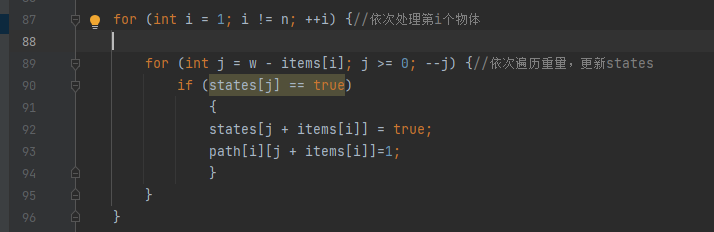


可见这种重量很大的情况也可以比较好的处理，和data1仅差了1s左右。但由于仅3个数据，由此也可见这种情况下的dp局限。

4.复杂度

时间：

States更新为dominated：



所以为O(物体数量乘以目标重量)

空间：

Path为dominated:

也是O(物体数量乘以目标重量)

5 两个数据集的比较：

运行时间：

如前所述，虽然data2数据集少，但是dp算法并“不智能“，慢于1000个数据处理1s左右的原因在于89行依次遍历重量：



由于目标以及每个数都很大，数之间的间隔也很大，这就意味着遍历1gb的次数时，states大部分元素都是false，states的true太稀疏，程序大部分时间在空循环，但是这也是dp的特点，枚举所有情况，保证最优解。

所需内存空间：

O(物体数量乘以目标重量)

Data1为1000\*1111111 10位数

Data2为3\* 1073741824 10位数

可见差不多

附：

和第一次实验一样，助教如果复现代码，要重新设置文件路径

