算法分析与设计

第三次实验报告

PB20050987梁兆懿

实验题目

0-1背包问题求解：假设有n个物品和一个背包，每个物品重量为wi (i=1,2,....,n)，价值为vi (i=1,2, ..., n)，背包最大容量为C，请问该如何选择物品才能使得装入背包中的物品总价值最大? 最大价值为多少?

(1) 请利用分治法来求解该问题，给出最优解值以及求解时间;

(2) 请利用动态规划算法来求解该问题，给出得到最优解值以及求解时间;

(3) 请利用贪心算法来求解该问题，给出得到的最优解值以及求解时间;

(4) 请利用回溯法来求解该问题，给出得到的最优解值以及求解时间;

(5) 请利用分支限界法来求解该问题，给出得到的最优解值以及求解时间;

(6) 请利用蒙特卡洛算法来求解该问题，给出得到的最优解值以及求解时间;

(7) （选做）请利用深度强化学习算法来求解该问题，给出最优解值以及求解时间。

(8) 给定相同输入，比较上述算法得到的最优解值和求解时间。当n比较大的时候，上述算法运算时间可能很长，请在算法中增加终止条件以确保在有限时间内找到最优解的值。

实验代码

在程序开头，首先定义相关的头文件、参数以及全局变量。在实验中，背包容量、最大重量以及物品最大价值使用宏定义，方便在程序测试时修改。相关全局变量则是方便数据的保存：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <queue>
#include <ctime>
using namespace std;

#define SIZE 30  //背包容量
#define C 40     //最大重量
#define MAX_WEIGHT 5  //物品最大重量
#define MAX_VAULE 7   //物品最大价值

int dp[SIZE+1][C+1];         //动态规划存储数组
int w[SIZE];                 //物品的体积
int v[SIZE];                 //物品的价值
int item[SIZE+1];            //最优解情况
double cw=0;                 //当前物品总重
double cv=0;                 //当前物品总价值
int bestp=0;                 //当前最大价值
int bestx[SIZE+1];           //暂时存放最优解
int total=1;                  //解空间中的节点数累计，全局变量
struct nodetype              //队列中的结点类型
```

```
{
    int no;
    int i;
    int w;
    int v;
    int x[SIZE+1];
    double ub;
};
```

实验使用随机数来生成背包物品数据，来保证算法的正确性。而初始化函数在各个算法实现前被调用，用于初始化全局变量：

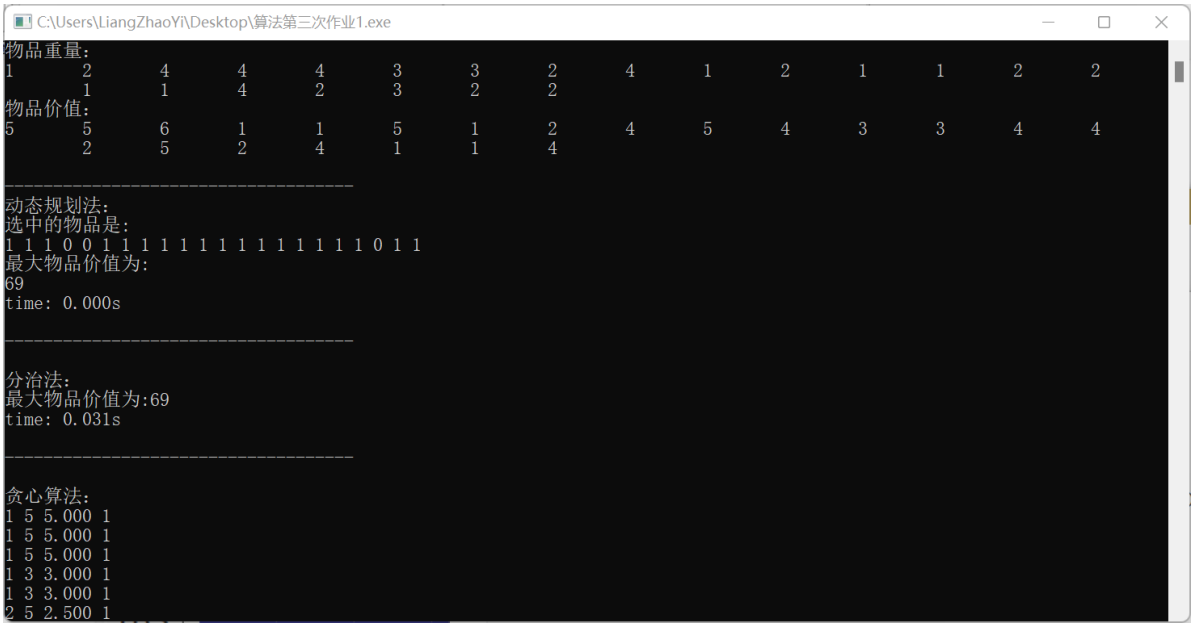
```
void Initialize(){
    int i;
    cw=0;
    cv=0;
    bestp=0;
    total=1;
    for(i=0;i<SIZE;i++)
    bestx[SIZE+1];
    for(i=0;i<SIZE;i++)
    item[i]=0;
}

int Create_Rand_Num(int temp[],int size,int randx)
{
    int i;
    for(i=0;i<size;i++)
    {
        temp[i]=rand()%randx;
        while(temp[i]==0){
            temp[i]=rand()%randx;
        }
    }
}

void Printf(){//打印函数
    int i;
    printf("物品重量：\n");
    for(i=0;i<SIZE;i++){
        printf("%d\t",w[i]);
     }
    printf("\n物品价值：\n");
    for(i=0;i<SIZE;i++){
        printf("%d\t",v[i]);
     }
    printf("\n");
}
```

一些简单的子函数，用于减少代码冗余。排序函数则是使用插入排序，根据物品的价值率重新排列物品。在贪心算法，回溯法中会用到：

```
void swap(int t[],int i,int j){//交换函数
    int temp;
    temp = t[i];
    t[i] = t[j];
```

```
    t[j] = temp;
}

void swap0(float t[],int i,int j){//交换函数
    float temp;
    temp = t[i];
    t[i] = t[j];
    t[j] = temp;
}

void strcpy0(int *a,int *b, int n){
    int i;
    b[0]=0;
    for(i=0; i<n;i++)
    b[i+1] =a[i];
    return;
}

void strcpy1(int *a,float *b, int n){
    int i;
    for(i=0; i<n;i++)
    b[i] =float(a[i]);
    return;
}

void sort1(float t[],float w[],float v[],int n){
    int max1;
    for(int i=0;i<n;i++){
        max1=i;
        for(int j=i+1;j<n;j++){
            if(t[j]>t[max1]||(w[j]<w[max1]&&t[j]==t[max1])){
                max1=j;
            }
        }
        swap0(t,i,max1);
        swap0(w,i,max1);
        swap0(v,i,max1);
    }
}
```

打印结果：

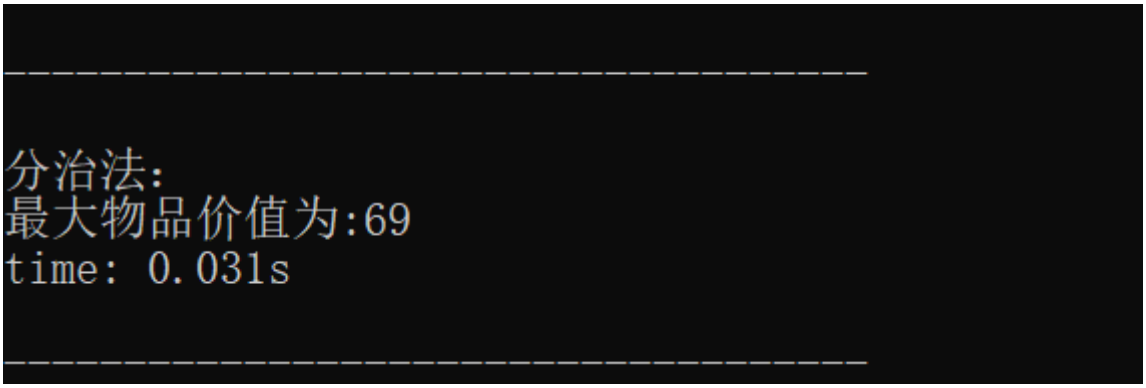


(1) 分治法

分治法，顾名思义，通过把问题一层层分解为子问题来求解。分治法通过自顶向下遍历所有解空间，因此速度较慢。在实验中我使用函数递归方法求解。i 为当前递归到的物品序号，j 为此时子问题的背包大小：

```
int Divide0(int i,int j){//分治法
    if(j <= 0) return 0;
    if(i == -1) return 0;
    if(j<w[i])
        return Divide0(i - 1, j);
    else
    return max(Divide0(i - 1, j), Divide0(i - 1, j - w[i]) + v[i]);
}

void Divide(){//分治法
    printf("\n-------------------------------------\n");
    int s;
    Initialize();
    clock_t begin, end;
    begin = clock();
    s = Divide0(SIZE-1,C);
    printf("\n分治法：\n最大物品价值为:%d\n" , s);
    end = clock();
    printf("time: %.3fs\n" , double(end - begin) / CLOCKS_PER_SEC) ;
}
```

运行结果：



对比下面动态规范算法可以看出，分治法时间复杂度较大，为O(2^n)。优点是实现代码简单，只需要简单的递归就可以实现。

(2) 动态规划

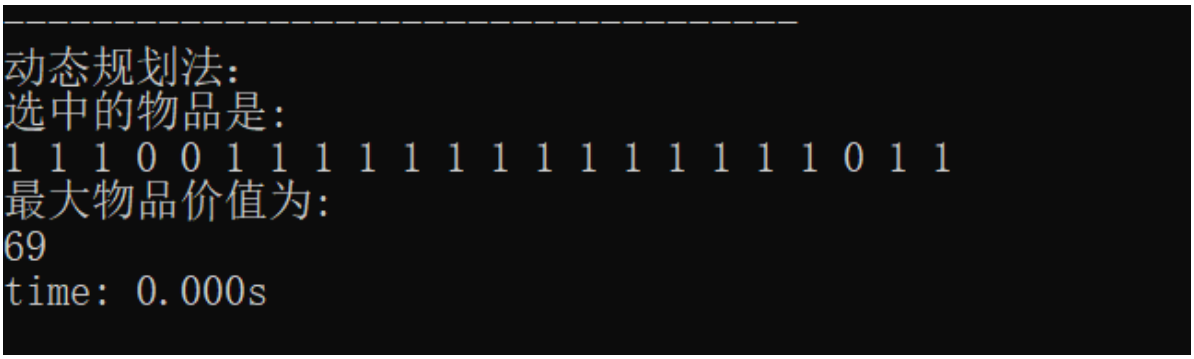
动态规划算法也是把原始问题分解为若干个子问题，然后自底向上，先求解最小的子问题，把结果存在表格中，在求解大的子问题时，直接从表格中查询小的子问题的解，避免重复计算，从而提高算法效率。01背包问题满足最优子结构，动态规划算法在该问题中表现较好。

dp二元数组存储了子问题的解，避免了重复计算。DPprint函数在dp数组生成后打印出结果：

```
int DPprint(int n){//打印选取的物品
            int i,j=C;
            for(i=n;i>=1;i--)
            {
            if(dp[i][j]>dp[i-1][j])
                {
                    item[i]=1;
                    j=j-w[i-1];
                }
            else
                item[i]=0;
            }
            printf("选中的物品是:\n");
            for(i=1;i<=n;i++)
                printf("%d ",item[i]);
            printf("\n");
}

int DP(int n,int item[])
{
    int i,j;
    for(i=0;i<=n;i++)
        dp[i][0]=0;
    for(j=0;j<=C;j++)
        dp[0][j]=0;
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=C;j++)
        {
            if(j<w[i-1])
            {
                dp[i][j]=dp[i-1][j];
            }

            else
            {
                dp[i][j]=max(dp[i-1][j],dp[i-1][j-w[i-1]]+v[i-1]);
            }
        }
    }
    DPprint(n);
    return dp[n][C];
}

void Dynamicpro(){
    printf("\n-----------------------------------\n");
    int s;
```

```
    Initialize();
    clock_t begin, end;
    begin = clock();
    printf("动态规划法：\n");
    s=DP(SIZE,item);
    printf("最大物品价值为:\n");
    printf("%d\n",s);
    end = clock();
    printf("time: %.3fs\n" , double(end - begin) / CLOCKS_PER_SEC) ;
}
```

运行结果：



可以看见，动态规划算法的时间复杂度为O(n^2)。性能相比分治法大大提高。

(3) 贪心算法

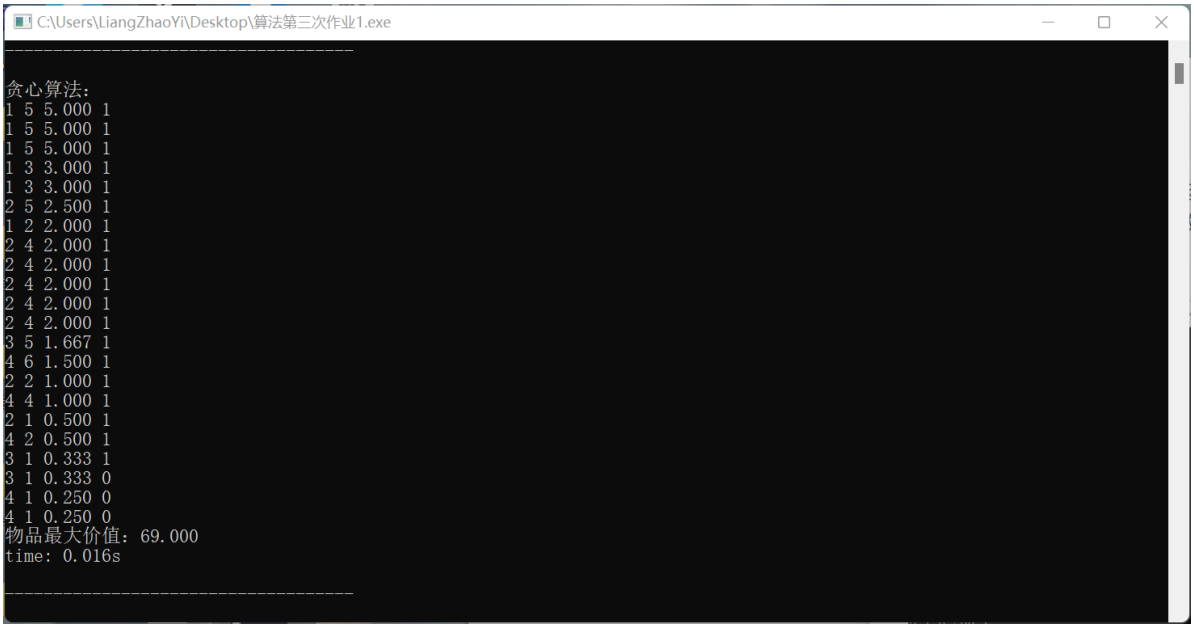
贪心算法的特点是每个阶段所作的选择都是局部最优的，通过所作的局部最优选择产生出一个全局最优解。在01背包问题中，贪心算法需要对物品的v/w进行排序，通过选择价值率最大的物品来进行贪心选择。由于只需要进行排序和一次选择，贪心算法的效率是相当快的。但是贪心算法并不能保证得出结果是最优的，因为其不能保证背包被填满，背包的空缺导致背包内的物品价值率下降。在打印结果时，由于对数组进行了重排，因此把物品的数组也相应打印出来，方便对照结果。

```
void greedy() {
    float tem[SIZE],temw[SIZE],temv[SIZE];
    int i,j;
    float c=C,maxvalue=0;
    strcpy1(w,temw,SIZE);
    strcpy1(v,temv,SIZE);
    for(i=0;i<SIZE;i++){
        tem[i] = temv[i]/temw[i];
    }
    sort1(tem,temw,temv,SIZE);
    for(j=0;j<SIZE;j++){
        if(w[j]>=c)continue;
        item[j]=1;
        c=c-temw[j];
        maxvalue=maxvalue+temv[j];
    }

    printf("\n贪心算法：");
    for(i=0;i<SIZE;i++){
        printf("\n%.0f %.0f %.3f %d",temw[i],temv[i],tem[i],item[i]);
    }
    printf("\n物品最大价值：%.3f",maxvalue);
    return ;
}
```

```
void Greedy(){
    printf("\n-------------------------------------\n");
    Initialize();
    clock_t begin, end;
    begin = clock();
    greedy();
    end = clock();
    printf("\ntime: %.3fs\n" , double(end - begin) / CLOCKS_PER_SEC) ;
}
```

由于贪心算法需要计算价值率，因此需要用到浮点型数据。在这里我选择把原数组拷贝到浮点型数组上，这样即使排序，也不会影响到原数组的值。

运行结果：



为了更直观显示选择的物品以及算法正确性，结果的四个数为物品重量，物品价值，价值率，是否选择。由于物品数量较小以及对数组进行过多操作，导致运行时间似乎没有太大优势。事实上，贪心算法也在解决物品数量较大的背包问题时，性能并不会有太大下滑。但是贪心算法并不能一定得到最优解，因此在解决01背包问题上没有优势。

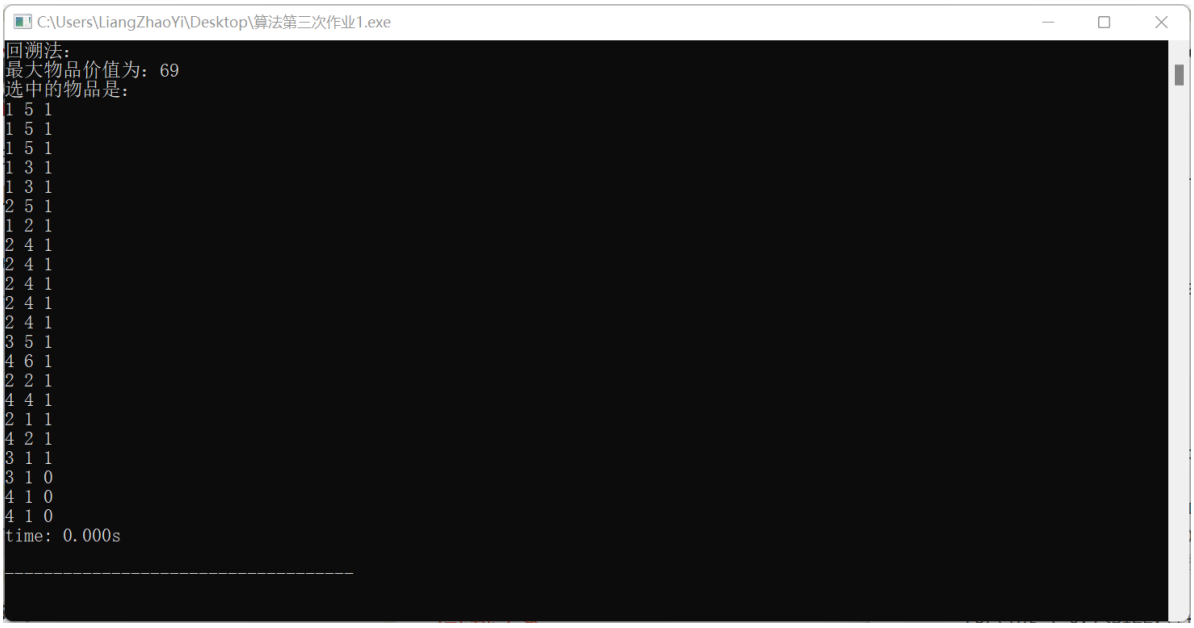
(4)回溯法

回溯法是一种深度优先搜索算法，通过回溯剪枝相对于深度优先搜索算法降低了时间复杂度。其使用剪枝，避免了遍历解空间二叉树，使得算法相较于直接搜索时间复杂度得到优化。在回溯法前，将数组预处理，按照价值率排序，可以使得剪枝更加充分，提高了算法效率：

```
double bound(float temw[],float temv[],int i)//求出上界
{
    double leftw= C-cw;
    double b = cv;
    while(i<SIZE && temw[i]<=leftw)
    {
        leftw-=temw[i];
        b+=temv[i];
        i++;
    }
    if(i<SIZE)
        b+=temv[i]/temw[i]*leftw;
    return b;
```

```
}

void backtrack(float temw[],float temv[],int i)
{
    //bound(temw,temv,i);
    if(i>=SIZE)
    {
        bestp = cv;
        return;
    }
    if(cw+temw[i]<=C)
    {
        cw+=temw[i];
        cv+=temv[i];
        item[i]=1;
        backtrack(temw,temv,i+1);
        cw-=temw[i];
        cv-=temv[i];
    }
    if(bound(temw,temv,i+1)>bestp)//判断剪枝
        backtrack(temw,temv,i+1);
}

void Backtrack(){
    printf("\n-----------------------------------\n");
    Initialize();
    clock_t begin, end;
    begin = clock();
    float tem[SIZE],temw[SIZE],temv[SIZE];
    int j;
    float c=C;
    strcpy1(w,temw,SIZE);
    strcpy1(v,temv,SIZE);
    for(j=0;j<SIZE;j++){
        tem[j] = temv[j]/temw[j];
    }
    sort1(tem,temw,temv,SIZE);
    backtrack(temw,temv,0);
    printf("\n\n回溯法：\n最大物品价值为：%d\n",bestp);
    printf("选中的物品是：");
    for(int i=0;i<SIZE;i++)
    {
        printf("\n%.0f %.0f %d",temw[i],temv[i],item[i]);
    }
    end = clock();
    printf("\ntime: %.3fs\n" , double(end - begin) / CLOCKS_PER_SEC) ;
}
```

预处理过程中，考虑到代码简单，使用了插入排序进行处理，实际上使用快速排序等高效排序算法可以进一步提高程序运行效率。上界由一般背包问题贪心算法求出，保证上界的可靠性。递归过程通过递归左右节点得出，函数实现比较简便。



回溯法理论时间复杂度为O(n2^n)。在实验中由于使用了预处理，优化了剪枝过程，因此处理较快。

(5)分支限界法

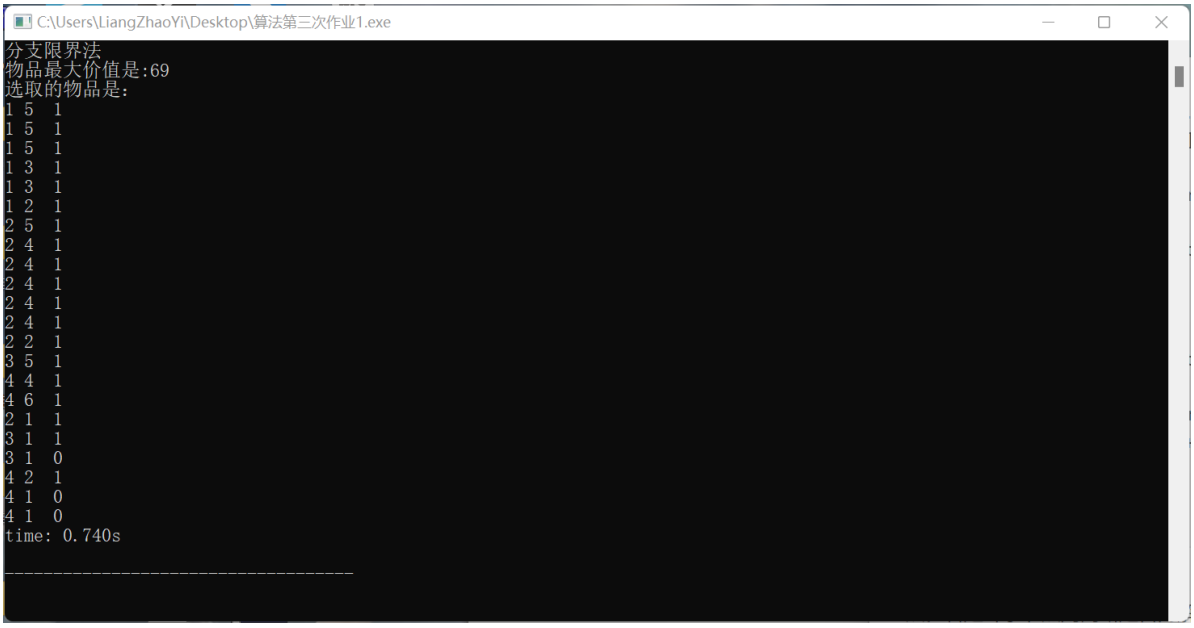
分支限界法与回溯法的最大区别是使用了广度优先的搜索算法。由于二叉树结构的特殊性，需要借助队列来实现广度优先搜索。在解决01背包问题时，我使用该两种方法的代码结构是相似的，但由于需要用到队列，代码冗余较大，因此时间复杂度有所增加。

```
void bound1(nodetype &e,int temw[],int temv[])
{
    double c=C;
    int i=e.i+1;
    int sumw=e.w;
    double sumv=e.v;
    while((sumw+temw[i]<=c)&&i<=SIZE)
    {
        sumw+=temw[i];
        sumv+=temv[i];
        i++;
    }
    if(i<=SIZE)
    e.ub=sumv+(c-sumw)*temv[i]/temw[i];
    else e.ub=sumv;
}

void enter(nodetype e,queue<nodetype> &qu,int temw[],int temv[])
{
    if(e.i==SIZE)
    {
        if(e.v>bestp)
        {
            bestp=e.v;
            for(int j=1;j<=SIZE;j++)
            item[j]=e.x[j];
        }
    }
    else qu.push(e);
}

void bfs(int temw[],int temv[])
```

```
{
    int j;
    nodetype e,e1,e2;
    queue<nodetype> qu;

    e.i=0;
    e.w=0;
    e.v=0;
    e.no=total++;

    for(j=1;j<=SIZE;j++)
    e.x[j]=0;
    bound1(e,temw,temv);
    qu.push(e);

    while(!qu.empty())
    {
        e=qu.front();qu.pop();
        if(e.w+temw[e.i+1]<=C)
        {
            e1.no=total++;
            e1.i=e.i+1;
            e1.w=e.w+temw[e1.i];
            e1.v=e.v+temv[e1.i];
            for(j=1;j<=SIZE;j++)
            e1.x[j]=e.x[j];
            e1.x[e1.i]=1;
            bound1(e1,temw,temv);
            enter(e1,qu,temw,temv);
        }
        e2.no=total++;
        e2.i=e.i+1;
        e2.w=e.w;
        e2.v=e.v;
        for(j=1;j<=SIZE;j++)
            e2.x[j]=e.x[j];
        e2.x[e2.i]=0;
        bound1(e2,temw,temv);
        if(e2.ub>bestp)
        enter(e2,qu,temw,temv);
    }
}

void output(int temw[],int temv[])
{
    printf("\n\n分支限界法\n物品最大价值是:%d\n选取的物品是: ",bestp);
    for(int i=1;i<=SIZE;i++)
        printf("\n%d %.d  %d",temw[i],temv[i],item[i]);

}
```

其中，bound1函数为求上界函数，与回溯法的bound一致。enter为进队列函数，需要判断是否为叶子节点。若是叶子节点则判断是否为解。bfs则是通过队列来遍历解二叉树，从而实现广度优先搜索。

运行结果：



理论上，分支限界法的时间复杂度小于回溯法，为O(2^n)。但由于在实际程序中，需要用到队列，代码冗余量较大，因此在测试中，反而慢于回溯法。

(6)蒙特卡洛法

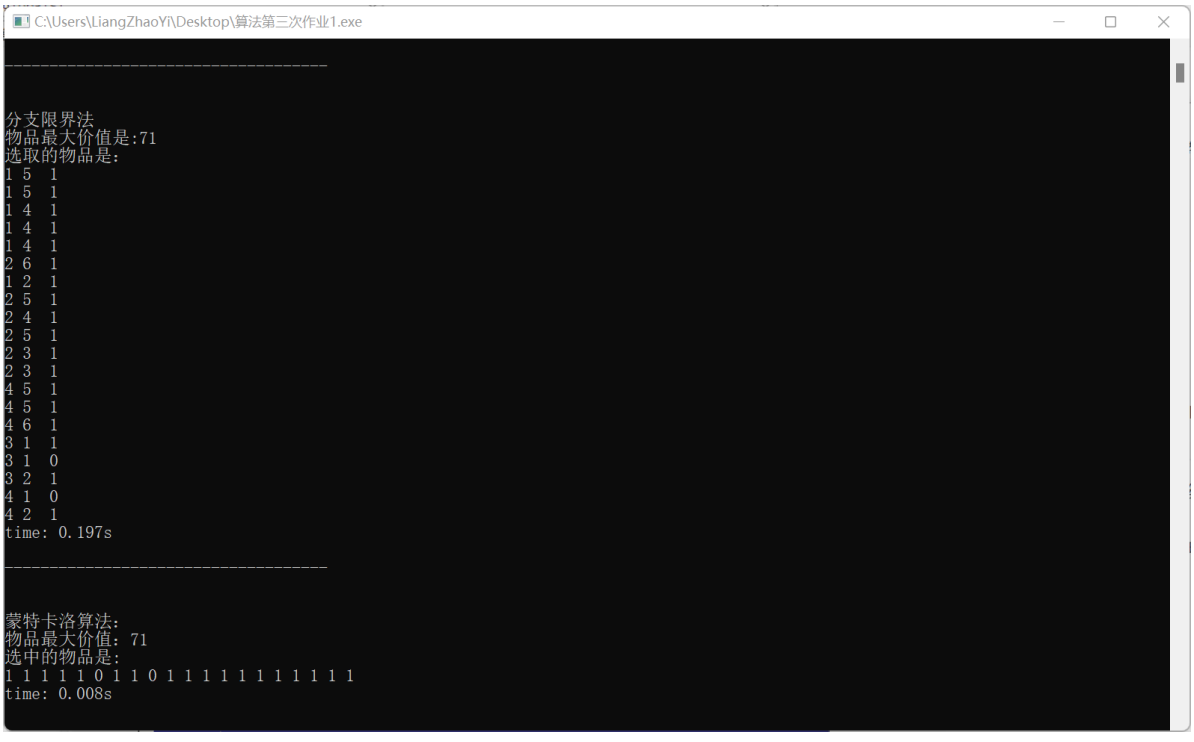
蒙特卡洛算法基本思想是基于随机事件出现的概率的算法，常常用于计算圆周率以及积分，用概率模型来解决问题。在实验中，使用随机数组模拟每一次抽取，每个物品都可能被抽到背包中，通过多次重复保证结果的完整性。

```
int MonteCarlo0(){
    bestp = 0;
    long int i=0;
    int key,j,nowvaule=0;
    //double prob = double(MAX_VAULE/MAX_WEIGHT*1/5);
    int item0[SIZE];
    float tem[SIZE];
    //printf ("%lf",prob);
    for(j=0;j<SIZE;j++){
        tem[j] =float(v[j]/w[j]);
    }
    for(int i=0;i<=10000;i++){
        int nowvaule = 0;
        int leftc = C;
        memset(item0,0,sizeof(item0));
        while(1){
            key = rand()%SIZE;
            if( item0[key]!=1 ){
            //  if( tem[key]>float(rand()%10/10*prob)) {
                if( w[key] <= leftc){
                    nowvaule = nowvaule + v[key];
                    leftc = leftc - w[key];
                    item0[key] = 1;
                }
                else if(leftc <=2) break;
            }
        }
        //}
        if(nowvaule>bestp){
            bestp = nowvaule;
            for(i=0;i<SIZE;i++)
```

```
                item[i] = item0[i];
            }
        //  printf("%d ",nowvaule);
        }
        printf("\n\n蒙特卡洛算法：\n物品最大价值：%d\n",bestp);
        printf("选中的物品是:\n");
        for(i=0;i<SIZE;i++)
            printf("%d ",item[i]);
        printf("\n");
        return 0;
    }

    void MonteCarlo(){
        printf("\n-------------------------------------\n");
        Initialize();
        clock_t begin, end;
        begin = clock();
        MonteCarlo0();
        end = clock();
        printf("time: %.3fs\n" , double(end - begin) / CLOCKS_PER_SEC) ;
    }
```

由于不能保证每次背包都能填满，因此在结束循环时设置了背包的冗余，避免卡死。又因为防止循环不充分，使用while循环的形式而不是for循环。通过一次次模拟，最后得出结果。在实验时设想每个物品根据价值率，以不同概率放入背包，以减少循环次数以及增加正确率。

实验结果：



蒙特卡洛法不能保证一定得出正确答案，在背包物品增加时，需要增加循环次数来提高得出正确解的概率，或者增加优化条件。

实验总结

第三次实验相比前面两次次实验，代码量较小。不过需要彻底消化六个算法，并且编写出相应程序，也要花不少功夫。回溯法和分支限界法涉及到树和队列的操作，虽然最后实现代码并不复杂，但要花费相当长的时间去理解以及实现。在实现分治法以及分支限界法时，我一度由于时间过长而怀疑算法出现了死循环，在经历了反复的debug后才发现，只是算法比较慢以及代码冗余量较大，在物品数大于22后，需要的时间将会大幅增加。