背包九讲

目录

第一讲 01背包

01背包是每种武平只能选择一次，计算出最大价值的问题，先上01背包的状态转移方程：

$$f[i][j] = \max\{f[i-1][j], f[i-1][j-w[i]] + v[i]\}$$

下面来解释一下这个状态转移方程：

这个方程非常重要，基本上所有跟背包相关的问题的方程都是由它衍生出来的。所以有必要将它详细解释一下："将前i件物品放入容量为v的背包中"这个子问题，若只考虑第i件物品的策略（放或不放），那么就可以转化为一个只牵扯前i-1件物品的问题。如果不放第i件物品，那么问题就转化为"前i-1件物品放入容量为v的背包中"，价值为f[i-1][v]；如果放第i件物品，那么问题就转化为"前i-1件物品放入剩下的容量为v-c[i]的背包中"，此时能获得的最大价值就是f[i-1][v-c[i]]再加上通过放入第i件物品获得的价值w[i]。即代码为：

Copy

```
//f[i][j]的意义是表示只看前i个物品，总体积是j的情况下，总价值最大是多少
for(int i = 1; i <= n; ++i){
    for(int j = 0; j <= C; ++j){
            if(j >= v[i])   dp[i][j]=max(dp[i-1][j],dp[i-1][j-v[i]] + w[i]); //取
            else    dp[i][j] = dp[i-1][j];          //不取第i个物品
    }
```

```
}
cout<<dp[n][C]<<endl;
```

这个方程还可以对空间进行优化，下面是一位数组实现01背包：

Copy

```
//f[i]的意义为当前体积为i的情况下背包的最大价值
for(int i = 1;i <= n;++i)
        for(int j = C;j >= w[i]; --j)
            dp[j] = max(dp[j], dp[j-w[i]] + v[i]);
cout << dp[C] << endl;
```

这里是背包容积为c的最大价值，将f数组全初始化为0，**如果想要求解背包体积恰好为c的情况下其最大价值是多少，只需将f[0]初始化为0，而将其他的初始化为-inf即可。**

例题：

Copy

```
#include <bits/stdc++.h>

using namespace std;
const int maxn = 1100;

int n, c;
int f[maxn], w[maxn], v[maxn];

int main()
{
    scanf("%d %d", &n, &c);
    for(int i = 1; i <= n; ++i){
        scanf("%d %d", &w[i], &v[i]);
    }
    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]);
        }
    }
    printf("%d\n", f[c]);
}
```

第二讲 完全背包

完全背包是指背包里面的物品可以选择无限次或每个物品有无限个，求最大价值。下面是完全背包的状态转移方程：

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

上面的状态转移方程可以理解为试探性取，即对于个数多与1的物品，我们可以试探性的取一次，取两次...不过这种算法的时间复杂度会比较高，其时间复杂度为O(n*V*sum(k))。下面是代码模板

```
for(int i = 1;i <= n; ++i)
        for(int j = 1; j <= c; ++j)
                for(int k = 0;k <= c/v[i]; ++k)
                        if(j > k *v[i]) dp[i][j] = max(dp[i-1][j], dp[i-1][j - w[i]]+v[i]);
                        else dp[i][j] = dp[i-1][j];
```

基于上面的代码，我们还可以做一些简单的优化，1.体积大于c的不要；2.同体积的取价值最大的(其余舍弃)。即使这么优化后，一般情况下还是会超时，下面是将完全背包转化为01背包来做，即状态转移方程为：

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

上面状态转移方程的意思为在第i件物品上，如果不取该物品，则dp[i][j]=dp[i-1][j]，如果去第i件物品，则最少需要dp[i][j-v[i]]+w[i]。下面是代码模板：

```
//f[i]表示当前体积为i的情况下，背包的最大价值是多少
for(int i = 1; i <= n; ++i)
```

```
        for(int j = v[i];j <= c; ++j)  //这里与01背包不同的地方是01背包是逆序枚举，完全背包是正序枚举
                dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
cout<<dp[c]<<endl;
```

完全背包例题模板，题目同01背包，条件加了每件物品可重复选取：

Copy

```
#include <bits/stdc++.h>

using namespace std;
const int maxn = 1100;

int n, c;
int f[maxn], w[maxn], v[maxn];

int main()
{
    scanf("%d %d", &n, &c);
    for(int i = 1; i <= n; ++i){
        scanf("%d %d", &w[i], &v[i]);
    }
    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]);
        }
    }
    printf("%d\n", f[c]);
}
```

第三讲 多重背包

多重背包类似于完全背包和01背包的结合版，即每个物品有有限个或则能取有限次，求最大价值的问题。

这里最简单的解法是直接将多重背包问题看做01背包问题，在对空间和物品价值遍历的中间在对个数遍历一遍就行了，其模板为：

```
//f[i]表示总体积为i的情况下，其背包的最大价值是多少。
for(int i = 1; i <= n; ++i){
    for(int j = c; j >= w[i]; --j){
        for(int k = 1; k * w[i] <= j && k <= num[i]; ++k){
            f[j] = max(f[j], f[j - k * w[i]] + k * v[i]);
        }
    }
}
printf("%d\n", f[c]);
//类似于在01背包的情况下，再加一个for循环求个数，也是逆向枚举。
```

但是一般情况下上面解法会超时，上面的解法还可以进行优化，其可以通过二进制优化和单调队列优化。

二进制优化:将多个物品分成一个个二进制，从而将问题转化为01背包问题，复杂度是 $m * n * \log_2(num)$；**注意：含二进制优化的话数组一定要开大一点，因为是将一个数组合拆成了多个组合。**

```
int main()
{
    scanf("%d %d", &n, &c);                                    //n个物品，c的体积
    for(int i = 1; i <= n; ++i){
       scanf("%d %d %d", &a, &b, &num);      // w, v, num
       for(int j = 1; j <= num; j <<= 1){
           w[++cnt] = a * j;
           v[cnt] = b * j;
           num -= j;
       }
       if(num){
           w[++cnt] = a * num;
           v[cnt] = b * num;
       }
    }
```

```
        for(int i = 1; i <= cnt; ++i){
            for(int j = c; j >= w[i]; --j){
                f[j] = max(f[j], f[j - w[i]] + v[i]);
            }
        }
        printf("%d\n", f[c]);
    }
```

单调队列优化：思路如下：

!

show code:

Copy

第四讲 混合背包

混合背包即多个背包都可以放（01背包，多重背包，完全背包）。求解方法是将多重背包转化为01背包，然后对01背包和完全背包分治求解最大值即可。模板如下：

Copy

```
#include <bits/stdc++.h>

using namespace std;
const int maxn = 11000;

struct node{
    int w, v, k;        //体积，价值，类别(0代表01背包，1代表完全背包)
}arr[maxn];
int n, c, f[maxn], tot;

int main()
```

```
{
    scanf("%d %d", &n, &c);
    for(int i = 1; i <= n; ++i){
        int a, b, c;        //体积，价值，数量
        scanf("%d %d %d", &a, &b, &c);
        if(c == -1){        //只能使用一次
            arr[++tot].w = a;
            arr[tot].v = b;
            arr[tot].k = 0;
        }
        else if(c == 0){    //能使用无数次
            arr[++tot].w = a;
            arr[tot].v = b;
            arr[tot].k = 1;
        }
        else{               //只能使用c此
            for(int j = 1; j <= c; j <<= 1){
                arr[++tot].w = a * j;
                arr[tot].v = b * j;
                arr[tot].k = 0;
                c -= j;
            }
            if(c){
                arr[++tot].w = a * c;
                arr[tot].v = b * c;
                arr[tot].k = 0;
            }
        }
    }
    for(int i = 1; i <= tot; ++i){
        if(arr[i].k == 0){                  //01背包逆序
            for(int j = c; j >= arr[i].w; --j){
                f[j] = max(f[j], f[j - arr[i].w] + arr[i].v);
            }
```

```
        }
        else{                               //完全背包正序
            for(int j = arr[i].w; j <= c; ++j){
                f[j] = max(f[j], f[j - arr[i].w] + arr[i].v);
            }
        }
    }
    printf("%d\n", f[c]);
}
```

第五讲 二维费用的背包问题

顾名思义，二维费用即有两个约束条件，一个为重量，一个为背包容量，保证物品总容积既不超过背包容积，物品总重量也不超过背包承受重量。

以二维费用的01背包为例（完全背包即从前向后枚举，多重背包类似），直接枚举个数、体积、重量即可，下面是模板AC代码：

Copy

```
int main()
{
    scanf("%d %d %d", &n, &c, &m);      //个数，容积，重量
    for(int i = 1; i <= n; ++i){
        scanf("%d %d %d", &w[i], &s[i], &v[i]); //体积，重量，价值
    }
    for(int i = 1; i <= n; ++i){
        for(int j = c; j >= w[i]; --j){
            for(int k = m; k >= s[i]; --k){
                f[j][k] = max(f[j][k], f[j - w[i]][k - s[i]] + v[i]);
            }
        }
    }
    printf("%d\n", f[c][m]);
}
```

第六讲 分组背包

分组背包既给定一个一定容积的背包，在给定若干组物品，每组有若干个物品，但是每组的物品只能选一个，求最大价值。

模板AC代码：

Copy

```
int main()
{
    scanf("%d %d", &n, &c);      //组数，容积
    for(int i = 1; i <= n; ++i){
        int t; scanf("%d", &t);      //一组中物品的个数
        for(int j = 1; j <= t; ++j){
            scanf("%d %d", &w[j], &v[j]);
        }
        //这里要先遍历容积再遍历物品，这样就可以保证每组物品只取一个
        for(int j = c; j >= 0; --j){
            for(int k = 1; k <= t; ++k){
                if(j < w[k])    continue;
                f[j] = max(f[j], f[j - w[k]] + v[k]);
            }
        }
    }
    printf("%d\n", f[c]);
}
```

第七讲 有依赖的背包问题

类似于拓扑排序，所选物品有依赖关系，选这个物品必须选这个物品的父节点，求一定体积的背包的最大收益。

Copy

```
#include <bits/stdc++.h>

using namespace std;
const int maxn = 110;
```

```
struct edge{
    int nex, to;
}Edge[maxn<<1];
int head[maxn], tot;
int n, c, f[maxn][maxn];    //f[i][j]表示以i为根的子树，空间为j的最大价值
int p, rt, w[maxn], v[maxn];

inline void add(int from, int to){
    Edge[++tot].to = to;
    Edge[tot].nex = head[from];
    head[from] = tot;
}
void dfs(int x)
{
    //选上x节点，则以x为节点的子树(体积为w[x]~c)的最小价值为v[x]。
    for(int i = w[x]; i <= c; ++i)  f[x][i] = v[x];
    for(int i = head[x]; i != -1; i = Edge[i].nex){
        int u = Edge[i].to;
        dfs(u);
        //j的范围为w[x] ~ c，不然x这个根选不进去
        for(int j = c; j >= w[x]; --j){
            //k的范围为0 ~ j - w[x]，表示在j空间的基础上选了x剩余的空间
            for(int k = 0; k <= j - w[x]; ++k){
                f[x][j] = max(f[x][j], f[x][j-k] + f[u][k]);
            }
        }
    }
}

int main()
{
    memset(head, -1, sizeof(head));
    scanf("%d %d", &n, &c);
```

```
    for(int i = 1; i <= n; ++i){
        scanf("%d %d %d", &w[i], &v[i], &p);
        if(p == -1) rt = i;
        else add(p, i);
    }
    dfs(rt);
    printf("%d\n", f[rt][c]);
}
```

第八讲 背包问题求方案数

以01背包为例，我们只需要再设置一个cnt数组记录每个空间最大价值的方案数就可以了。01背包方案数举例如下：

```
#include <bits/stdc++.h>

using namespace std;
const int mod = 1e9 + 7;
const int maxn = 1100;

int f[maxn], n, c, cnt[maxn];
int w[maxn], v[maxn];

int main()
{
    scanf("%d %d", &n, &c);
    for(int i = 1; i <= n; ++i){
        scanf("%d %d", &w[i], &v[i]);
    }
    fill(cnt, cnt + maxn + 1, 1);       //不能用memset置为1，因为这样会使数组值全为0x01010101
    for(int i = 1; i <= n; ++i){
        for(int j = c; j >= w[i]; --j){
            int val = f[j - w[i]] + v[i];
```

```
            if(val > f[j]){
                f[j] = val;
                cnt[j] = cnt[j - w[i]];
            }
            else if(val == f[j]){
                cnt[j] = (cnt[j] + cnt[j - w[i]]) % mod;
            }
        }
    }
    printf("%d\n", cnt[c]);
}
```

第九讲 求具体方案

这里以01背包为例，在一定背包容量的情况下，输出在最大价值的情况下，选了哪些物品，使得字典序最小。

Copy

```
#include <bits/stdc++.h>

using namespace std;
const int maxn = 1100;

int f[maxn][maxn], n, c;        //f[i][j]表示在体积为j的情况下，取i~n中物品的最大价值
int w[maxn], v[maxn];

int main()
{
    scanf("%d %d", &n, &c);
    for(int i = 1; i <= n; ++i){
        scanf("%d %d", &w[i], &v[i]);
    }
    for(int i = n; i >= 1; --i){
        for(int j = 0; j <= c; ++j){
```

```
            f[i][j] = f[i+1][j];        //因为f[i][j]是从f[i+1][j]来的，所以要倒叙遍历n~1
            if(j >= w[i])   f[i][j] = max(f[i][j], f[i+1][j - w[i]] + v[i]);
        }
    }
    int cur = c;
    for(int i = 1; i <= n; ++i){        //因为要字典序最小，所以要正序遍历1~n
        if(cur >= w[i] && f[i][cur] == f[i+1][cur - w[i]] + v[i]){
            printf("%d ", i);           //此时取i能保证价值最大且字典序最小
            cur -= w[i];
        }
    }
}
```

分类: 动态规划_背包dp

标签: 动态规划