讨论一 校车问题：

利用SPFA算法，依次枚举学校地址。设d[x][i]为学校在x时到区域i的最短路。

采用状压DP。每个点访问与否作为二进制0或1压缩，得到状态转移方程。

Dp[s][i]=min{dp[s][i],dp[s^(1<<(i-1))]+dist[k][i]}.

Dist表示两点间最短路。

初始化：dp[s][i]=dist[start][i](s==1<<(i-1))。

**讨论二 背包问题：**

①二维费用01背包

受体积限制的同时受到质量限制。初始一个(D+1)\*(C+1)的0矩阵，在之前01背包一维的基础上，方式添加一个质量维度，现在有两个维度状态转移为dp[i][j] = max(dp[i-w[i]][j-z[i]]+v[i]],dp[i][j])。代码如下

dp = [[0 for \_ in range(D +1)] for \_ in range(C + 1)]

for k in range(1,len(w)+1):

for i in reversed(range(w[k-1]-1,C+1)):

for j in reversed(range(z[k-1]-1,D+1)):

if w[k-1] <= i and z[k-1] <= j:#限制体积和质量

dp[i][j] = max(dp[i][j],dp[i-w[k-1]][j-z[k-1]]+v[k-1])

print(dp)

②分组背包问题

在每个分组内的策略有n+1个，n为组内物品个数，即不选，或者选其中的某一个。在选第i个组的物品，状态转移为d[j]=max(d[j],d[j-w[i-1][k]]+v[i-1][k])，代码如下：

w = [[2,3],[3,4],[3,4],[3,5]]

v = [[3,4],[4,5],[4,5],[4,6]]

n\_list = [2,2,2,2]

dp = [0 for \_ in range(C+1)]

for i in range(1, len(w)+1):#物品循环

for j in reversed(range(1, C+1)):#剩余体积循环

for k in range(n\_list[i-1]): #别的和0，1背包一样 就是这里枚举一下每个组内的值，在每个组内选出一个max值

if j-w[i-1][k] >= 0:

dp[j] = max(dp[j], dp[j-w[i-1][k]] + v[i-1][k])

print(dp[C])

③思路

（1）采取01背包的状态行不通的，dp[i][j]：前i个物品，总重量为j的最大价值，第二维的空间会超出

（2）转换状态，dp[i][j]：表示前i个，物品总价值为j时的最小重量。

（3）选取满足条件的最大的价值

#include<cstdio>

#include<cstring>

#include<stdlib.h>

#include<iostream>

#include<algorithm>

#include<stack>

#include<queue>

#include<map>

#include<set>

#include<vector>

using namespace std;

typedef long long ll;

typedef pair<int,int> pii;

const int maxn = 2e5 + 5;

const int inf = 0x3f3f3f3f;

int dp[101][10001];

int w[maxn];

int v[maxn];

int n,W;

int main(){

cin >> n >> W;

int max\_v = 0;

for(int i = 0;i < n;i++){

cin >> w[i] >> v[i];

max\_v = max(max\_v,v[i]);

}

for(int i = 0;i < 100;i++){

for(int j = 0;j < 10000;j++){

dp[i][j] = inf;

}

}

dp[0][0] = 0;

for(int i = 0;i < n;i++){

for(int j = 0;j <= n \* max\_v;j++){

if(j < v[i]) dp[i + 1][j] = dp[i][j];

else dp[i + 1][j] = min(dp[i][j],dp[i][j - v[i]] + w[i]);

}

}

int res = 0;

for(int i = 0;i <= n \* max\_v;i++)

if(dp[n][i] <= W) res= i;

cout << res << endl;

return 0;

}

讨论三 状态压缩：

输入:每组数据第一-行是n和k，接下来一-行是n个整数,k种颜色用1到k表示，以两个0表示结束

输出:拿走石头使其合法的最小块数

样例:

输入:

103

2122113133

00

输出: 2

如何表示状态：这里有两个约束条件就是我们要知道他前面出现过了什么颜色，才可以判断我们现在的这个颜色是否可以插入我们还必须知道这个状态的末尾元素，如果前面出现过这种颜色且末尾不是这个元素那么我们不能插入，如果正好末尾是这个元素那么我们就可以插入。

D[m][ i][j ]:表示前m块石头的合法方案中，对应串的尾色是i，且出现过的颜色的编码是j的这些方案的拿走石头块数最少的数量。接下来就是状态转移就是对第i个枚举前面出现过的每种颜色和每种末尾。如果前面这个颜色没出现过可以直接的插入如果前面这个颜色出现过，而且末尾元素正好和a[i]元素一样，也可以直接插入。每次都可以选择去掉它，然后Dp[i][j][k]=dp[i-1][j][k]+1

所以，答案为dp[n][i][j] i从0到(1<<n) j从(0到k)

讨论四 计数DP：

限制条件：

1≤m≤n≤1000

2≤M≤10000

输入

n = 4

m = 3

M = 10000

输出

4 （1+1+2=1+3=2+2=4）

设dp[i][j]为j的i的划分总数

递推方法 dp[i][j]= dp[i][j-i]+dp[i-1][j];

让我们来看看上述两个 部分： 首先前半部分 j-i的 i划分，这个代表的意思是，在j的i划分中，若划分出的所有元素都大于0，则它的划分等于每个元素都减1的 共j-i的i划分。 后半部分，j的i-1划分，这个是当存在元素为0，则将其总划分数减一，至于为什么减一，这是因为在递推过程中可以发现减到i=0的情况，已经加入进去了。

讨论五 流水作业调度：

1.判断动态规划：

首先m1的加工结束时间就是所有的时间和，但m2的加工时间和最小值，

求解过程是跳跃性的 所以判定为动态规划

2.这道题用到了johnson算法，我是拿个例子来理解的，

比如：假设再m1上的加工时间为a,在m2上的加工时间为b

如果作业i和作业j满足min(aj,bi) > min (ai,bj) 则称作业i和j

满足johnson法则

i在m1和m2上的加工时间为 3,4

j在m1和m2上的加工时间为 6,7

min(4,6) > min(3,7)

则作业i和j满足johnson法则

若先加工i

m2的结束时间 = 3+4+2+7 = 16

若先加工j

m2的结束时间 = 6+7+4 = 17

所以说johnson确定了加工的顺序

#include<bits/stdc++.h>

using namespace std;

struct Node{

int number1; // 在m1上的加工时间

int number2;// 在m2上的加工时间

};

//N1集合当中ai的递增排序

bool sort\_N1(Node a,Node b){

return a.number1 < b.number1;

}

//N2中按bi的降序排序

bool sort\_N2(Node a,Node b){

return a.number2 > b.number2;

}

int main(){

int N;

int a[101];

int b[101];

Node \*stu1 = new Node[101];

Node \*stu2 = new Node[101];

Node \*stu3 = new Node[101];

cin >> N;

for(int i = 0; i < N; i++){

cin >> a[i] >> b[i];

}

// for(int i = 0; i < N; i++){

// cout << b[i] << ' ';

// }

//开始处理数据在N1的集合当中是作业ai < bi(即在m2上的加工时间大于在m1上的加工时间)

//N2上的集合是作业的(ai > bi)

//还要注意的是在N1上是按照ai的递增排序，在N2上是按照bi的递减排序

int k1 = 0,k2 = 0;

for(int i = 0; i < N; i++){

//集合N1上

if(a[i] < b[i]){

stu1[k1].number1 = a[i];

stu1[k1].number2 = b[i];

k1++;

}else{

//集合N2上

stu2[k2].number1 = a[i];

stu2[k2].number2 = b[i];

k2++;

}

}

// for(int i = 0; i < k1; i++){

// cout << stu1[i].number1 << ' ' << stu1[i].number2 << endl;

// }

//对N1集合进行排序（按ai的递增排序）

sort(stu1,stu1+k1,sort\_N1);

//对N2集合进行排序（按bi的递减顺序进行排序）

sort(stu2,stu2+k2,sort\_N2);

//将N1和N2集合合并（N1在前，N2在后）

int k3 = 0;

for(int i = 0; i < k1; i++){

stu3[k3].number1 = stu1[i].number1;

stu3[k3].number2 = stu1[i].number2;

k3++;

}

for(int i = 0; i < k2; i++){

stu3[k3].number1 = stu2[i].number1;

stu3[k3].number2 = stu2[i].number2;

k3++;

}

//验证数据

// for(int i = 0; i < k3; i++){

// cout << stu3[i].number1 << ' ';

// }

//计算时间m1,m2的结束时间

int m1,m2;

m1 = stu3[0].number1;//第一个工作在m1执行完的时间

m2 = stu3[0].number2 + m1;//第一个工作的总体执行时间

for(int i = 1; i < N; i++){

m1 = m1 + stu3[i].number1;//第i个工作在m1上的执行时间

if(m1 < m2){//说明m2上的工作还没有完成

m2 = m2 + stu3[i].number2;//工作累积

}else if(m1 > m2){//说明m2需要等待，因为m1上的工作还未完成

m2 = m1 + stu3[i].number2;

}

}

cout << m1 << ' ' << m2;

}