一、实验题目与要求

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 both as fractional knapsack and 0/1 knapsack.

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

2. A simple scheduling problem. We are given jobs $j_1, j_2, \ldots, j_n$ all with known running times $t_1, t_2, \ldots, t_n$ respectively. We have a single processor. What is the best way to schedule these jobs in order to minimize the average completion time. Assume that it is a nonpreemptive scheduling: once a job is started, it must run to completion. The following is an instance.

(a) $(j_1, j_2, \ldots, j_n)$：(15, 8, 3, 10)。

3. Single-source shortest paths. The following is the adjacency matrix, vertex A is the source.

	A	B	C	D	E
A		-1	3		
B			3	2	2
C					
D	1	5			
E			-3		

4. All-pairs shortest paths. The adjacency matrix is as same as that of problem 3. (Use Floyd or Johnson's algorithm)

二、算法思想

1. Knapsack Problem

部分背包问题允许部分地拿取一件物品，其核心思想为贪心算法：每次都选择性价比最高（单位重量的价值最大）的物品，如果该物品不能完全放入背包，则放入部分。

0-1 背包问题可以使用动态规划求解：令 $c[i,j]$ 表示当背包容量为 j 时，选择前 i 件物品，能取得的最大价值；若选择第 i 件物品，其价值为 v_i，重量为 w_i，则背包中物品的价值总和为 $c[i-1, j-w_i]+v_i$，即前 $i-1$ 件物品装入背包，且第 i 件物品装入背包，所得到的价值综合；若不选择第 i 件物品，则背包的价值即为 $c[i-1, j]$。因此动态规划转移方程为

$$c[i,w]=\begin{cases}0, & i=0\text{或}w=0,\\ c[i-1,w], & w_i>w,\\ \max(c[i-1,w-w_i]+v_i, c[i-1,w]), & i>0\text{且}w\geq w_i\end{cases}$$

2. A simple scheduling problem

给定多件工作及其所需的运行时间，若工作依次顺序进行，且不能打断（抢占），则若这些工作都在 0 时刻到达，为了使平均完成时间最少，只需要将耗时最短的工作先完成即可。

可以这样思考该算法的正确性：若运行时间最少的任务不最先执行，而是位于第 i 个位置执行，则将该任务与队列起始位置的任务交换顺序，此时从 $i+1$ 开始的任务都不需要移动位置，但位置 i 前的任务整体前移，平均完成时间减少，因此执行时间最短的任务必然位于最开始的位置。

3. Single-source shortest paths

该题目可以使用 Dijkstra 算法和 Bellman-Ford 算法完成，但由于题给图中存在负权重的边，因此只能使用 Bellman-Ford 算法完成。

定义数组 d，其中 $d[n]$ 表示从起始节点到索引为 n 的节点的最短路径长度。对于节点数为 n 的图，该算法需要迭代 $n-1$ 次，每次都以相同的顺序遍历每一条边：对于边 (u, v)，$d[v] = \min(d[v], d[u] + w(u, v))$，即每次都检查从任意节点到节点 v 的路径是否可以更短。

迭代结束时，若仍然存在边 (u, v) 满足 $d[v] = \min(d[v], d[u] + w(u, v))$，则表示存在负权重的回路，此时最短路径不再有意义，算法将终止。

4. All-pairs shortest paths

Floyd-Warshall 算法是一种动态规划算法，定义 $c_{ij}^{(k)}$ 表示从节点 i 到节点 j，且路径上的所有中间节点的索引都不大于 k 的最短路径长度。要求解的结果即为 $c_{ij}^{(n)}$。

计算 $c_{ij}^{(k)}$ 时，若最短路径包含节点 k，则路径长度为 $c_{ik}^{(k-1)} + c_{kj}^{(k-1)}$；若最短路径不包含节点 k，则路径长度为 $c_{ij}^{(k-1)}$。因此动态规划转移方程为 $c_{ij}^{(k)} = \min(c_{ik}^{(k-1)} + c_{kj}^{(k-1)}, c_{ij}^{(k-1)})$。

注意到计算 $c_{ij}^{(k)}$ 时只需要用到上标为 $k-1$ 的值，因此统一计算所有相同上标的元素即可。

三、算法步骤与核心代码

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. A simple scheduling problem

该算法实现于调度问题中。

按照上文描述和证明，该算法只需要对任务按照执行时间从小到大的顺序排序即可：

```
time=list(map(int,input("请输入每个工作分别需要的时间："").split()))
print("最优的调度方案为：{}".format(sorted(time)))
```

3. Single-source shortest paths

单源最短路径的经典算法就是Dijkstra算法，它的核心在于双重循环中类似于路径压缩的操作：

```
def dijkstra(s):
    distance[s] = 0
    while True:
        v = -1
        for u in range(n):
            if not used[u] and (v == -1 or distance[u] < distance[v]):
                v = u
        if v == -1:break
        used[v] = True
        for u in range(n):
            distance[u] = min(distance[u], distance[v] + cost[v][u])
```

4. All-pairs shortest paths

多源最短路径的经典算法就是Floyd算法，它的核心代码就是简洁的三重循环：

```
for k in range(n):
    for i in range(n):
        for j in range(n):
            graph[i][j] = min(graph[i][j], graph[i][k] + graph[k][j])
```

四、总结

在本次实验中，我练习了多种解决不同问题的图算法和贪心算法，有效地熟悉和掌握了这 些算法背后的本质和实现原理。