算法实验2题解

A - My Huge Bag

题意非常清晰，“正常”的01背包问题，但是注意到数据范围：

$$
\begin{aligned}
1 \leq N \leq 100 \\
1 \leq W \leq 10^9 \\
1 \leq w_i \leq W \\
1 \leq v_i \leq 10^3
\end{aligned}
$$

背包的容量巨大，显然不能开10^9的数组来跑朴素的01背包动态规划。注意到物品总和与价值的上限，考虑最好的情况，我们最多能得到的最大的价值为

$$
V = N \cdot \max\{v_i\} = 100 \cdot 10^3 = 10^5
$$

如果将此作为动态规划的突破口，V是一个可以接受的空间大小，因此，我们考虑基于价值的动态规划，如下：

$$
dp(x)\text{表示当获取总价值为}x\text{的物品时所需要的最小背包容量。}
$$

状态转移方程如下：

$$
dp(x) = \min_{0 \leq x \leq V}\{dp(x - v[i]) + w[i]\}
$$

初始化：

$$
dp(x) = \begin{cases} 0 & x = 0 \\ \infty & 0 < x \leq V \end{cases}
$$

完成上述动态规划之后，我们可以写出解的表达式：

$$
\max\{x \mid dp(x) \leq W\}
$$

即遍历dp数组，选取满足题目要求的背包大小的最大价值即可。

技巧滚动数组

```
for(i = 1; i < N; i++)
{   // 前i件物品
        for(j = 1; j < N; j++)
        {   // 背包剩余空间
            if(w[i] > j){   // 第i件物品太重放不进去
                dp[i][j] = dp[i - 1][j];
            }
```

```
            else{
                value1 = dp[i-1][j - w[i]] + v[i];
                value2 = dp[i-1][j];
                dp[i][j] = max(value1, value2);
            }
        }
}
//v[i]为价值,w[i]重量,0-1背包，注意倒序
for(int i = 0; i < n; i++)
{
    for(int j = maxVal; j > 0; j--)
    {
        if(j >= v[i])
        {
            dp[j] = min(dp[j],dp[j-v[i]] + w[i]);
        }
    }
}
```

`

B - Beat That Monster

题意：

1. 怪物生命上限为H。
2. 你有N种咒语，第i种咒语的伤害为A_i，消耗为B_i，每种咒语可以施展多次。
3. 设每种咒语的施展次数为k_i，则你的目标是求取打死怪物所需的最小消耗，即：

$$\text{约束条件：}H-\sum_{i=0}^{N}k_iA_i\le 0$$
$$\text{目标：}\min\sum_{i=0}^{N}k_iB_i$$

目标很直接——最小消耗，而数据量也在可行的范围内，所以直接作为动态规划的目标即可，考虑如下动态规划：

$$dp[i]\text{表示给予怪物}i\text{点伤害所需的最小消耗}$$

和朴素的01背包不同，题目中的"物品"，即咒语，可以使用多次，因此dp数组的上限需要注意：

$$UpperBound=H+\max\{A_i\}$$

尽管可以释放很多次咒语，但是当怪物已经死了，继续施展咒语是没有意义的，最多在怪物快死的时候（生命值很低的时候）释放一次大招（伤害很高的咒语），因此有意义的解所造成的伤害一定严格小于上述值。

故每次更新的状态转移方程如下：

$$dp(x+A_i)=\min\{dp(x)+B_i\},\ \text{其中}\begin{cases}x\ge 0\\ x+A_i<UpperBound\end{cases}$$

该方程表示，在已经给怪物造成x点伤害的基础上，转移到给怪物造成$x+A_i$点伤害这一状态，使用第i种咒语是否消耗更小。

初始化：

$$dp(x)=\begin{cases}0 & x=0\\ \infty & 0<x<UpperBound\end{cases}$$

完成上述动态规划之后，我们可以写出解的表达式：

$$\min\{dp(x)\ \big|\ H\le x<UpperBound\}$$

即，在保证约束条件成立的情况下，选取最小的消耗。

`

```
//完全背包，正序迭代
for(int i = 0; i < n; i++)
{
    for(int j = 0; j <= h; j++)
    {
        if(j >= w[i])
        {
            dp[j] = min(dp[j],dp[j-w[i]] + v[i]);
        }
        else
        {
            dp[j] = min(dp[j],dp[0] + v[i]);
        }
    }
}
```

`

C - Merge Slimes!!!

题意：

有N堆史莱姆，大小分别为a_i，每次只能合并相邻的两堆，每次合并所消耗的代价是两堆史莱姆的大小之和，而我们的目标是最小化合代价地将它们合并成一堆。

和课内讲的矩阵链乘法是一道题哦！这里不做过多解释，直接给出状态转移方程。

$$dp(l,r)表示从第l堆合并到第r堆所需要的最小代价$$

状态转移方程：

$$dp(l,r)=\begin{cases}0 & l=r\\ \min_{l\le k<r}\{dp(l,k)+dp(k+1,r)+\sum_{i=l}^{r}a_i\} & l\ne r\end{cases}$$

最终的解即为$dp(1,N)$。

`

```
for(int k = 1;k <= n-1; k++)
{
    for(int i = 1; i+k <= n; i++)
    {
        for(int j =i; j <= i+k;j++)
        {
            dp[i][i+k] = min(dp[i][i+k],dp[i][j]+dp[j+1][i+k]+sum[i+k]-sum[i-1]);
        }
    }
}
```

、

因为矩阵链乘法中的代价计算仅仅是一次常数级别的乘法，而这里需要求和，因此复杂度会更高，如何降下来呢？这里给出提供一种前缀和优化。

我们可以在读取数据之后，使用线性级别的时间计算数列$\{a_i\}$的和，并保存：

$$S_n = \sum_{i=1}^{n} a_i$$

而上述状态转移方程中的求和部分可以写成

$$\sum_{i=l}^{r} a_i = \sum_{i=1}^{r} a_i - \sum_{i=1}^{l-1} a_i = S_r - S_{l-1}$$

这样，在这个问题中，我们也可以使用常数级别的时间计算代价了！

D - Happy TSP

之所以这道题会出现在这里，是因为我们上课讨论过动态规划算法的复杂度以及 NP 问题。这里希望同学们能了解并实操一下使用动态规划算法在小数据范围内暴力精确求解一个 NP Hard 问题。

旅行商问题的详细内容不在此赘述，直接进入分析。

首先，记题中的距离函数如下：

$$dist(p,q) = |p.x - q.x| + |p.y - q.y| + \max(0,\ q.z - p.z);$$

接着，我们有一个很直接的想法，将当前城市和已经游历过的城市集合作为动态规划的量，因此有：

$$dp(i,S)\text{表示你正游历到第}i\text{个城市，而你已经游历过的城市集合为}S\text{时的最小代价}$$

那么这个量怎么确定呢？因为不能重复游历同一个城市，因此自然是从另一个城市j前往城市i，而在此之前游历过的城市集合S必然不包含i，据此写出状态转移方程：

$$dp(i,S) = \min\{dp(j, S/\{i\}) + dist(j,i) \mid j \in S/\{i\}\}$$

最后，问题的解即为，在已经遍历所有城市，而此时落脚在第k个城市的情况下，回到起点（第1个城市）的总代价的最小值：

$$\min_{1\le k\le n}\{dp(k,N) + dist(k,1)\}\text{，其中}N\text{表示全部城市构成的集合}$$

问题至此看似是搞定了，但是可行嘛？——集合怎么表示？

因为是小数据范围内，城市最多只有17个，因此这里给出一种富有技巧性的集合表示方法——二进制表示。

例如，对于集合$\{2,3,5\}$，如果采用二进制表示法，则有$S=2^2+2^3+2^5=44$。

再如，枚举一个大小为n的集合的全部子集，参考代码如下：

```
int lim = (1 << n) - 1;           // n 位全 1 的数，表示原集合
for (int i = 1; i <= lim; ++i) {  // 遍历全部非空子集i
    cout << "Subset " << i << ": ";
    for (int j = 0; j < n; ++j) {  // 有哪几个元素在当前枚举的子集i中
        if (i & (1 << j))          // 判断第j元素是否在当前枚举的子集i中
            cout << j << " ";
        else
            cout << "x ";
    }
    cout << endl;
}
```

当$n=4$时，上述代码的输出如下：

```
Subset 1: 0 x x x
Subset 2: x 1 x x
Subset 3: 0 1 x x
Subset 4: x x 2 x
Subset 5: 0 x 2 x
Subset 6: x 1 2 x
Subset 7: 0 1 2 x
Subset 8: x x x 3
Subset 9: 0 x x 3
Subset 10: x 1 x 3
Subset 11: 0 1 x 3
Subset 12: x x 2 3
Subset 13: 0 x 2 3
Subset 14: x 1 2 3
Subset 15: 0 1 2 3
```

至此，我们可算是解决了问题，像这种使用二进制表示集合或状态的动态规划，一般称为状态压缩动态规划问题。

我的思路：

稍微有点变化，状态定义和上面的类似，但我把正在游历的城市i算在已经游历过的城市里面，这样状态转移方程变为：

$$dp(i,S)=\min\{dp(j,S\cup\{j\})+dist(i,j)\,|\,j\notin S\}$$

同时因为从0节点出发，最后返回0节点，故可定义边界状态为$dp(0,V)=0$，则最后的结果为$dp(0,\emptyset)$。因为决策的顺序不像前面的题，很难用循环表示，所以我使用了记忆化搜索的方法，代码如下：

```
#include <bits/stdc++.h>
#define inc(i, j, k) for (int i = j; i < k; i++) // 注意我宏定义枚举左闭右开区间
using namespace std;

int f[17][1 << 17], n, g[17][17], x[17], y[17], z[17];
int dp(int i, int s) {
    if (f[i][s] != -1) return f[i][s]; // 该状态算过了就不用算
    f[i][s] = 0x3fffffff;
    inc(j, 0, n)
```

```
        if ((s & (1 << j)) == 0) f[i][s] = min(f[i][s], dp(j, s | (1 << j))
+ g[i][j]);
    return f[i][s];
}
int main() {
    scanf("%d", &n); inc(i, 0, n) scanf("%d%d%d", &x[i], &y[i], &z[i]);
    // 计算任意两点之间的距离
    inc(i, 0, n) inc(j, 0, n) g[i][j] = abs(x[j] - x[i]) + abs(y[j] - y[i])
+ max(0, z[j] - z[i]);
    inc(i, 1, n) inc(j, 0, 1 << n) f[i][j] = -1; // 状态初始化为没算过
    f[0][0] = -1; f[0][(1 << n) - 1] = 0; inc(i, 1, (1 << n) - 1) f[0][i] =
0x3fffffff; // 如果终点不是0则为非法状态，所以将其定义为一个很大的值，让状态转移的时候不
要转移到这些非法状态
    printf("%d", dp(0, 0)); return 0;
}
```

E - Crush Proper Gems

题意：

1. 我们有N个宝石，标号从1开始。
2. 我们可以进行一种操作：选择一个正整数x，将标号为x的倍数的宝石全部砸碎。
3. 完成一系列操作之后，对于没有砸碎的的宝石i，我们可以得到相应的奖励a_i，但是奖励也有可能是负的（滑稽），所以我们的目标是最大化奖励和。

这个题就很裸了，和课上讲的最大流建模类似，以下是具体的建模过程：

首先，确定目标：这是一个 S-T cut 问题。我们需要将序列$1, 2, 3, \ldots, N$划分成两组，S组为需要被砸碎的宝石序号组，T组为剩下的宝石序号组。

当然，理想情况下，我们希望所有负值奖励的宝石全部被砸碎，正值奖励的宝石全部保留。设X为理想中所能获得的最大奖励，即

$$X = \sum_{i=1,\ a_i>0}^{N} a_i$$

但是，理想归理想，不是总能满足的。对于序号$i, j \in \{1, 2, \ldots, N\}$，$j$为$i$的倍数，若$i$碎了，$j$没有碎，这就存在着矛盾。因此，为了避免这种矛盾，我们必须建立如下惩罚机制：

1. 对于序号$i, j \in \{1, 2, \ldots, N\}$，$j$为$i$的倍数，$i \in S$，$j \in T$，惩罚系数设置为无穷大。
2. 对于$a_i \leq 0$，$i \in T$，惩罚系数为$-a_i$。
3. 对于$a_i > 0$，$i \in S$，惩罚系数为a_i。

据此，我们建立一个$N + 2$个点的图G（加入源点s和终点t），图中的边即根据上述惩罚机制加入容量属性，如下：

1. 如果$a_i \leq 0$，加入边$< s, i >$，容量为$-a_i$。
2. 如果$a_i > 0$，加入边$< i, t >$，容量为a_i。
3. 对于所有的$j = 2i, 3i, \ldots$，加入边$< i, j >$，容量为∞。

这样，我们就能在图G上跑最大流算法啦！！！设最大流算法在图G上得出的结果为Y，它表示最小的惩罚系数和（想想为什么？），因此，题目的解即为$X - Y$。

我的思路：

直接拿老师上课讲的模型进行转化。

首先，老师上课讲的是对一个带点权的有向无环图，选取一个子图，该子图所有节点的前驱都在该子图中，希望子图所有点权和最大。如果我设该问题所有被砸碎的宝石为要选择的子图，则每砸一个宝石，编号为其倍数的宝石都要砸掉，故应该对于每个宝石，将编号为其倍数的宝石连一条有向边指向它。

其次是最值问题，因为这题要求没砸的宝石价值最大，那么就是砸掉的宝石价值最小，也就是选取的子图总点权最小。为了转化为老师讲的点权最大的模型，可以对所有的宝石价值求个相反数，然后跑一遍老师的算法，求最大点权，在求一次相反数，就是最小点权了。因此建立网络流模型的时候是源点向所有价值为正的宝石连边，所有价值为负的宝石连边。

最后的结果

=宝石总价值-被砸掉的宝石价值

=宝石总价值-子图点权和

=宝石总价值-(-新图跑网络流算法之后T割的点权和)

=宝石总价值+新图跑网络流算法之后T割的点权和

=宝石总价值+新图中所有正点和-新图最小割

=原图中所有正点的价值和-原图中所有负点的绝对值和+原图中所有负点绝对值和-新图最小割

=原图中所有正点-新图最大流

```
    #define inc(i, j, k) for (int i = j; i < k; i++) // 注意我宏定义枚举左闭右开区间
    int n; scanf("%d", &n); int s = 0, t = n + 1; long long sum = 0;
    inc(i, 0, n) {
        scanf("%d", &a[i]);
        if (a[i] > 0) pe(s, i + 1, a[i]), sum += a[i];
        if (a[i] < 0) pe(i + 1, t, -a[i]);
    }
    inc(i, 0, n) inc(j, i + 1, n) if ((j + 1) % (i + 1) == 0) pe(j + 1, i + 1, 0x3fffffff);
    printf("%lld", sum - dinic(s, t)); return 0;
```