一、实验题目

编写满足下面要求的0-1背包算法，（必做）

0-1背包问题：物品*i*或者被装入背包，或者不被装入背包，设*xi*表示物品*i*装入背包的情况，则当*xi*=0时，表示物品*i*没有被装入背包，*xi*=1时，表示物品*i*被装入背包。根据问题的要求，有如下约束条件和目标函数：

$$max\sum_{i=1}^{n}v_ix_i$$

$$s.t.\begin{cases}\sum_{i=1}^{n}w_ix_i\le C\\ x_i\in\{0,1\}\quad(1\le i\le n)\end{cases}$$

寻找一个满足约束条件式1，并使目标函数式2达到最大的解向量*X*=(*x*1, *x*2, …, *xn*)。

（1）证明该问题满足最优子结构；

（2）给出递推式子；

（3）基于动态规划实现算法；

（4）分析算法的复杂度。

实现对上述0-1背包问题的改进，（二选一）

（1）上述算法要求所给物品的重量必须是整数，而实际处理问题时无法避免物品的重量是小数的情况，试编写一个能够处理重量为小数的情况。

（2）当背包容量c很大时，算法需要计算的时间很大，该算法的时间复杂度在c>2^n时为n*c; 在算法中，注意到m(i,j)是阶梯状单调不减函数。请试图改进该算法，提高算法复杂度。

二、算法原理

2.1 证明该问题满足最优子结构

用反证法证明：

设$(x_1, x_2, \ldots, x_n)$是所给0/1背包问题的一个最优解，则$(x_1, x_2, \ldots, x_{n-1})$是下面一个子问题的最优解：

$$max \sum_{i=1}^{n-1} v_i x_i$$

$$s.t. \begin{cases} \sum_{i=1}^{n-1} w_i x_i \leq C - w_n x_n \\ x_i \in \{0, 1\} \quad (1 \leq i \leq n-1) \end{cases}$$

如若不然，设$(y_1, y_2, \ldots, y_{n-1})$是上述子问题的一个最优解，及

$$max \sum_{i=1}^{n-1} v_i x_i < max \sum_{i=1}^{n-1} v_i y_i$$

$$max \sum_{i=1}^{n-1} v_i x_i + v_n x_n = max \sum_{i=1}^{n} v_i x_i < max \sum_{i=1}^{n-1} v_i y_i + v_n x_n$$

得到$(x_1, x_2, \ldots, x_n)$ 不 是所给0/1背包问题的一个最优解，与题设矛盾。

因此，该问题满足最优子结构。

2.2 求解递推式

设 DP 状态$f(i, j)$为在**只能放前 i 个物品的情况下，容量为 j 的背包所能达到的最大总价值**。

考虑转移。假设当前已经处理好了前 $i-1$ 个物品的所有状态，那么对于第 i 个物品，当其不放入背包时，背包的剩余容量不变，背包中物品的总价值也不变，故这种情况的最大价值为 $f(i-1, j)$；当其放入背包时，背包的剩余容量会减小 w_i，背包中物品的总价值会增大 v_i，故这种情况的最大价值为 $f(i-1, j-w_i) + v_i$。

由此可以得出状态转移方程：

$$f(i,j)=\begin{cases}0 & (i=0)\vee(j=0)\\ f(i-1,j) & j<w_i\\ max\{f(i-1,j-w_i)+v_i,f(i-1,j)\} & j\geq w_i\end{cases}$$

2.3 记录解向量

事实上，根据题目要求，仅仅使用$f(i,j)$求出最大价值是不够的，还需要记录最大的解向量$X=(x_1,x_2,\ldots,x_n)$。我们可以考虑$f(i,j)$的定义，从$f(N,M)$出发，倒退求解向量：

- 初始$i=N,j=M$
- 若$f(i,j)>f(i-1,j)$，则第i个物品被选中，用于更新$f(i,j)$，$i=i-1,j=j-w_i$
- 若$f(i,j)=f(i-1,j)$，则第i个物品没有被选中，$i=i-1,j$保持不变

```
for (int i = N, j = M; i >= 1; i--)
{
    if (F[i][j] > F[i-1][j])
    {
        X[i] = 1;
        j -= W[i];
    }
    else
        X[i] = 0;
}
```

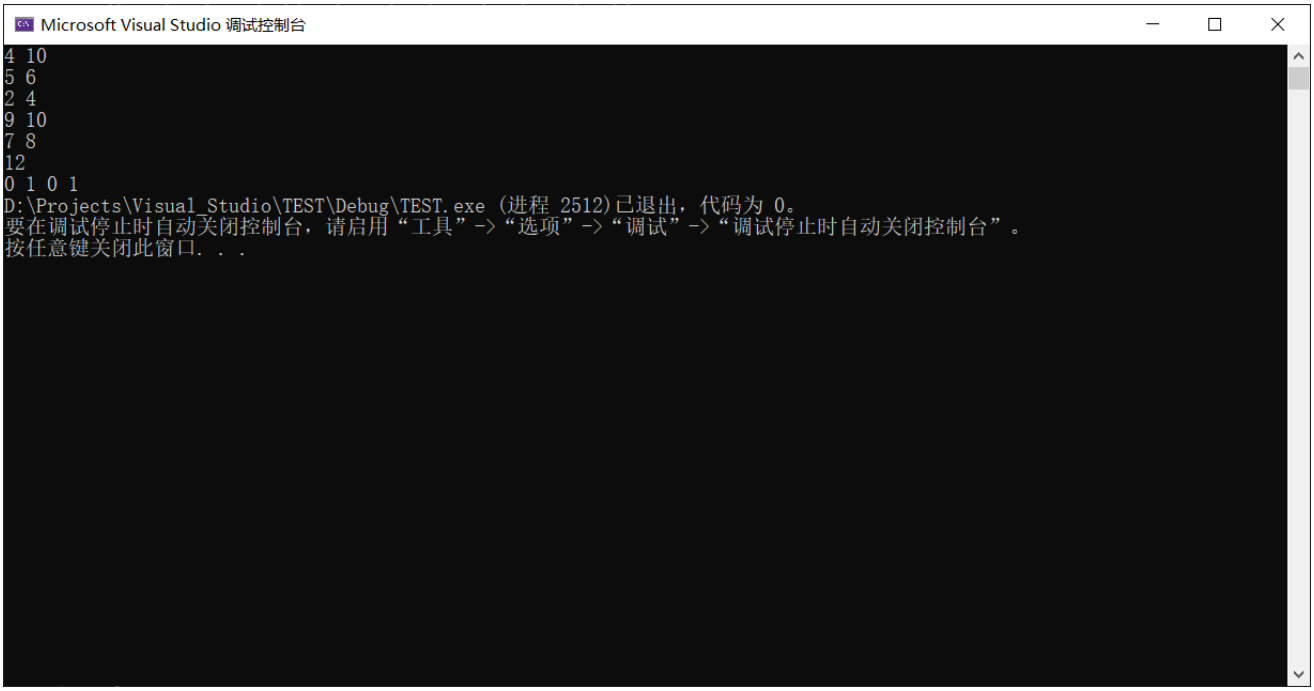
三、算法实现

两层循环，第一层循环物品，第二层循环重量。

```cpp
#include <iostream>
#define gmax(_a, _b) ((_a) > (_b) ? (_a) : (_b))
using namespace std;
const int maxn = 1000, maxm = 10000;
int W[maxn + 10], V[maxn + 10], X[maxn + 10], F[maxn][maxm];
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] = gmax(F[i][j], F[i - 1][j - W[i]] +
V[i]);
        }
    for (int i = N, j = M; i >= 1; i--)
    {
        if (F[i][j] > F[i-1][j])
        {
            X[i] = 1;
            j -= W[i];
        }
        else
            X[i] = 0;
    }
    cout << F[N][M] << endl;
    for (int i = 1; i <= N; ++i) cout << X[i] << " ";
    return 0;
}
```

测试：4个物品，背包容量为10，

$w_1 = 5, v_1 = 6; w_2 = 2, v_2 = 4; w_3 = 9, v_3 = 10; w_4 = 7, v_4 = 8$



最大价值为12，解向量$X=[0,1,0,1]$

四、算法分析

N为物品总数，M为背包容量

时间复杂度：两层循环，$O(NM)$

空间复杂度：存储状态的辅助空间F，为$O(NM)$

五、优化分析

5.1 空间复杂度的优化

由于对 f_i 有影响的只有 f_{i-1} ，可以通过滚动数组的方式去掉第一维，直接用 f_j 来表示处理到当前物品时背包容量为 j 的最大价值，得出以下方程：

$$f(j)=\begin{cases}0 & (j=0)\\ f(j) & j<w_i\\ max\{f(j-w_i)+v_i,f(j)\} & j\geq w_i\end{cases}$$

需要注意的是，在使用滚动数组时， j 的循环顺序必须是从大到小，因为使用滚动数组本质上是把 $f(j-w_i)$ 当作 $f(i-1,j-w_i)$ 使用，因此必须保证$f(j-w_i)$ 还没有被当前第 i 个物品更新过（否则会引起第 i 个物品的重复放入，变成多重或完全背包）。而在从大到小循环 j 时，被更新过的只有 $k>j$ 的 $f(k)$ ，用于更新$f(j)$ 的 $f(j-w_i)$ 一定还没有被更新，等同于 $f(i-1,j-w_i)$。

代码如下：

```
#include <iostream>
using namespace std;
const int maxn = 1000, maxm = 10000;
int W[maxn + 10], V[maxn + 10], F[maxm];
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 = M; j >= W[i]; j--)
        if (F[j - W[i]] + V[i] > F[j]) F[j] = F[j - W[i]] +
V[i];
    cout << F[M];
    return 0;
}
```

空间复杂度从$O(NM)$优化到$O(M)$

5.2 重量为小数

1. 浮点数直接转为整数

   如果已知物品重量的浮点精度p较低，则可以直接乘上 10^p，将其转为整数，再按照0-1背包的方法计算，时间复杂度为 $O(NM10^p)$。进一步思考，可以求出重量数据组的最大公约数，所有数据除掉该最大公约数，可减少枚举规模，进一步减小时间复杂度。

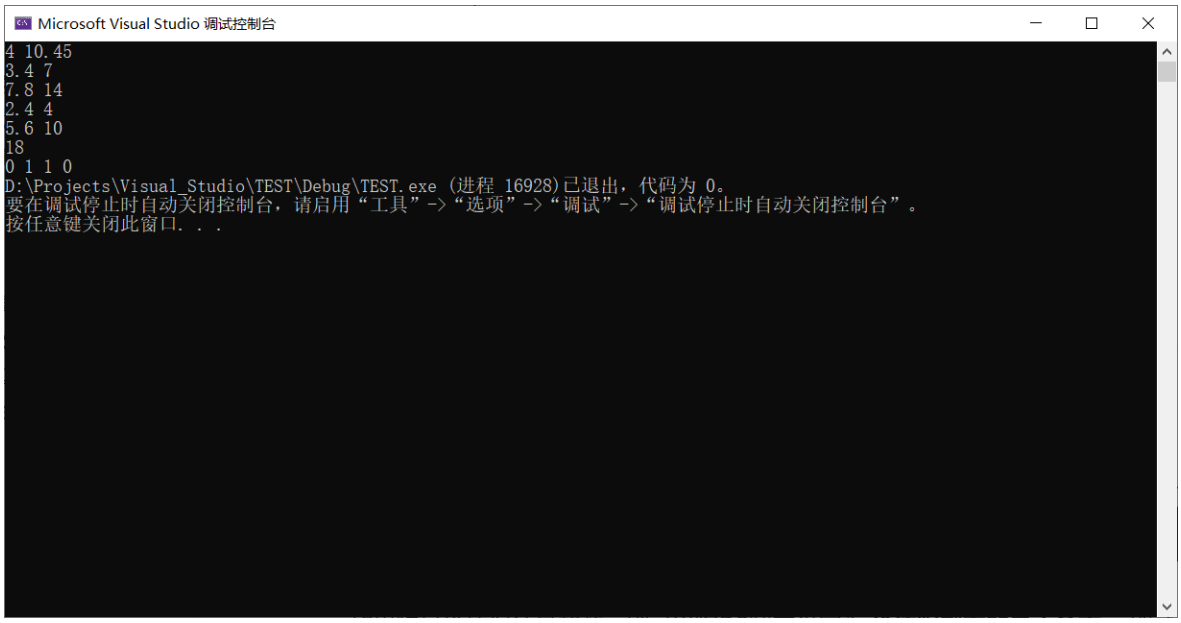
   代码如下（假设精确到小数点后两位）：

```
#include <iostream>
#define gmax(_a, _b) ((_a) > (_b) ? (_a) : (_b))
using namespace std;
const int maxn = 1000, maxm = 10000;
int W[maxn + 10], V[maxn + 10], X[maxn + 10], F[maxn]
[maxm];
int main()
{
    int N, M;
    double x;
    cin >> N >> x;
    M = x * 100;
    for (int i = 1; i <= N; i++)
    {
        cin >> x >> V[i];
        W[i] = x * 100;
    }
    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] = gmax(F[i][j], F[i - 1][j -
W[i]] + V[i]);
        }
    for (int i = N, j = M; i >= 1; i--)
    {
        if (F[i][j] > F[i-1][j])
```

```
        {
            X[i] = 1;
            j -= W[i];
        }
        else
            X[i] = 0;
    }
    cout << F[N][M] << endl;
    for (int i = 1; i <= N; ++i) cout << X[i] << " ";
    return 0;
}
```

测试：4个物品，背包容量为10.45，

$w_1 = 3.4, v_1 = 7; w_2 = 7.8, v_2 = 14; w_3 = 2.4, v_3 = 4; w_4 = 5.6, v_4 = 10$



最大价值为18，解向量$X = [0, 1, 1, 0]$

2. 求价值限制下的最小负重

如果物品价值保证是整数，则可以将DP状态设定为 **价值限制下的最小负重**，即 f_j 来表示处理到当前物品时 **背包价值为 j 的负重最小值**，转移方程为：

$$f(j) = min\{f(j - v_i) + w_i, f(j)\}$$

初始令所有 $f(j)=+\infty, f(0)=0$，然后做与0-1背包类似的双层循环：

```
for (int i = 1; i <= N; i++)
        for (int j = maxv; j >= V[i]; j--)
        if (F[j - V[i]] + W[i] - eps < F[j])
            F[j] = F[j - V[i]] + W[i];
```

其中：`maxv`等于所有物品价值之和，为背包能装下物品价值的理论上界。

完成状态转移后，从maxv到0寻找满足 $f(j)<=M$ 的最大 j，该 j 即为答案。

其时间复杂度为两层循环次数，$O(NMaxV)$

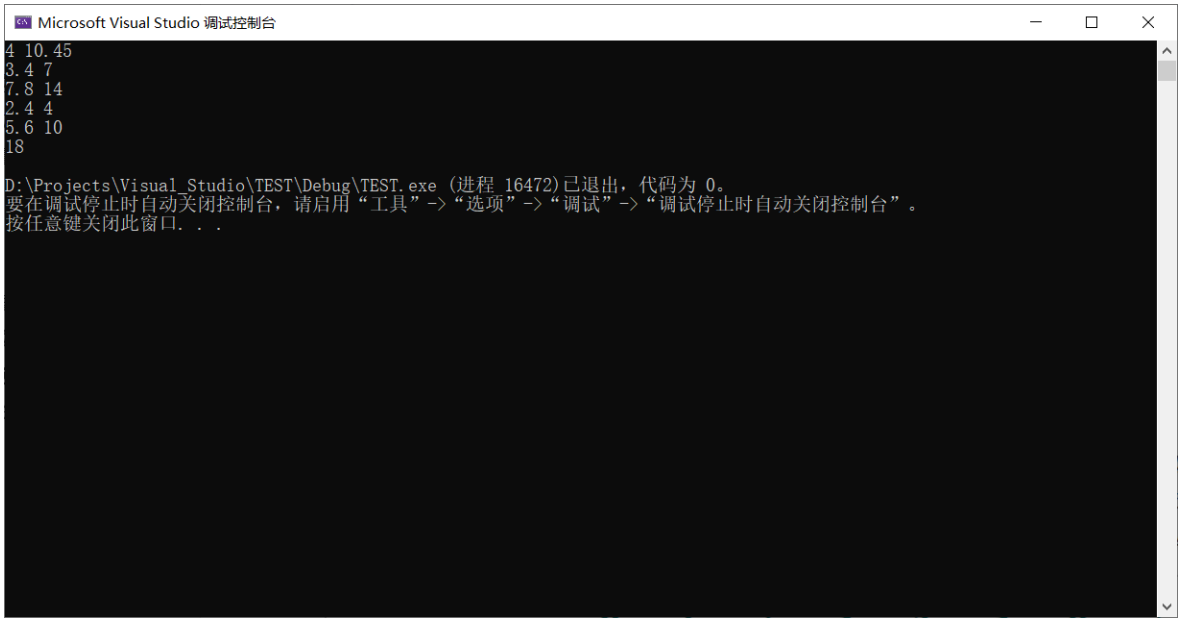
完整代码如下：

```
#include <iostream>
using namespace std;
const int maxn = 1000, maxm = 10000;
const double INF = 1e7;
const double eps = 1e-6;
double W[maxn + 10], F[maxm];
int V[maxn + 10];
int main()
{
    int N;
    double M;
    int maxv = 0;
    cin >> N >> M;
    for (int i = 1; i <= N; i++)
    {
        cin >> W[i] >> V[i];
        maxv += V[i];
    }
    for (int i = 1;  i <= maxv; ++i) F[i] = INF;
    F[0] = 0;
    for (int i = 1; i <= N; i++)
        for (int j = maxv; j >= V[i]; j--)
        if (F[j - V[i]] + W[i] - eps < F[j]) F[j] =
F[j - V[i]] + W[i];
```

```
    for (int i = maxv; i >= 0; i--)
        if (F[i] - eps <= M)
        {
            cout << i << endl;
            break;
        }
    return 0;
}
```

测试：4个物品，背包容量为10.45，

$w_1 = 3.4, v_1 = 7; w_2 = 7.8, v_2 = 14; w_3 = 2.4, v_3 = 4; w_4 = 5.6, v_4 = 10$



最大价值同样为18，与方法1结果相同

3. 直接枚举

考虑每样物品装或不装，枚举所有情况，复杂度为 $O(2^N)$

5.3 大容量背包

在背包容量过大时，从0到M遍历背包容量会产生大量的“无用功”，每次考虑一个新的物品时，从F[i-1]到F[i]只有少数点发生了更新，其余大量点位只是执行了从F[i-1]到F[i]的拷贝，注意到 $F[i][j]$ 是阶梯状单调不减函数。因此我们可以用跳跃点来记录一组数据，仅针对跳跃点进行F数组的更新，从而避免大量无用遍历。

		0	1	2	3	4	5	6	7	8	9	10
	0	0	0	0	0	0	0	0	0	0	0	0
w_1=2 v_1=6	1	0	0	6	6	6	6	6	6	6	6	6
w_2=2 v_2=3	2	0	0	6	6	9	9	9	9	9	9	9
w_3=6 v_3=5	3	0	0	6	6	9	9	9	9	11	11	14
w_4=5 v_4=4	4	0	0	6	6	9	9	9	10	11	13	14
w_5=4 v_5=6	5	0	0	6	6	9	9	12	12	15	15	15

用 $F[i][j]$ 由0-1背包原始算法可知，跳跃点有以下三种情况：

1. 跳跃点集$p[i-1]$:由$F[i-1][j]$决定,对应于不装第i个物品的所有装包情形
2. 跳跃点集$q[i-1]$:由$F[i-1][j-w_i]+v_i$ 决定,对应于装上第i个物品的所有情形，

   $q[i-1]=\{(j+w_i,F[i-1][j]+v_i)|(p,F[i-1][j])\in p[i-1],p+w_i\leq M\}$

   定义上述运算为$q[i-1]=p[i-1]\oplus(w_i,v_i)$
3. 受控跳跃点 :设 (x_1,v_1) 和 (x_2,v_2)是 $p[i-1]\cup q[i-1]$ 中的2个跳跃点,则当$x2\geq x1$且

   $v_2<v_1$时，(x_2,v_2)受控于(x_1,v_1) ,从而 (x_2,v_2)不是$p[i]$中的跳跃点。

   由于受控跳跃点是不正确的跳跃点，因此得到转移方程

   $p[i]=p[i+1]\cup q[i+1]-\{$受控跳跃点$\}$

计算跳跃点的过程如下：

1. 初始化跳跃点集

   $p[n].insert((0,0)); q[n] = p[n] \oplus (w[n], v[n])$

2. 从n到1遍历，顺序执行以下三步操作：

   1) $p[i] = p[i+1] \cup q[i+1]$

   2) 遍历$p[i]$每个跳跃点，删除受控跳跃点。

   3) $q[i] = p[i] \oplus (w[i], v[i])$

计算结束后，$p[1] \cup q[1] - \{$受控跳跃点$\}$ 中的最高点即为结果。

用`struct point`结构体存储跳跃点，并重载其`<`符号，使点集在集合中依照横坐标从小到大的顺序排列，在横坐标相同时，纵坐标大的排在前面，方便删除受控跳跃点

```
struct point
{
    int w, v;
    point(int _w = 0, int _v = 0) : w(_w), v(_v) {};
    //重载小于号 ，使点集在集合中依照横坐标从小到大的顺序排列，在横坐标
相同时，纵坐标大的排在前面，方便删除受控跳跃点
    bool operator<(const point& rhs) const {
        if (w == rhs.w)
            return v < rhs.v;
        return w < rhs.w;
    };
};
```

计算$p[i+1] \cup q[i+1] - \{$受控跳跃点$\}$的过程为：先求出$p[i+1] \cup q[i+1]$，存储在临时集合temp中。然后在temp中从小到达遍历一遍，存下当前遍历到的最大纵坐标`MaxV`，检查每个跳跃点。如果当前跳跃点的纵坐标小于$MaxV$，则对于当前点(x, v)必然存在一个$x_1 <= x$，$v_1 > v$的点x_1, v_1，因此当前点属于受控跳跃点。

将所有非受控跳跃点加入p集合中。

```
void Union(set <point>& p, set <point>& q)
{
    set<point> temp;
    //使用stl的集合求并集函数set_union,第1、2个参数为第一个集合的范围，第3、4个参数为第二个集合的范围，最后一个参数为计算结果存储位置，这里我们将p[i+1]∪q[i+1]存储在temp集合中
    set_union(begin(p), end(p), begin(q), end(q), inserter(temp, temp.begin()));
    set<point>::iterator it, jt;
    //清空p集合，准备用来存储temp去除受控跳跃点后的结果
    p.clear();
    //遍历temp中所有元素
    for (it = temp.begin(); it != temp.end(); it++)
    {
        int ok = 1;
        point x = *it;
        for (jt = temp.begin(); jt != temp.end(); jt++)
        {
            point y = *jt;
            if (y.w <= x.w && y.v > x.v)
                ok = 0;
        }
        if (ok)
            p.insert(x);
    }
}
```

计算$p[i] \oplus (w[i], v[i])$的过程为：遍历p集合中的每个跳跃点，只要其横坐标加上w_i不超出M就可以加入q集合。

```
void Oplus(set <point>& p, set <point>& q, int x)
{
    q.clear();
    set<point>::iterator it;
    for (it = p.begin(); it != p.end(); it++)
    {
        point n = *it;
        if (n.w + W[x] <= M)
            q.insert(point(n.w + W[x], n.v + V[x]));
    }
}
```

集合的运算使用STL中的set，且p, q集合可以使用“滚动集合”，优化空间复杂度（与5.1中优化同理），完整代码如下：

```
#include <iostream>
#include <set>
#include <algorithm>
#define Reg register
#define FOR(_i,_a,_b) for (Reg int _i = (_a) ; _i <= (_b) ;
_i++)
#define gmax(_a, _b) ((_a) > (_b) ? (_a) : (_b))
using namespace std;
const int maxn = 1000, maxm = 10000;
int W[maxn + 10], V[maxn + 10];
int N, M;
struct point
{
    int w, v;
    point(int _w = 0, int _v = 0) : w(_w), v(_v) {};
    bool operator<(const point& rhs) const {
        if (w == rhs.w)
            return v < rhs.v;
        return w < rhs.w;
    };
```

```
};
set <point> p, q;

void Oplus(set <point>& p, set <point>& q, int x)
{
    q.clear();
    set<point>::iterator it;
    for (it = p.begin(); it != p.end(); it++)
    {
        point n = *it;
        if (n.w + W[x] <= M)
            q.insert(point(n.w + W[x], n.v + V[x]));
    }
}

void Union(set <point>& p, set <point>& q)
{
    set<point> temp;
    set_union(begin(p), end(p), begin(q), end(q),
inserter(temp, temp.begin()));
    set<point>::iterator it, jt;
    p.clear();
    for (it = temp.begin(); it != temp.end(); it++)
    {
        int ok = 1;
        point x = *it;
        for (jt = temp.begin(); jt != temp.end(); jt++)
        {
            point y = *jt;
            if (y.w <= x.w && y.v > x.v)
                ok = 0;
        }
        if (ok)
            p.insert(x);
    }
}

int main()
```

```
{
    cin >> N >> M;
    FOR(i, 1, N)
        cin >> W[i] >> V[i];
    p.clear();
    p.insert(point(0, 0));

    FOR(i, 1, N)
    {
        Oplus(p, q, i);
        Union(p, q);
    }
    set<point>::iterator it;
    int maxV = 0;

    for (it = p.begin(); it != p.end(); it++)
    {
        point x = *it;
        if (x.v > maxV)
            maxV = x.v;
    }
    cout << maxV;
}
```

时间复杂度分析

每次考虑一个新的物品，修改跳跃点集时，删除受控跳跃点操作需要遍历一遍跳跃点集，因此每次遍历的时间复杂度为$O(\text{点数目})$。

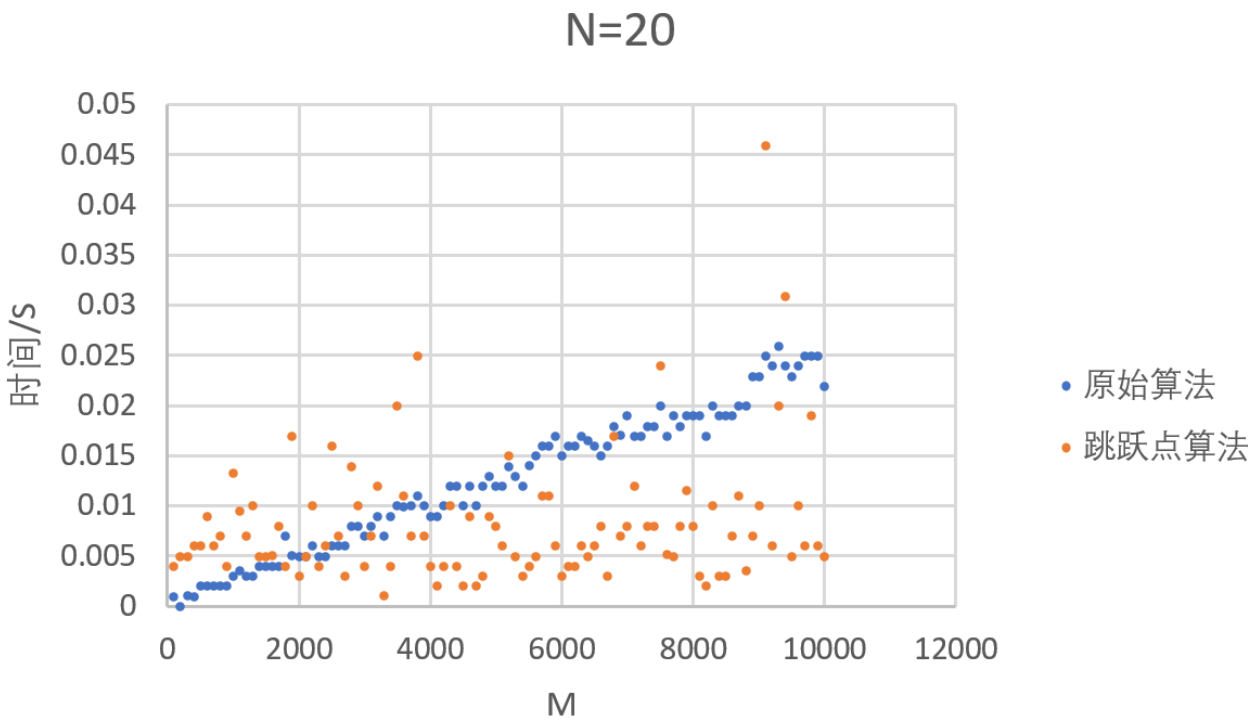
对原跳跃点集中的每个点有两种操作方式：保留该点或加上新物品的重量和价值进行位移。因此在最坏情况下，每一次最多会将跳跃点集的点数目翻倍。又由于跳跃点集最多只可能有最大重量M 个点（否则，必然出现同一重量不同价值总和的跳跃点，违背定义），即考虑第i个物品时跳跃点集中最多有$Min\{2^i, M\}$个点

总消耗时间$T(N,M)=\sum_{i=1}^{N}T(Min\{2^i,M\})=O(Min\{2^N,NM\})$

算法测试

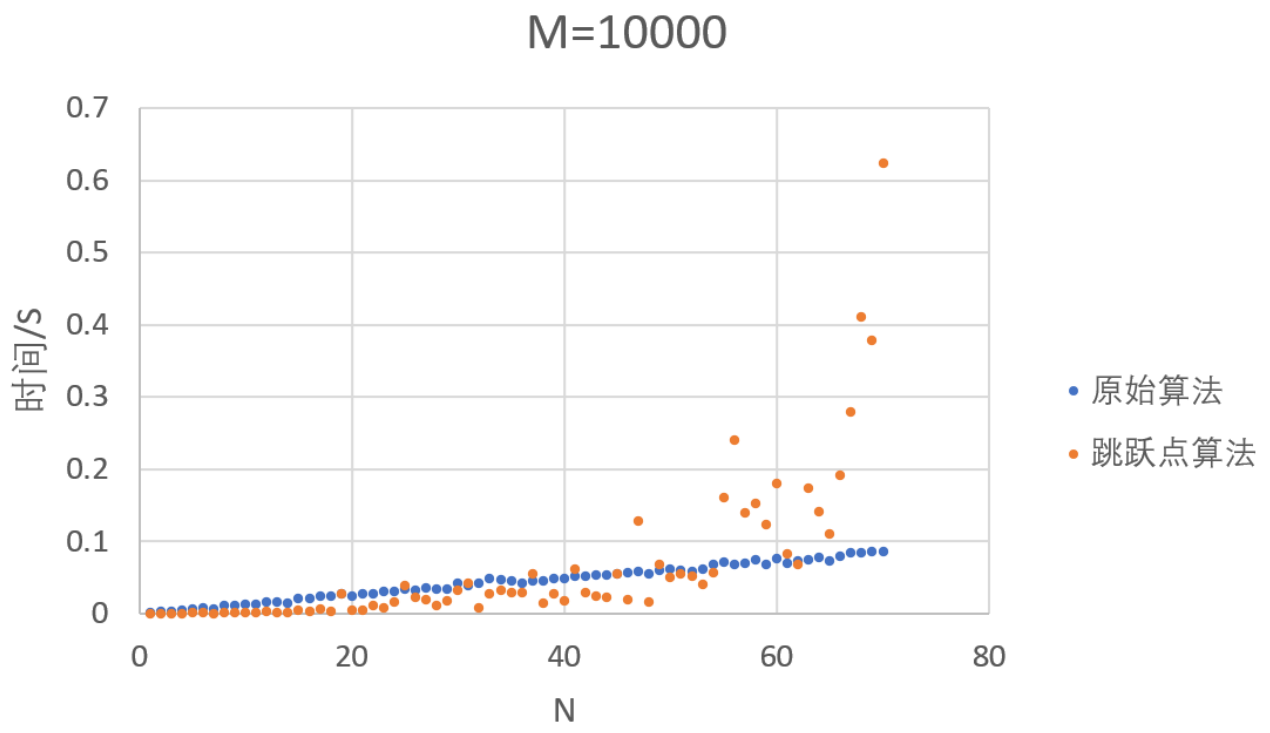
生成随机数据，同时输入原始0-1背包算法与跳跃点算法。

在物品数量N一定时，改变背包容量M，测试两种算法运行效率：



结果显示，在背包容量较小时，原始算法效率高；在背包容量较大时，跳跃点算法效率高。

在背包容量M一定时，改变物品数量N，测试两种算法运行效率：



结果显示，在物品数量较少时，跳跃点算法效率高；在物品数量较多时，原始算法效率高。

详细数据如下表：

N	M	原始算法结果	TIME1(S)	跳跃点算法结果	TIME2(S)
20	100	2443	0.000992	2443	0.003989
20	200	4333	0	4333	0.004985
20	300	3794	0.000998	3794	0.004986
20	400	2930	0.000983	2930	0.005984
20	500	2965	0.001995	2965	0.005983
20	600	2747	0.001995	2747	0.008976
20	700	3272	0.001995	3272	0.005984
20	800	2849	0.001995	2849	0.006981
20	900	2128	0.001995	2128	0.003989
20	1000	2957	0.002992	2957	0.013319
20	1100	3986	0.003496	3986	0.009475
20	1200	2569	0.002993	2569	0.006981
20	1300	3457	0.002992	3457	0.009974

N	M	原始算法结果	TIME1(S)	跳跃点算法结果	TIME2(S)
20	1400	4212	0.003989	4212	0.004987
20	1500	3642	0.00399	3642	0.004986
20	1600	3225	0.00399	3225	0.004997
20	1700	4553	0.00399	4553	0.007979
20	1800	2668	0.006983	2668	0.003989
20	1900	3943	0.005013	3943	0.016924
20	2000	3106	0.004987	3106	0.002992
20	2100	1414	0.004987	1414	0.004987
20	2200	4131	0.005983	4131	0.009982
20	2300	2737	0.004987	2737	0.003989
20	2400	2886	0.004987	2886	0.006007
20	2500	3344	0.005995	3344	0.015957
20	2600	2956	0.005984	2956	0.007027
20	2700	3761	0.005984	3761	0.002992
20	2800	2223	0.007988	2223	0.013953
20	2900	4557	0.007978	4557	0.009973
20	3000	2999	0.00698	2999	0.003992
20	3100	2476	0.007978	2476	0.006981
20	3200	3558	0.008976	3558	0.011967
20	3300	2009	0.00698	2009	0.000997
20	3400	3634	0.008976	3634	0.00399
20	3500	4050	0.009972	4050	0.019947
20	3600	6133	0.009965	6133	0.01097
20	3700	4965	0.009973	4965	0.006982
20	3800	5615	0.01097	5615	0.024933
20	3900	2230	0.009972	2230	0.006982
20	4000	2214	0.008974	2214	0.00399
20	4100	2962	0.008974	2962	0.001995

N	M	原始算法结果	TIME1(S)	跳跃点算法结果	TIME2(S)
20	4200	2131	0.009972	2131	0.003991
20	4300	4023	0.011972	4023	0.009992
20	4400	2894	0.011967	2894	0.003988
20	4500	2257	0.009974	2257	0.001985
20	4600	3805	0.011953	3805	0.008974
20	4700	2786	0.009973	2786	0.001994
20	4800	4523	0.011968	4523	0.002993
20	4900	4468	0.012964	4468	0.008976
20	5000	2403	0.011968	2403	0.007978
20	5100	4061	0.011968	4061	0.005983
20	5200	3200	0.013962	3200	0.014961
20	5300	3149	0.012965	3149	0.004987
20	5400	2634	0.011968	2634	0.002992
20	5500	2867	0.013984	2867	0.00399
20	5600	4330	0.014959	4330	0.004987
20	5700	3224	0.015973	3224	0.010971
20	5800	4362	0.015958	4362	0.010971
20	5900	4144	0.016954	4144	0.005984
20	6000	2172	0.014959	2172	0.002993
20	6100	3530	0.015957	3530	0.003989
20	6200	3915	0.015957	3915	0.003991
20	6300	3159	0.016955	3159	0.005984
20	6400	3409	0.016502	3409	0.004987
20	6500	2892	0.015958	2892	0.006019
20	6600	2923	0.01496	2923	0.007978
20	6700	2598	0.015957	2598	0.002992
20	6800	3799	0.017953	3799	0.016954
20	6900	4154	0.016998	4154	0.006981

N	M	原始算法结果	TIME1(S)	跳跃点算法结果	TIME2(S)
20	7000	3842	0.018949	3842	0.007978
20	7100	2582	0.016991	2582	0.011967
20	7200	2792	0.016954	2792	0.005984
20	7300	4859	0.017961	4859	0.007982
20	7400	4806	0.017953	4806	0.00798
20	7500	4202	0.019947	4202	0.023937
20	7600	2592	0.016954	2592	0.005197
20	7700	4897	0.018949	4897	0.004986
20	7800	4294	0.017953	4294	0.007978
20	7900	2824	0.018951	2824	0.011485
20	8000	3927	0.01895	3927	0.007979
20	8100	2221	0.01895	2221	0.002992
20	8200	2218	0.016955	2218	0.001995
20	8300	3870	0.019971	3870	0.009973
20	8400	3267	0.018951	3267	0.002992
20	8500	2380	0.018949	2380	0.002991
20	8600	3345	0.018949	3345	0.006981
20	8700	3035	0.019946	3035	0.010971
20	8800	3546	0.019947	3546	0.003484
20	8900	4689	0.022939	4689	0.006982
20	9000	3387	0.022939	3387	0.009981
20	9100	3959	0.024934	3959	0.045904
20	9200	4044	0.023939	4044	0.005984
20	9300	4911	0.025931	4911	0.019947
20	9400	2671	0.023936	2671	0.030917
20	9500	4111	0.022939	4111	0.004986
20	9600	4693	0.023936	4693	0.009973
20	9700	4593	0.024934	4593	0.005984

N	M	原始算法结果	TIME1(S)	跳跃点算法结果	TIME2(S)
20	9800	3980	0.024951	3980	0.018979
20	9900	2971	0.024933	2971	0.005984
20	10000	3385	0.021942	3385	0.004987

N	M	原始算法结果	TIME1(S)	跳跃点算法结果	TIME2(S)
1	10000	145	0.002013	145	0
2	10000	216	0.002989	216	0
3	10000	923	0.002991	923	0
4	10000	1106	0.004985	1106	0
5	10000	1458	0.006982	1458	0.000996
6	10000	1194	0.00798	1194	0.001031
7	10000	1310	0.006988	1310	0
8	10000	1307	0.01097	1307	0.001026
9	10000	1454	0.011967	1454	0.001999
10	10000	1502	0.012966	1502	0.001995
11	10000	1787	0.012967	1787	0.001002
12	10000	3369	0.015957	3369	0.004046
13	10000	2550	0.015958	2550	0.000997
14	10000	1996	0.013963	1996	0.000999
15	10000	2966	0.021942	2966	0.004987
16	10000	3057	0.020945	3057	0.00399
17	10000	3517	0.023937	3517	0.006981
18	10000	2699	0.023936	2699	0.002993
19	10000	5856	0.027924	5856	0.027924
20	10000	3964	0.024934	3964	0.004977
21	10000	2333	0.027928	2333	0.004986
22	10000	4391	0.02745	4391	0.01097
23	10000	5336	0.030428	5336	0.007978
24	10000	3193	0.030917	3193	0.016954

N	M	原始算法结果	TIME1(S)	跳跃点算法结果	TIME2(S)
25	10000	3757	0.033418	3757	0.038939
26	10000	3869	0.032418	3869	0.022939
27	10000	6381	0.035903	6381	0.018951
28	10000	5351	0.034907	5351	0.011972
29	10000	5474	0.034907	5474	0.018468
30	10000	5852	0.041888	5852	0.032911
31	10000	3888	0.038897	3888	0.042885
32	10000	2965	0.041902	2965	0.008488
33	10000	4890	0.04887	4890	0.02692
34	10000	5749	0.046873	5749	0.031914
35	10000	5248	0.045877	5248	0.028923
36	10000	4253	0.042886	4253	0.02992
37	10000	5572	0.044881	5572	0.055851
38	10000	5026	0.044884	5026	0.01496
39	10000	5027	0.048899	5027	0.028433
40	10000	5981	0.048871	5981	0.017951
41	10000	4756	0.052846	4756	0.061293
42	10000	5843	0.051861	5843	0.02992
43	10000	5757	0.053856	5757	0.024935
44	10000	5666	0.053856	5666	0.022965
45	10000	4973	0.054853	4973	0.05585
46	10000	5572	0.056853	5572	0.01895
47	10000	4511	0.058843	4511	0.128656
48	10000	3955	0.055852	3955	0.015949
49	10000	7586	0.060847	7586	0.068817
50	10000	5668	0.061835	5668	0.050864
51	10000	5333	0.060838	5333	0.054853
52	10000	4834	0.058288	4834	0.051874

N	M	原始算法结果	TIME1(S)	跳跃点算法结果	TIME2(S)
53	10000	5832	0.061272	5832	0.039894
54	10000	6689	0.067832	6689	0.056294
55	10000	7349	0.072315	7349	0.161569
56	10000	7059	0.067818	7059	0.240821
57	10000	4505	0.069813	4505	0.139181
58	10000	7901	0.074815	7901	0.153543
59	10000	7395	0.068271	7395	0.124131
60	10000	6868	0.075797	6868	0.180217
61	10000	4824	0.069857	4824	0.083276
62	10000	5741	0.073811	5741	0.068306
63	10000	6830	0.07422	6830	0.173558
64	10000	5712	0.078248	5712	0.141171
65	10000	4813	0.073803	4813	0.111209
66	10000	6167	0.079785	6167	0.191488
67	10000	6937	0.084322	6937	0.279253
68	10000	6450	0.083776	6450	0.411215
69	10000	6497	0.086769	6497	0.37799
70	10000	4903	0.085771	4903	0.624311

改变物品数量N，背包容量M，两种算法得到结果均一致，证明正确性。