子集和问题的fft实现

摘要

本实验选择阅读了提供的第四篇文献（Subset Sum Made Simple），利用文献所给方法利用一维，二维快速傅里叶变换编码实现了给定一个集合和一个正整数，求解这个集合的所有小于这个给定的正整数子集之和。对于一个具有n个正整数的集合和一个目标正整数u，这个算法具有$O(ulogu\sqrt{nlogn})$的时间复杂度，虽然这个算法不是最快的，但是它易于理解可以作为算法课上讲解fft运用的例子.

这个算法的主要思路是分治和fft快速求解多项式乘法还有利用同余的划分性质加快运算.最后还利用计时工具对复杂度进行了实践上的检验.

实验目的

看懂理解英文文献,并最终动手实现这个算法,并进行复杂度分析.

论文主要内容及复杂度分析

引入一些符号

1.$[u]=\{0,1,2,3,\ldots,u\}$

2.$S_u(X)=\{\sum Y|Y\subseteq X\}\cap[u]$

例如：$X=\{1,2,3\},u=5,S_u(X)$意思就是X的不超过u的子集和的集合,在这里$S_u(X)=\{0,1,2,3,4,5\}$（0是空集）

3.$S_u^{\#}(X)=\{(\sum Y,|Y|)|Y\subseteq X\}\cap[u\times\mathbb{N}]$

例如$X=\{1,2,3\},u=5,S_u^{\#}(X)$意思就是$X$的不超过u的子集和和这个子集对应的元素个数的集合在这里$S_u(X)=\{(1,1),(2,1),(3,1),(3,2)(4,2),(5,2)\}$

如果$X,Y\subset\mathbb{N}$

4.$X\oplus_u Y=\{x+y|x\in X,y\in Y\}\cap[u]$

如果$X,Y\subset\mathbb{N}\times\mathbb{N}$

5.$X\oplus_u Y=\{(x_1+y_1,x_2+y_2)|(x_1,x_2)\in X,(y_1,y_2)\in Y\}\cap[u\times\mathbb{N}]$

两个函数

> $ALLSUBSETSUMS$
>
> 输入：一个集合S和正整数u
>
> 输出：S所有不大于u的子集和

> $ALLSUBSETSUMS^{\#}$
>
> 输入：一个集合S和正整数u
>
> 输出：S所有不大于u的子集和以及它对应的子集的元素个数

所以$ALLSUBSETSUMS$就是求出$S_u(X)$

定理0

如果X,Y是两个不相交的集合那么$S_u(X\cup Y)=S_u(X)\oplus_u S_u(Y)S_u^{\#}(X\cup Y)=S_u^{\#}(X)\oplus_u S_u^{\#}(Y)$

这个定理说明$ALLSUBSETSUMS$和$ALLSUBSETSUMS^{\#}$是可以划分为子问题，也就为后面的分治提供了基础

定理1

（A）给定两个集合$S,T\subseteq[u]$,可以在$O(ulogu)$的时间复杂度计算$S\oplus_u T$

（B）给定k个集合$S_1,\ldots,S_k\subseteq[u]$,可以在$O(kulogu)$的时间复杂度计算$S_1\oplus_u\cdots\oplus_u S_k$

（C）给定两个二维的点集合$S,T\subseteq[u]\times[v]$,可以在$O(uvlog(uv))$的时间复杂度计算$S\oplus_u T$

证明：（A）令$f_s=\sum_{i\in S}x^i,f_t$同理，那么计算$S\oplus_u T$其实相当于求f_s*f_t,运用fft实现快速多项式乘法即可，这个时间复杂度就是$O(ulogu)$

（B）相当于上述运算进行k次，于是复杂度为$O(kulogu)$

（C）令$f_s=\sum_{(i,j)\in S}x^iy^j,f_t$同理，$S\oplus_u T$就相当于进行多元多项式乘法，利用二维$fft$即可，复杂度是$O(uvlog(uv))$

定理2

$ALLSUBSETSUMS^{\#}$可以在$O(ulogu\,nlogn)$时间复杂度解决

证明：对于给定的集合S,有n个数，和限界u。将它分为两个数量相等的子集合S_1,S_2,假如$S_u^{\#}(S_1),S_u^{\#}(S_2)$已经求出来了。则由定理0可知$S_u^{\#}(S)=S_u^{\#}(S_1)\oplus_u S_u^{\#}(S_2)$,而由定理1，可以在$O(unlogu)$的时间复杂度计算$S\oplus_u T$，所以有$T(n)=2T(\frac{n}{2})+O(unlogu)$,解递归即得$T(n)=O(ulogu\,nlogn)$

由定理2给出$ALLSUBSETSUMS^{\#}$的伪代码

> $ALLSUBSETSUMS^{\#}$
>
> 输入：一个集合S和正整数u
>
> 输出：S所有不大于u的子集和以及它对应的子集的元素个数

> **1.if** S={X}
>
> **2.** **return**{(0,0),(x,1)}
>
> **3.**T⟵S的一半
>
> **4.return**$ALLSUBSETSUMS^{\#}(T,u)\oplus_u ALLSUBSETSUMS^{\#}(S/T,u)$

定理3

给定$l,b\in\mathbb{N},l<b$,给定一个集合$S\subseteq\{x\in\mathbb{N}|x\equiv l\pmod b\}$，那么可以在$O((u/b)nlognlogu)$的复杂度计算$S_u(S)$

证明：对于每个$x\in S,x=yb+l$,令$Q=\{y|yb+l\in S\}$,对于任意一个有j个数的子集$X=\{y_1b+l,\dots,y_j+b\}\subseteq S$，$\sum_{x\in X}x=\sum_{i=1}^{j}(y_ib+l)=(\sum_{i=1}^{j}y_i)b+jl$,令$(z,j)\in S_{u/b}^{\#}(Q)$,令$Y=\{y_1,\dots,y_j\}\subseteq Q$,则$z=\sum_{i=1}^{j}y_i$.则对于任一对点$(z,j)\in S_{u/b}^{\#}(Q)$,都对应一个数$zb+jl\in S_u(S)$,那么计算$S_u(S)$，就相当于计算$S_{u/b}^{\#}(Q)$,于是复杂度为$O((u/b)nlognlogu)$

$ALLSUBSETSUMS$**的伪代码**

> $ALLSUBSETSUMS$
>
> 输入：一个集合S和正整数u
>
> 输出：S所有不大于u的子集和
>
> **1.**$b\leftarrow\lfloor\sqrt{nlogn}\rfloor$
>
> **2.for** $l\in[b-1]$ **do**
>
> **3.** $S_l\leftarrow S\cap\{x\in\mathbb{N}|x\equiv l\pmod b\}$
>
> **4.** $Q_l\leftarrow\{(x-l)/b|x\in S_l\}$
>
> **5.** $S_{u/b}^{\#}(Q_l)\leftarrow ALLSUBSETSUMS^{\#}(Q_l,\lfloor u/b\rfloor)$
>
> **6.** $R_l\leftarrow\{zb+jl|(z,j)\in S_{u/b}^{\#}(Q_l)\}$
>
> **7.return** $R_0\oplus_u\cdots\oplus_u R_{b-1}$

$ALLSUBSETSUMS$**的时间复杂度为**$O(ulogu\sqrt{nlogn})$

证明：上述伪代码先把S分为$b=\lfloor\sqrt{nlogn}\rfloor$份,对于每一个$S_l$,都可以用定理3那种方法去计算。计算所有$S_l$的复杂度为$\sum_{l\in[b-1]}O((u/b)n_llogn_llogu)=O((u/b)nlognlogu)$.最后再合并起来,计算$R_0\oplus_u\cdots\oplus_u R_{b-1}$的复杂度由定理1为$O(bulogu)$。因此，总的复杂度为$O((u/\lfloor\sqrt{nlogn}\rfloor)nlognlogu+\lfloor\sqrt{nlogn}\rfloor ulogu)=O(ulogu\sqrt{nlogn})$.

实验设计流程

在读完论文后,可以知道一下几点是比较关键的

1.本实验中有两种集合，一个是一维的集合，另一个是二维的点的集合。要用一种恰当的数据结构去实现集合

在这里对于一维的集合，就用c++的$vector<int>$实现,因为$vector<int>$是变长数组，适用于这里的情况。

对于二维的集合，使用$vector<vector<int>>$这样一个二维的数组去实现，其中这个数组列就表示一个点，故列数就是集合描述的点的数量。而点的坐标（x,y）的值由行去描述。所以行数固定为2。其中第0行对应x的值，第1行对应y的值，例如：

$S[0][0]=1,S[0][1]=5,S[0][2]=3$

$S[1][0]=2,S[1][1]=9,S[1][2]=6$

那么集合S描述的是$(1,2),(5,9),(3,6)$这三个点

2.实现$\oplus_u$对应于上面两种集合的情况，最核心的就是用一维，二维fft快速多项式乘法，和快速多元多项式乘法

首先要把上面的集合转换成特征多项式,也就是定理1提到的$f_s=\sum_{i\in S}x^i$，$f_s=\sum_{(i,j)\in S}x^iy^j$

转换后，才可以进行快速多项式乘法。对于一维多项式，先利用fft把上述多项式转换成点值表示$O(ulogu)$，再$O(u)$的时间对两个多项式的点值表示进行简单的相乘，再来fft逆变换把点值表示转换成特征多项式的形式，最后再将特征多项式转换成最开始的集合的形式。复杂度为$O(ulogu)$

二维的多元多项式也是同理的。

在本次实验，我调用了fftw库（~~虽然fft我是会的~~，但是二维fft就不会了，所以干脆全部使用fftw库）来完成fft和ifft。fftw库是全世界最快的快速傅里叶变换，可自动适应机器的配置例如， 缓存，内存大小，寄存器个数等，并进行最优的设置。

代码实现（需安装fftw库才能运行）（代码做过改进与ppt上的不同，但大意相同）

```
#include <iostream>
#include "fftw3.h"
#pragma comment(lib, "libfftw3-3.lib")
#include <vector>
#include<algorithm>
using namespace std;
#include <cmath>
#include<Windows.h>
fftw_complex* in_2d_fft;
fftw_complex* in_2d_ifft;
fftw_complex* out_2d_afft;
fftw_complex* out_2d_bfft;
fftw_complex* out_2d_ifft;

fftw_plan p_forward_2d_a, p_forward_2d_b, p_backward_2d;

fftw_complex* in_1d_fft, * in_1d_ifft;
```

```
fftw_complex* out_1d_afft;
fftw_complex* out_1d_bfft;
fftw_complex* out_1d_ifft;

fftw_plan p_forward_1d_a, p_forward_1d_b, p_backward_1d;

vector<int> ploy1d_fft_mult(vector<int>& acoef, vector<int>& bcoef, int u, int N)//a,b输入，c输出,u为限制大小，
a_max_coef, b_max_coef表示a,b的最高次数(是真实的最高次数，所以要加1)
{
    vector<int> ccoef;
    //先根据系数构造系数表示法的数组;
    for (int i = 0; i < acoef.size(); i++)
    {
        if (acoef[i] <= u)in_1d_fft[acoef[i]][0] = 1;
    }
    fftw_execute(p_forward_1d_a);

    for (int i = 0; i < acoef.size(); i++)
    {
        if (acoef[i] <= u)in_1d_fft[acoef[i]][0] = 0;
    }
    for (int i = 0; i < bcoef.size(); i++)
    {
        if (bcoef[i] <= u)in_1d_fft[bcoef[i]][0] = 1;
    }
    fftw_execute(p_forward_1d_b);

    for (int i = 0; i < bcoef.size(); i++)
    {
        if (bcoef[i] <= u)in_1d_fft[bcoef[i]][0] = 0;
    }


    //已经有a, b的点值表示了接下来O（n）的时间算乘法
    for (int i = 0; i < N; i++) {  //(a+bi)(c+di) = (ac - bd) + (bc + ad)i
        in_1d_ifft[i][0] = out_1d_afft[i][0] * out_1d_bfft[i][0] - out_1d_afft[i][1] * out_1d_bfft[i][1];
        in_1d_ifft[i][1] = out_1d_afft[i][1] * out_1d_bfft[i][0] + out_1d_afft[i][0] * out_1d_bfft[i][1];
    }
    fftw_execute(p_backward_1d);
    //std::cout << "a * b poly \n";
    for (int i = 0; i < N && i <= u; i++) {
        //cout << out[i][0] / N << "\n";
        if (out_1d_ifft[i][0] / N >= 0.8) {
            ccoef.push_back(i);
        }
    }//out[i][0]表示了x^i的系数

    return ccoef;
}

void ploy2d_fft_mult(vector<vector<int> >& acoef, vector<vector<int> >& bcoef, vector<vector<int> >& ccoef,
int u, int row, int col)
{
    for (int i = 0; i < acoef[0].size(); i++)
    {
        if (acoef[0][i] <= u) {
            in_2d_fft[acoef[0][i] + col * acoef[1][i]][0] = 1;
        }
    }
    fftw_execute(p_forward_2d_a);
    for (int i = 0; i < acoef[0].size(); i++)
    {
        if (acoef[0][i] <= u) {
            in_2d_fft[acoef[0][i] + col * acoef[1][i]][0] = 0;
        }
    }

    //b,再对b进行二维傅里叶变换
    for (int i = 0; i < bcoef[0].size(); i++)
    {
        if (bcoef[0][i] <= u) {
            in_2d_fft[bcoef[0][i] + col * bcoef[1][i]][0] = 1;
        }
    }

    fftw_execute(p_forward_2d_b);

    for (int i = 0; i < bcoef[0].size(); i++)
    {
        if (bcoef[0][i] <= u) {
            in_2d_fft[bcoef[0][i] + col * bcoef[1][i]][0] = 0;
        }
    }

    vector<vector<vector<double> >> bfft(row, vector<vector<double> >(col, vector<double>(2)));//三维数组，row
行，col列，每个还有0，1两个范围，0表示实数，1表示虚数
```

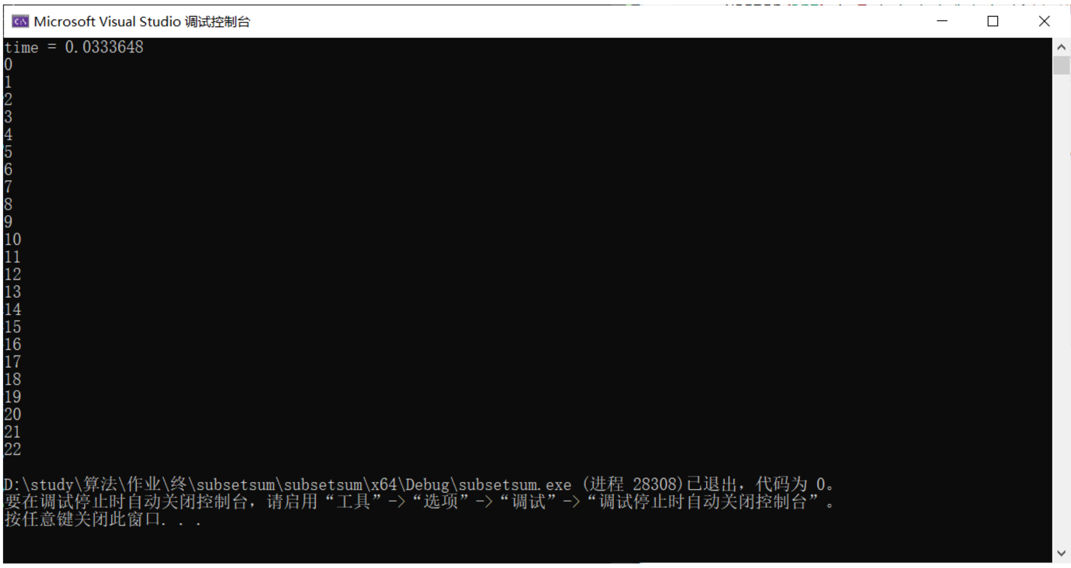
```
    // 计算结果存储到afft中


    for (int i = 0; i < row; ++i)
    {
        for (int j = 0; j < col; ++j)
        {
            in_2d_ifft[j + col * i][0] = out_2d_afft[j + col * i][0] * out_2d_bfft[j + col * i][0] -
out_2d_afft[j + col * i][1] * out_2d_bfft[j + col * i][1];//(a+bi)(c+di)=ac-bd+i(ad+bc)
            in_2d_ifft[j + col * i][1] = out_2d_afft[j + col * i][0] * out_2d_bfft[j + col * i][1] +
out_2d_afft[j + col * i][1] * out_2d_bfft[j + col * i][0];
        }
    }

    // ifft

    fftw_execute(p_backward_2d);
    for (int i = 0; i < row; ++i)
    {
        for (int j = 0; j < col && j <= u; ++j)
        {
            if (out_2d_ifft[j + col * i][0] / (row * col) >= 0.8) {//防止浮点数的上下浮动，所以不能取1
                ccoef[0].push_back(j);
                ccoef[1].push_back(i);
            }
        }
    }




}

vector<vector<int> > all_subset_sums_withinfo(vector<int > S, int u, int row, int col)
{
    if (S.size() == 1)//表示集合仅有一个元素
    {
        vector<vector<int> > re(2, vector<int>(2));
        re[0][0] = re[1][0] = 0;
        re[0][1] = S[0];
        re[1][1] = 1;
        return re;
    }
    else if (S.size() == 0) {//表示为空集
        vector<vector<int> > re(2, vector<int>(1));
        re[0][0] = re[1][0] = 0;
        return re;
    }
    else {
        int size = S.size();
        vector<int> T(S.begin(), S.begin() + size / 2);
        vector<int> S_div_T(S.begin() + size / 2, S.end());
        vector<vector<int> > T_tem = all_subset_sums_withinfo(T, u, row, col);
        vector<vector<int> > S_div_T_tem = all_subset_sums_withinfo(S_div_T, u, row, col);
        vector<vector<int> > re(2);
        ploy2d_fft_mult(T_tem, S_div_T_tem, re, u, row, col);
        return re;
    }
}

vector<int > all_subset_sums(vector<int > S, int u)
{
    int n = S.size();
    int b = sqrt(log(n) * n);
    if (b == 0) {
        vector<int > re = { 0 };
        re = ploy1d_fft_mult(re, S, u, 0);
        return re;
    }
    int l = 0;
    vector<vector<int> > Q_l(b);
    vector<vector<int> > R_l(b);
    for (int i = 0; i < n; i++)
    {
        l = S[i] % b;//l表示除b的余数
        Q_l[l].push_back(S[i] / b);//在第l行加入与b的除数
    }//构造Q_l
    int row = n + 1, col = (u / b) * 2 + 1;

    in_2d_fft = (fftw_complex*)fftw_malloc(row * col * sizeof(fftw_complex));
    in_2d_ifft = (fftw_complex*)fftw_malloc(row * col * sizeof(fftw_complex));
    out_2d_afft = (fftw_complex*)fftw_malloc(row * col * sizeof(fftw_complex));
    out_2d_bfft = (fftw_complex*)fftw_malloc(row * col * sizeof(fftw_complex));
    out_2d_ifft = (fftw_complex*)fftw_malloc(row * col * sizeof(fftw_complex));

    p_forward_2d_a = fftw_plan_dft_2d(row, col, in_2d_fft, out_2d_afft, FFTW_FORWARD, FFTW_MEASURE);
    p_forward_2d_b = fftw_plan_dft_2d(row, col, in_2d_fft, out_2d_bfft, FFTW_FORWARD, FFTW_MEASURE);
```

```
    p_backward_2d = fftw_plan_dft_2d(row, col, in_2d_ifft, out_2d_ifft, FFTW_BACKWARD, FFTW_MEASURE);

    for (int i = 0; i < row; ++i)
    {
        for (int j = 0; j < col; ++j)
        {
            in_2d_fft[j + col * i][0] = 0;
            in_2d_ifft[j + col * i][0] = 0;
            in_2d_fft[j + col * i][1] = 0;
            in_2d_ifft[j + col * i][1] = 0;
        }
    }



    for (l = 0; l < b; l++)
    {
        vector<vector<int> > S_Q_l = all_subset_sums_withinfo(Q_l[l], u / b, row, col);
        int x;
        for (int i = 0; i < S_Q_l[0].size(); i++)
        {
            x = S_Q_l[0][i] * b + S_Q_l[1][i] * l;
            R_l[l].push_back(x);
        }
    }//构造R_l
    vector<int > re = R_l[0];
    int N = 2 * u + 1;
    in_1d_fft = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * N);
    in_1d_ifft = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * N);
    out_1d_afft = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * N);
    out_1d_bfft = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * N);
    out_1d_ifft = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * N);
    for (int i = 0; i < N; i++) {
        in_1d_fft[i][0] = 0;
        in_1d_fft[i][1] = 0;
        in_1d_ifft[i][0] = 0;
        in_1d_ifft[i][1] = 0;
    }
    p_forward_1d_a = fftw_plan_dft_1d(N, in_1d_fft, out_1d_afft, FFTW_FORWARD, FFTW_MEASURE);
    p_forward_1d_b = fftw_plan_dft_1d(N, in_1d_fft, out_1d_bfft, FFTW_FORWARD, FFTW_MEASURE);
    p_backward_1d = fftw_plan_dft_1d(N, in_1d_ifft, out_1d_ifft, FFTW_BACKWARD, FFTW_MEASURE);
    for (l = 0; l < b - 1; l++)
    {
        re = ploy1d_fft_mult(re, R_l[l + 1], u, N);
    }
    fftw_destroy_plan(p_forward_2d_a);
    fftw_destroy_plan(p_forward_2d_b);
    fftw_destroy_plan(p_backward_2d);
    fftw_destroy_plan(p_forward_1d_a);
    fftw_destroy_plan(p_forward_1d_b);
    fftw_destroy_plan(p_backward_1d);
    return re;
}

int main() {

    LARGE_INTEGER t1, t2, tc;
    QueryPerformanceFrequency(&tc);
    QueryPerformanceCounter(&t1);
    vector<int> S = { 518533,1037066,2074132,1648264,796528,1593056,
686112,1372224,244448,488896,977792,1955584,1411168,322336,644672,1289344,78688,157376,314752,629504,1259008
};
    vector<int > re = all_subset_sums(S, 2000);//行取值为0，1，0表示x的次方，1表示y的次方。列取值为点的个数,在这里x是子集
的和，y是对应的子集的元素个数
    QueryPerformanceCounter(&t2);
    double time = (double)(t2.QuadPart - t1.QuadPart) / (double)tc.QuadPart;
    cout << "time = " << time << endl;  //输出时间（单位：s）
    for (int i = 0; i < re.size(); i++)
    {
        cout << re[i] << endl;
    }
    //text();


}
```

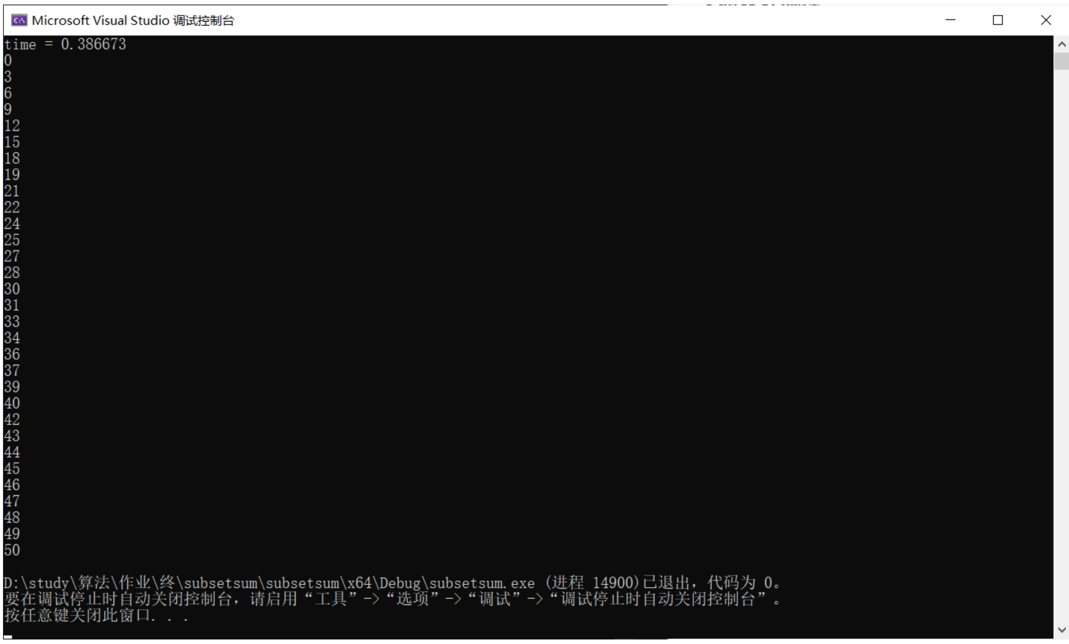
实验结果

对于附件给出的测试样例，程序都给出了正确结果

例如$S=1,2,4,8,16,32,u=22$



又如$S=\{25,27,3,12,6,15,9,30,21,19\},u=50$



而对于
$S=\{518533,1037066,2074132,1648264,796528,1593056,686112,1372224,244448,488896,977792,1955584,1411168,322336,644672,1289$
这样的测试样例，依次改变u有这样的结果

u	时间（s为单位）
2	0.0038006
6	0.0041643
12	0.0112849
20	0.0655983
200	0.362134
2000	2.20246
20000	2.09445
200000	24.719
2000000	211.513

可以看到当n不变随着u的变大，算法的增长规模确实是$O(ulogu)$的，但是只在u<20时成立.甚至在u>20时是接近$O(u)$，猜测可能是因为，当u>20时输入的数据量n=21远远少于实际的u对应的该有的数据量（考虑二维fft，u=2000时，这时fft的处理的特征多项式列数为573，行数有11行，意味着可以有6303个数据，但是实际上只有21个，所以有大量的0），于是fft能进行的比较快的进行。

总结与反思

这个算法是比较粗糙的，因为这个算法只能给出小于u的子集和，就只能给出和，而不能给出这个子集和对应的子集的元素。而且虽然复杂度是$O(ulogu\sqrt{nlogn})$，但这并不意味着它能比指数级别的搜索有效，特别是对应于，u特别大，而问题规模却只取决于问题规模与u无关，所以能更加快。但是对于普遍的求最大子集和的问题，都是u要大于问题规模的数据，所以这个算法实际上并不占优势，但是对于问题规模与u比较靠近甚至是大于的数据，那么这个算法就可以很快的处理。