

5.1 课堂笔记 8

本周作业：

1. P2725 自己再做一下
2. 练习 1

有n个正整数，找出其中和为t(t也是正整数)的可能的组合方式。如：

n=5,5个数分别为1,2,3,4,5，t=5；

那么可能的组合有5=1+4和5=2+3和5=5三种组合方式。

输入

输入的第一行是两个正整数n和t，用空格隔开，其中 $1 \leq n \leq 20$,表示正整数的个数，t为要求的和($1 \leq t \leq 1000$)

接下来一行是n个正整数，用空格隔开。

输出

和为t的不同的组合方式的数目。

样例输入

5 5

1 2 3 4 5

样例输出

3

3. 练习 2：求最短编辑距离

字符串编辑距离：

- Levenshtein距离是一种计算两个字符串间的差异程度的字符串度量 (string metric) 。
- Levenshtein距离就是从一个字符串修改到另一个字符串时，其中编辑单个字符（比如修改、插入、删除）所需要的最少次数。
- 俄罗斯科学家Vladimir Levenshtein于1965年提出了这一概念。
- 例如：由串“NOTV”修改为“LOVER”需要4次字符编辑操作
- 解释：NOTV -> LOTV -> LOV -> LOVE -> LOVER

上机，求两个字符串的编辑距离

作业讲解：

作业 1：

在一个地图上有 n 个地窖 ($n \leq 200$) ,每个地窖中埋有一定数量的土豆。同时,给出地窖之间的连接路径,并规定路径都是单向的,且保证都是小序号地窖指向大序号地窖,也不存在可以从一个地窖出发经过若干地窖后又回到原来地窖的路径。某人可以从任一处开始挖土豆,然后沿着指出的连接往下挖 (仅能选择一条路径),当无连接时挖土豆工作结束。设计一个挖土豆的方案,使他能挖到最多的土豆。

如下图所示:圆圈内的1 2 3 4 5 6,代表6个地窖的编号,地窖编号旁边的数字代表这个地窖土豆的数量!

输入

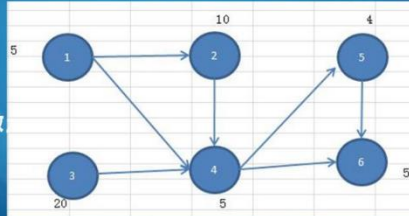
第一行:地窖的个数;
第二行为依次每个地窖土豆的个数
下面若干行:

$x_i y_i$

表示从 x_i 可到 y_i , $x_i < y_i$ 。

输出

第一行输出挖土豆的地窖编号的顺序: $k_1-k_2-\dots-k_v$
第二行输出一个整数,代表最多能挖到的土豆的数量



样例输入

```

6
5 10 20 5 4 5
1 2
1 4
2 4
3 4
4 5
4 6
5 6

```

样例输出

```

3-4-5-6
34

```

代码:

```

#define maxn 100000
using namespace std;

int a[maxn + 5];
int dp[maxn + 5];
int from[maxn + 5]; ///from[i] 代表i是从from[i] 转移过来的
vector<int> G[maxn + 5];
int main(){
    int n;
    cin >> n;
    for(int i = 1; i <= n; i++){
        cin >> a[i];
    }
    int u, v;
    while(cin >> u >> v){
        G[v].push_back(u);
    }

    int ans = 0;
    int ansid;

```

```

for(int i = 1; i <= n; i++){
    dp[i] = a[i];
    from[i] = i;
    for(int j = 0; j < G[i].size(); j++){
        if(dp[G[i][j]] + a[i] > dp[i]){
            dp[i] = dp[G[i][j]] + a[i];
            from[i] = G[i][j];
        }
    }
    if(dp[i] > ans){
        ans = dp[i];
        ansid = i;
    }
}

stack<int> Q;
int now = ansid;
while(from[now] != now){
    Q.push(now);
    now = from[now];
}
Q.push(now);

cout << Q.top();
Q.pop();
while(!Q.empty()){
    cout << "-" << Q.top();
    Q.pop();
}

cout << ans << endl;
return 0;

```

作业 2:
P2008

```

#include <bits/stdc++.h>
#define maxn 10000
using namespace std;
struct Point{
    int len;
    int sum;
    Point(){}
    Point(int len, int sum){
        this->len = len;
        this->sum = sum;
    }
    friend bool operator<(const Point &a, const Point &b){
        return a.len < b.len;
    }
}dp[maxn + 5];
int num[maxn + 5];

int main (){
    int n;
    cin >> n;
    for(int i = 1; i <= n; i++){
        cin >> num[i];
    }
    dp[1] = Point(1, num[1]);
    for(int i = 2; i <= n; i++){
        dp[i] = Point(1, num[i]);
        for(int j = 0; j < i; j++){
            if(num[i] >= num[j]){
                dp[i] = max(dp[i], Point(dp[j].len + 1, dp[j].sum + num[i]));
            }
        }
    }
    for(int i = 1; i <= n; i++){
        cout << dp[i].sum;
        if(i == n)
            cout << endl;
        else
            cout << " ";
    }
    return 0;
}

```

作业 3:

P1233

代码 1 (n*n 版本):

```
struct Point{
    int L;
    int W;
}p[maxn + 5];

bool cmp(Point a, Point b){
    if(a.L == b.L)
        return a.W > b.W;
    return a.L > b.L;
}
int dp[maxn_ + 5];
```

```

int main (){
    int n;
    cin >> n;
    for(int i = 0; i < n; i++){
        cin >> p[i].L >> p[i].W;
    }
    sort(p, p + n, cmp);
    int len = 1;
    dp[0] = p[0].W;
    for(int i = 1; i < n; i++){
        if(p[i].W > dp[len - 1]){
            dp[len++] = p[i].W;
        }else{
            for(int j = 0; j < len; j++){
                if(p[i].W <= dp[j]){
                    dp[j] = p[i].W;
                    break;
                }
            }
        }
    }
    cout << len << endl;
    return 0;
}

```

代码 2: ($n \cdot \log n$ 版本,用了指针,具体思路就是二分,同学们可以自己写二分,不是必须用 upper_bound)


```

struct Point{
    int L;
    int W;
}p[maxn + 5];

bool cmp(Point a, Point b){
    if(a.L == b.L)
        return a.W > b.W;
    return a.L > b.L;
}

int dp[maxn + 5];
int main (){
    int n;
    cin >> n;

    for(int i = 0; i < n; i++){
        cin >> p[i].L >> p[i].W;
    }
    sort(p, p + n, cmp);

    int len = 1;
    dp[0] = p[0].W;
    for(int i = 1; i < n; i++){
        if(p[i].W > dp[len - 1]){
            dp[len++] = p[i].W;
        }else{
            ///大于等于的第一个数
            int *it = lower_bound(dp, dp + len, p[i].W); ///>=
            *it = p[i].W;
        }
    }
    cout << len << endl;
    return 0;
}

```

新知识:

1. 01 背包

问题引入:

有 N 件物品和一个容量为 V 的背包。第 i 件物品的费用是 $w[i]$, 价值是 $v[i]$, 求将哪些物品装入背包可使价值总和最大。

简单想法:

设 $F(n, c)$ 为把前 n 个物品放进容量为 C 的背包里能获取的最大价值

1. 如果我们不放第 n 个物品, 那么 $F(n, c) = F(n - 1, c)$
2. 如果我们放第 n 个物品, 那么 $F(n, c) = F(n - 1, c - w[n]) + v[n]$
因此: 我们得到了递推式 (状态转移方程为)
 $F(i, c) = \max(F(i - 1, c), F(i - 1, c - w[i]) + v[i])$
实现方法: 记忆化搜索、动态规划

初始代码:

```
for (int i = 1; i <= n; i++)  
    for (int j = w[i]; j <= C; j++)  
        F[i][j] = max(F[i - 1][j], F[i - 1][j - w[i]] + v[i])
```

空间优化:

上面的动态规划算法使用了 $O(n \times C)$ 的空间复杂度 (因为我们使用了二维数组来记录子问题的解), 其实我们完全可以只使用一维数组来存放结果, 但同时我们需要注意的是, 为了防止计算结果被覆盖, 我们必须从后向前分别进行计算。

优化后的最终代码:

```
for (int i = 1; i <= n; i++)  
    for (int j = C; j >= w[i]; j--)  
        f[j] = max(f[j], f[j - w[i]] + v[i]);
```

初始化的亿点细节:

我们看到的求最优解的背包问题题目中, 事实上有两种不太相同的问法。有的题目要求 "恰好装满背包" 时的最优解, 有的题目则并没有要求必须把背包装满。这两种问法的区别在初始化的时候不同。

如果是第一种问法, 要求恰好装满背包, 那么在初始化时除了 $f[0]$ 为 0 其它 $f[1...C]$ 均设为 $-\infty$, 这样就可以保证最终得到的 $f[C]$ 是一种恰好装满背包的最优解。

如果并没有要求必须把背包装满, 而是只希望价格尽量大, 初始化时应该将 $f[0...C]$ 全部设为 0。

这个小技巧完全可以推广到其它类型的背包问题, 后面也就不再对进行状态转移之前的初始化进行讲解。

2. 完全背包

问题引入:

有 N 种物品和一个容量为 V 的背包, 每种物品都有无限件可用。放入第 i 种物品的费用是 C_i , 价值是 W_i 。求解: 将哪些物品装入背包, 可使这些物品的耗费的费用总和不超过背包容量, 且价值总和最大。

首先直接想到的递推公式:

$$F(i, j) = \max(F(i - 1, j - k * w[i]) + k * v[i], F(i, j))$$

但是这样一来时间复杂度就多了个 $*k$, 可不可以优化呢?

优化策略:

一、贪心策略

二、转换成 0/1 背包问题

- 直接转换
- 二进制优化

把第 i 种物品拆成费用为 $v[i]*2^k$ 、价值为 $w[i]*2^k$ 的若干件物品，其中 k 取遍满足 $C_i*2^k \leq V$ 的非负整数。这是二进制的思想。因为，不管最优策略选几件第 i 种物品，其件数写成二进制后，总可以表示成若干个 2^k 件物品的和。这样一来就把每种物品拆成 $O(\log [V / C_i])$ 件物品，是一个很大的改进。

- 利用本层数据(重点)和代码

代码：

```
for (int i = 1; i <= n; i++)  
    for (int j = w[i]; j <= C; j++)  
        f[j] = max(f[j], f[j - w[i]] + v[i]);
```

你会发现，这个代码与 01 背包问题的伪代码只有 j (枚举的背包空间) 的循环次序不同而已。

为什么这个算法就可行呢？首先想想为什么 01 背包中要按照 j 递减的次序来循环。让 C 递减是为了保证第 i 次循环中的状态 $F[i, j]$ 是由状态 $F[i - 1, j - C_i]$ 递推而来。

换句话说，这正是为了保证每件物品只选一次，保证在考虑“选入第 i 件物品”这件策略时，依据的是一个绝无已经选入第 i 件物品的子结果 $F[i - 1, j - C_i]$ 。而现在完全背包的特点恰是每种物品可选无限件，所以在考虑“加选一件第 i 种物品”这种策略时，却正需要一个可能已选入第 i 种物品的子结果 $F[i, j - C_i]$ ，所以就可以并且必须采用 j 递增的顺序循环。这就是这个简单的程序为何成立的道理。

3. 多重背包

问题引入：

有 N 种物品和一个容量为 V 的背包。第 i 种物品最多有 M_i 件可用，每件耗费的空间是 C_i ，价值是 W_i 。求解将哪些物品装入背包可使这些物品的耗费的空间总和不超过背包容量，且价值总和最大。

初步想法：

这题目和完全背包问题很类似。基本的方程只需将完全背包问题的方程略微一改即可。因为对于第 i 种物品有 $M_i + 1$ 种策略：取 0 件，取 1 件……取 M_i 件(M_i 取值是满足 $C_i * M_i \leq j$ 的最大值)。令 $F[i, j]$ 表示前 i 种物品恰放入一个容量为 j 的背包的最大价值，则有状态转移方程：

$$F[i, j] = \max\{F[i - 1, j - k * C_i] + k * W_i \mid 0 \leq k \leq M_i\}$$

复杂度是 $O(V \sum M_i)$ 。

二进制优化

仍然考虑二进制的思想，我们考虑把第 i 种物品换成若干件物品，使得原问题中第 i 种物品可取的每种策略——取 $0 \dots M_i$ 件——均能等价于取若干件代换以后的物品。

另外，取超过 M_i 件的策略必不能出现。

方法是：将第 i 种物品分成若干件 01 背包中的物品，其中每件物品有一个系数。这件物品的费用和价值均是原来的费用和价值乘以这个系数。令这些系数分别为

$1, 2, 2^2, \dots, 2^{k-1}, M_i - 2^k + 1$ ，且 k 是满足 $M_i - 2^k + 1 > 0$ 的最大整数。

例如，如果 M_i 为 13，则相应的 $k=3$ ，这种最多取 13 件的物品应被分成系数分别为 1, 2, 4, 6 的四件物品。

分成的这几件物品的系数和为 M_i ，表明不可能取多于 M_i 件的第 i 种物品。另外这种方法也能保证对于 $0 \dots M_i$ 间的每一个整数，均可以用若干个系数的和表示。（课堂上依旧进行过简单证明，证明方法不要求掌握，记住这个结论即可，以后还能用于很多场景）

这样就将第 i 种物品分成了 $O(\log M_i)$ 种物品，将原问题转化为了复杂度为 $O(V \sum \log M_i)$ 的 01 背包问题，是很大的改进。

课堂练习代码：

1. P1049

```
int dp[maxn + 5];
int v[maxn + 5];
int main () {
    int V, n;
    cin >> V >> n;
    for (int i = 1; i <= n; i++) {
        cin >> v[i];
    }

    for (int i = 1; i <= n; i++) {
        for (int j = V; j >= v[i]; j--) {
            dp[j] = max(dp[j - v[i]] + v[i], dp[j]);
        }
    }
    cout << V - dp[V] << endl;
    return 0;
}
```

2. P1060

```

#include <bits/stdc++.h>
#define maxn 30000
#define LL long long int
using namespace std;

int dp[maxn + 5];
int v[maxn + 5];
int p[maxn + 5];

int main(){
    int n, m;
    cin >> n >> m;
    for(int i = 0; i < m; i++){
        cin >> v[i] >> p[i];
    }

    for(int i = 0; i < m; i++){
        for(int V = n; V >= v[i]; V--){
            dp[V] = max(dp[V], dp[V - v[i]] + v[i] * p[i]);
        }
    }
    cout << dp[n] << endl;
}

```

3. P1048

```

int dp[maxn + 5];
int t[maxn + 5];
int v[maxn + 5];

int main(){
    int T, m;
    cin >> T >> m;
    for(int i = 0; i < m; i++){
        cin >> t[i];
        cin >> v[i];
    }
    for(int i = 0; i < m; i++){
        for(int V = T; V >= t[i]; V--){
            dp[V] = max(dp[V], dp[V - t[i]] + v[i]);
        }
    }

    cout << dp[T] << endl;
}

```

4. P1616

```

LL dp[maxn + 5];
LL t[maxm + 5];
LL v[maxm + 5];

int main(){
    int T, m;
    cin >> T >> m;
    for(int i = 0; i < m; i++){
        cin >> t[i];
        cin >> v[i];
    }
    for(int i = 0; i < m; i++){
        for(int V = t[i]; V <= T; V++){
            dp[V] = max(dp[V], dp[V - t[i]] + v[i]);
        }
    }

    cout << dp[T] << endl;
}

```

5. 最长公共子串

最长公共子串(Longest Common Substring)

如果有两个字符串如下：

s1 = "helloworld"

s2 = "loop"

最长公共子串为：s = "lo"

1. 求两个串最长公共子串长度
2. 输出这个子串

```

string s1,s2;
int dp[maxn + 5][maxn + 5];
int main(){
    cin >> s1 >> s2;
    int n, m;
    n = s1.length();
    m = s2.length();
    int maxlen = INT_MIN;
    int maxid;
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= m; j++){
            if(s1[i - 1] == s2[j - 1]){
                dp[i][j] = dp[i - 1][j - 1] + 1;
                if(dp[i][j] > maxlen){
                    maxlen = dp[i][j];
                    maxid = i;
                }
            }
        }
    }
    cout << maxlen << endl;
    cout << s1.substr(maxid - maxlen, maxlen) << endl;
    return 0;
}

```