背包问题笔记

标签（空格分隔）： dp 背包 学习中的一些疑问解决

# [[TOC]] 目录

# 1. 1_01背包问题

## 1.1暴力搜索写法：

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1e3 + 10;
int inf = -(1e9 + 10);
int n, m;
int f[maxn][maxn];
int v[maxn], w[maxn];
int dfs(int now, int have) //分别表示第几个，以及现在的背包重量是多少。
{
    if (have > m)
        return inf;
    if (now > n)
        return 0;
    return max(dfs(now + 1, have), dfs(now + 1, have + w[now]) + v[now]);
}
int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0), cout.tie(0);
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
        cin >> w[i] >> v[i];、
    cout<<dfs(1,0)<<'\n';

}
```

## 1.11回顾tips

1. dfs函数的意义是，当前状态之后的最优状态。也可以理解为子问题：（背包已经使用容量have，从第now个物件开始选择的最优解。）该问题的解。
2. 可以看到，该问题的解是和其他状态相关的。

---

## 1.2记忆化搜素写法：

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1e3 + 10;
int inf = -(1e9 + 10);
int n, m;
int f[maxn][maxn];
```

```
int v[maxn], w[maxn];
int dfs(int now, int have) //分别表示第几个，以及现在的背包重量是多少。
{
    if (have > m)
        return inf;
    if (now > n)
        return 0;
    if (f[now][have] > 0)
        return f[now][have];
    return f[now][have] = max(dfs(now + 1, have), dfs(now + 1, have + w[now]) +
v[now]);
}
int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0), cout.tie(0);
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
        cin >> w[i] >> v[i];
    cout << dfs(1, 0) << '\n';
}
```

## 1.21回顾tips:

1. 相对于暴力搜索，将每一个人函数作为一颗树的节点，发现、一整个搜索树中，有一些状态会是重复的。
2. 将某个状态的计算结果保存，可以做到一定的优化。减少重复子树的计算。

# 1.3抽象出二重循环，计算各个子问题的解(正推如下)。

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1e3 + 10;
int w[maxn];
int v[maxn];
int f[maxn][maxn];
int main()
{
    int n, m;
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
        cin >> w[i] >> v[i];
    for (int i = 1; i <= n; i++)
        for (int j = 0; j <= m; j++)
        {
            f[i][j] = f[i - 1][j];
            if (j >= w[i])
                f[i][j] = max(f[i][j], f[i - 1][j - w[i]] + v[i]);
        }
    int ans = 0;
    for (int i = 1; i <= m; i++)
        ans = max(ans, f[n][i]);
    cout << ans << '\n';
}
```

- 详解如下
  - $d_{i,j}$定义为问题（背包容量为j，从第1到第i个物品开始选择方案的最大价值）的解；
  - 满足题意得情况下;
    $d_{i,j} = max(d_{i-1,j}, d_{i-1,j-w_i} + v_i)$
  - 一般疑惑的点;
    - 是什么，为什么，怎么样。
    - 理解这种转移，并且接受这一种转移。
      - 计算一个$d_{ij}$时，要用到规模更小的解，我们先保证用到的特征函数是正确的解。至于怎么得到正确的解是另外一个问题。
      - 对于待解决的子问题，一个方法是枚举所有的解结构，但是我们已经记录下所有比它规模更小的子问题。
      - 只关注第一步的选择。选择$a_i$，不选择$a_i$.然后分别有不同的空间选择前$i-1$个物品。分别是$j-w_i$和$j$。
      - 所以问题转变为，剩下的这些空间选择前i-1个物品的最优解。而这个最优解我们已经计算完成。
      - 上述问题可以等价为，有某大小的背包去选择装前$i-1$物品的最优解的子问题的最优解问题。（背包空间大小和剩余空间大下一样。）
      - 可以知道第一个问题解都是第二个等效的问题的合法解。只要它们的花费小于资源即可。而两种问题的资源是一样多的。
      - 综上来理解状态转移方程的可行性。
      - 而初始化时我们保证了规模最小的一类子问题的解的正确性。（规模越小越容易计算）这样，由这个迁移方式得到的所有子问题的解都是合法的。
    - 认识$d_{i,j}$是什么。
      - 用体积为j的背包，选择前i个物品的方案中可得的最大价值。（对所有的方案的结果直接或者间接的做了max比较运算，该运算系统是含幺的。$max(a,a)=a$）.
      - 本质上就是对一些量做了取$max$运算。在二重循环中由于迭代是与一个线性的。因此要把着一些其它情况的也计算出来。只需要知道，它们最终计算的结果就是相应解。和物理一样掌握了规律就可以描述，预测描述模型。虽真正模拟这一个过程中有一些量不会涉及，但是它们实实在在的对应一个问题，以及存储着它们的解。
    - 就看这一个dfs体系，实际上有一些点根本不会出现。有一些子问题都是抽象推广出来的。该怎么理解它们的存在、作用?
      - 同上，可以通过小规模问题的解，计算出连续的，更大规模的问题的解。

# 1.4对简单二重循环的优化。（滚动数组，常数优化）

```
#include <bits/stdc++.h>
using namespace std;
int n, m;
const int maxn = 1003;
int f[maxn];
int v[maxn], w[maxn];
int main()
{
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
        cin >> w[i] >> v[i];
    for (int i = 1; i <= n; i++)
        for (int j = m; j >= w[i]; j--)
```

```
            f[j] = max(f[j], f[j - w[i]] + v[i]);
    cout << f[m] << '\n'; //第一个疑问点。为什么f[m]就是ans;
}
```

## 1.41回顾tips:

- 可以发现，在状态得转移中，就这一个表结构上有如下规律：
  - 满足题意得情况下：
    1．第i行得转移只用到了第i-1行的数据。
    2．只要保证可以获取某个数据转移计算需要用到的i-1行中的数据。每个dp的计算是独立的，即先后关系没有影响，只要保证转移用到的状态可获取就行了。
  - 做出如下改进：
    1．发现一个特征函数转移对应的i-1行的特征函数，都是在它之前的。第二重循环中从后往前计算指标函数。这样可以保证，前面的未计算数据可以用到转移过程中需要的特征函数因为改变只发生在之后。
    2．可以修正第二重循环的上界为$w_i$，更前面的只能够选择$f_{i-1,j}$就是相当于没有变化。
- 关于最后$f_m$就是答案结论的证明。
  - 最直白的理解： 从它的意义上看，我们确保了转移后得到的指标函数值，就是对应子问题的最优解。这样该问题不要求严格装满的题意下：$f_m >= f_k, f_k = max\_v$。
  - 就正推的转移方式上看
    - 假设$f_k = max_value$
    - 选择第一个物品后的$f_i$的分布情况是$0\ldots,0,v_1,v_1\ldots.$
    - 按照转移规则，第二次选择
      $f_k => max(if(k >= w_i)f_{k-w_i}, f_k)$
      $f_m => max(if(m >= w_i)f_{m-w_i}, f_m)$
      相对于$f_m$的转移对象来说，$f_k$的转移对象一直位于更前面的位置。而上一个序列的情况是是单调的，因此，经过这一次转移后，$f(x)$也保持了单调性。
    - 数学归纳法得到了完成转移之后，$f_m >= f_k$
- 2022 9 1