

浙江大学



实验报告

作者姓名 吕珂杰、李梦圆、张雨欣

王竞康、丁晨炜

指导教师 王梁昊

学科(专业) 信息工程

所在学院 信息与工程学院

提交日期 2016 年 12 月 12 日

算法大作业实验报告

信息工程 吕珂杰、李梦圆、张雨欣、王竞康、丁晨炜

0 开发软件说明

Dev-C++是一个 Windows 下的 C 和 C++程序的集成开发环境，是一款自由软件，遵守 GPL 许可协议分发源代码。它使用 MingW32/GCC 编译器，遵循 C/C++ 11 标准，同时兼容 C++98 标准。开发环境包括多页面窗口、工程编辑器以及调试器等，在工程编辑器中集合了编辑器、编译器、连接程序和执行程序，提供高亮度语法显示，以减少编辑错误。多国语言版中包含简繁体中文语言界面及技巧提示，还有英语、俄语、法语、德语、意大利语等二十多个国家和地区语言提供选择。

1 数学分析与处理

为了完成既定的目标，我们首先做简要的数学处理。

首先计算搭建不同种类的地基长度为 i 的金字塔所需要的立方体数。

第一类是 High 金字塔，搭建地基为 i 的 High 金字塔所需要的立方体数为：

$$n = \sum_{j=1}^i j^2 = \frac{1}{6}i(i+1)(2i+1)$$

第二类是 Low 金字塔中地基长度 i 为奇数的金字塔，搭建高度 $h = \frac{i+1}{2}$ ，此类金字塔所需要的立方体数为：

$$n = \sum_{j=1}^h (2j-1)^2$$

第三类为 Low 金字塔中地基长度 i 为偶数的金字塔，搭建高度 $h = \frac{i}{2}$ ，此类金字塔所需要的立方体数为：

$$n = \sum_{j=1}^h (2j)^2 = \sum_{j=1}^h 4j^2$$

经过数学推导可以得到，第二类和第三类的金字塔数目可以统一为一个表达式，即：搭建地基为 i 的 Low 金字塔所需要的立方体数为：

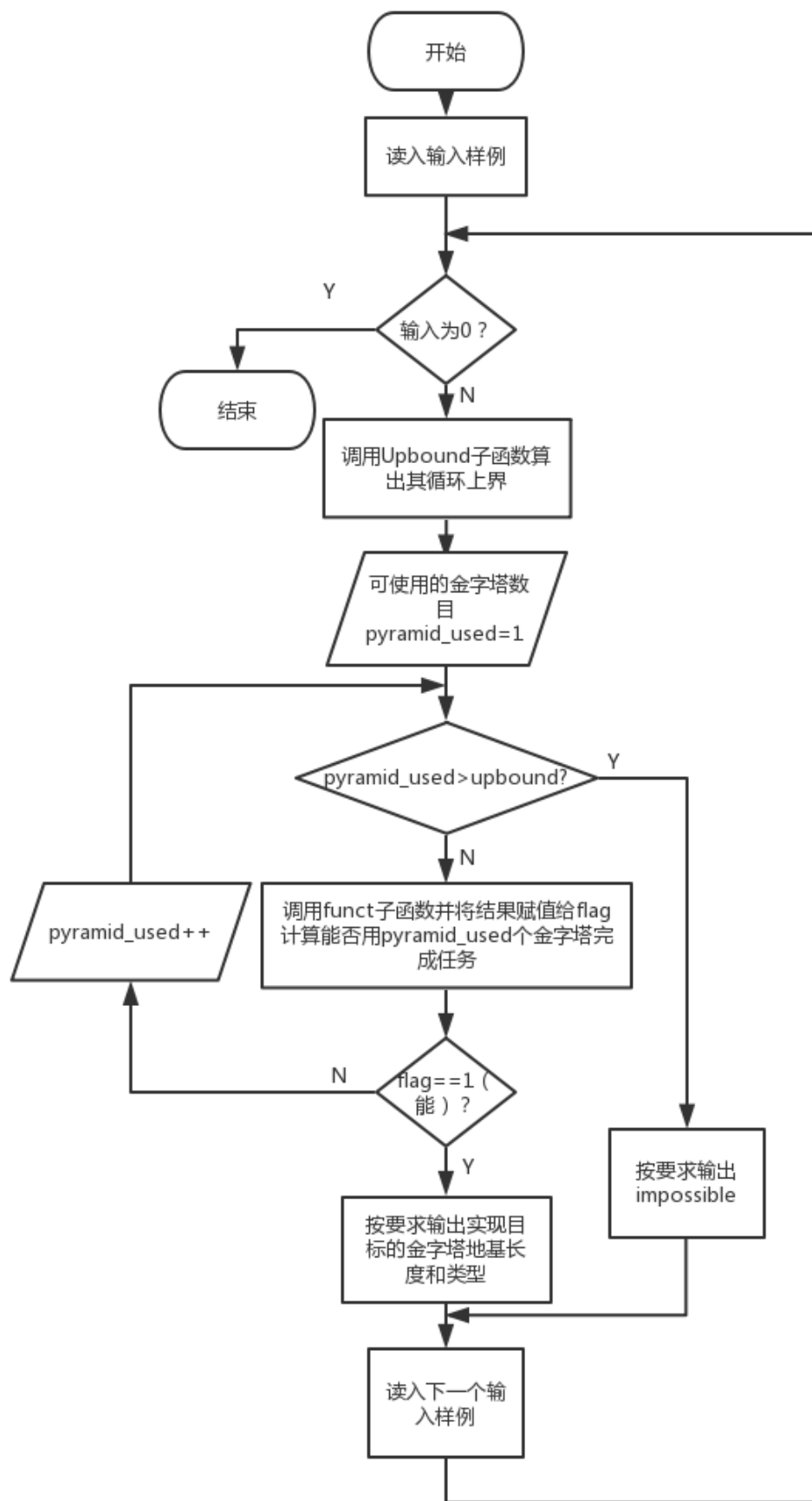
$$n = \frac{1}{6}i(i+1)(i+2)$$

接下来，可以根据上述结果，计算出立方体个数不少于 10^6 的 High 型金字塔最小地基长度为 144，立方体个数不少于 10^6 的 Low 型金字塔最小地基长度为 181。

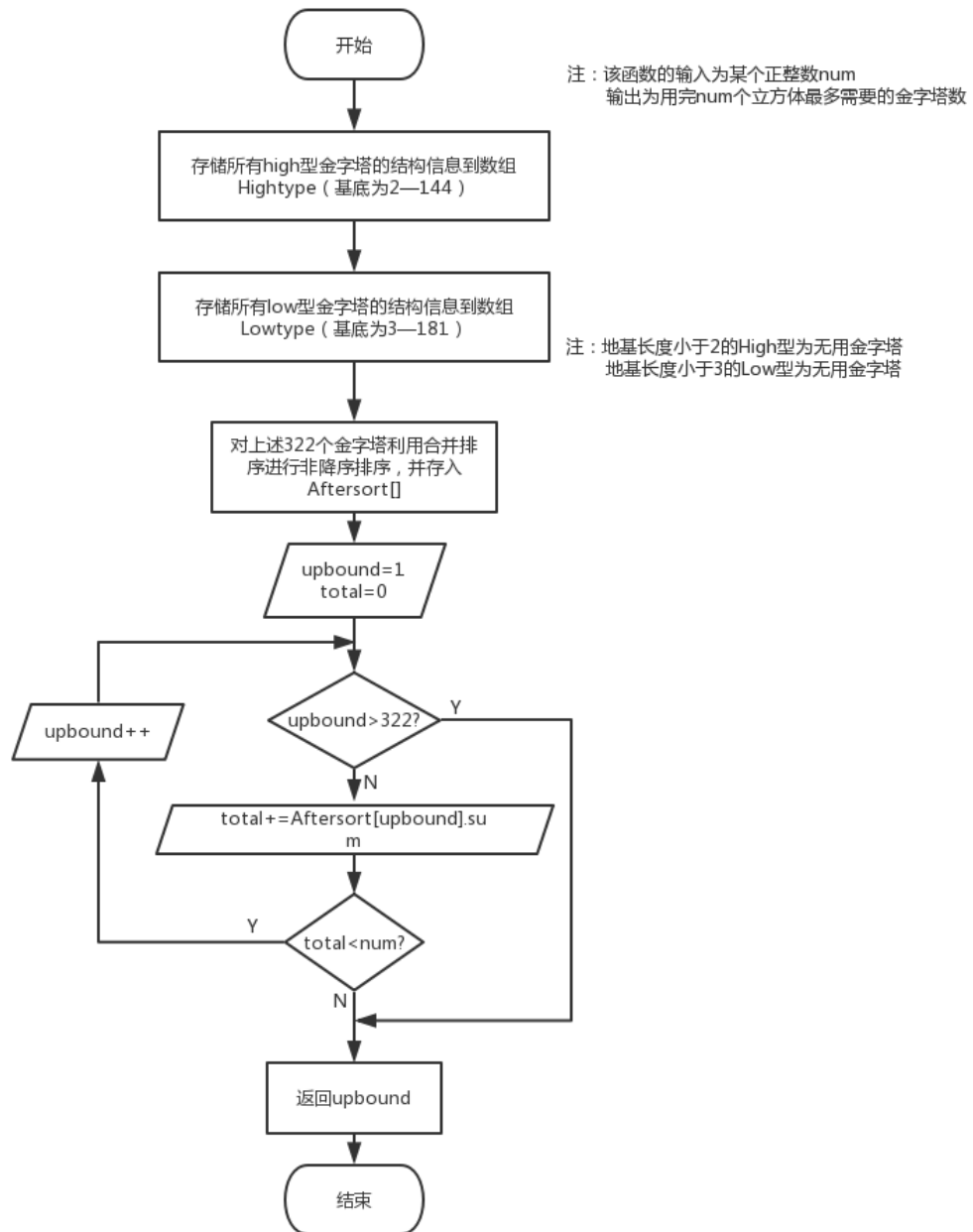
2 算法具体步骤

在 pyramid.c 源程序中我们采用了一个主函数和两个子函数，下面分别给出其算法流程图。

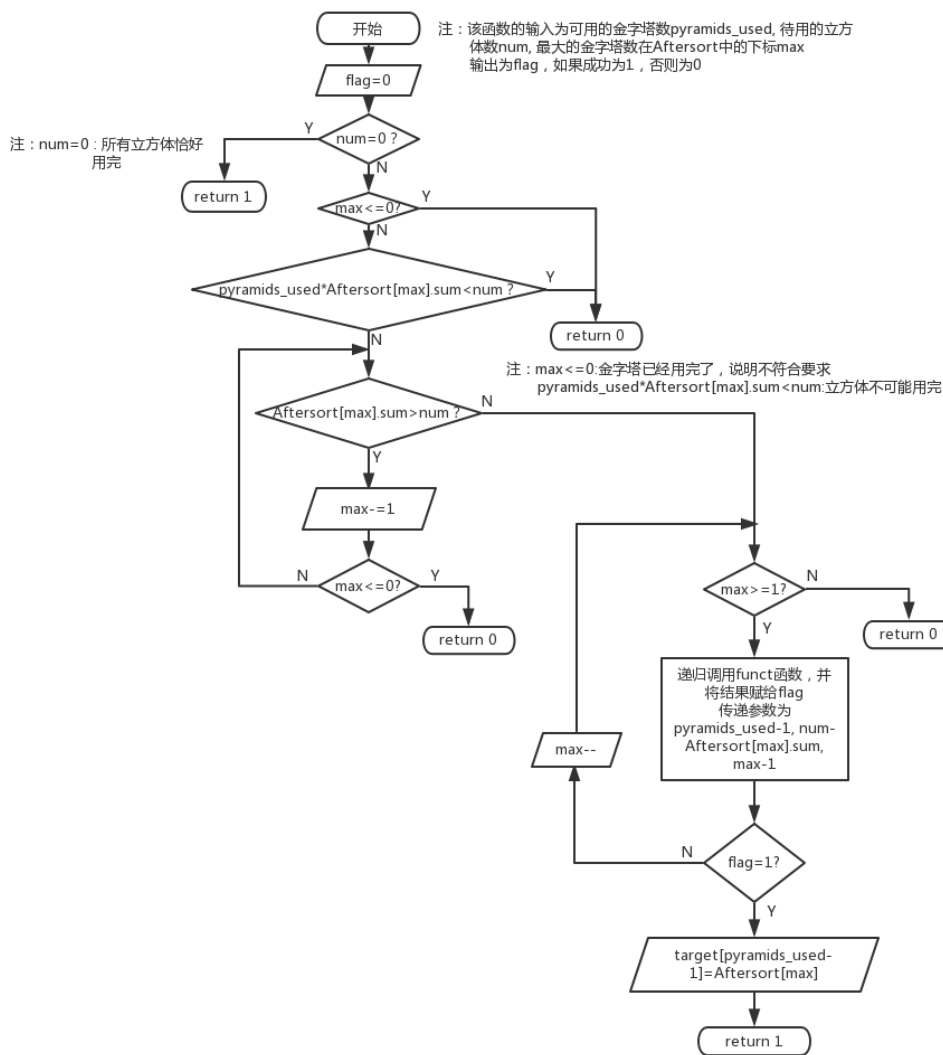
1. 主程序



2. Upbound 子程序



3. funct 子程序



3 算法实现要点

事实上在这个题目中，输出 **impossible**，即判断不可能的条件是非常难找的。因为对于计算机来说，第一次如果没有找到解，那么就会一次又一次地迭代下去。因此我们编写了一个求最多可能需要的金字塔数——上界（**upbound**）的函数，将基底为 2—144 的 **High** 金字塔和基底为 3—181 的 **Low** 金字塔所需的立方体数目作一个非降序排列存入一个数组，将该数组储存的数字依次相加，直到总和大于所给的立方体数，此时的金字塔数就是上界。因为要求得上界，我们就要考虑最差的情况，因此我们每次都优先选择所需立方体数最小的金字塔。由于所给立方体数目的限制，需要的立方体数不能超过所给立方体数，由此我们就得出了最多可能需要的金字塔数。那么在算法中，需要的金字塔数一旦超过这个上界，我们就能够得出在此情况下是 **impossible** 的。

另外在主要功能实现函数 `func` 中，要点是 `pyramid_used*Aftersort[max].sum<num`，`pyramid_used` 是目前能用的金字塔数目，`num` 是目前剩下的立方体数目，`Aftersort[max].sum` 是目前最接近 `num` 的金字塔需要的立方体数（一定 `<num`），如果此时

`pyramid_used*Aftersort[max].sum<num`，需要最多立方体数的金字塔乘以能用的金字塔数目依然小于剩下的立方体数，那么无论如何也不可能用完剩下的立方体，从而我们能够直接跳出循环，进行下一步的（`pyramid_used+1`）的工作，因而能够提前终止迭代。

另外一个要点是减治法的运用：如果再 `funct` 函数中找到了第一个可能的金字塔，下一步就是，更改参数——将能用的金字塔数-1，所剩的立方体数目-第一个金字塔用去的立方体数，重新调用 `funct` 函数，从而将问题减治为一个比原问题更小的问题。

4 实验结果展示及分析

4.1 测试

Case 1

输入：

```
29
28
0
```

期望输出：

```
Case 1: 3H 3L 2H
Case 2: impossible
```

实际输出：

```
Case 1: 3H 3L 2H
Case 2: impossible
Program ended with exit code: 0
```

测试通过。

测试目的：这是题目所给的样例，是最基本的一组测试样例。除了测试输出答案的准确性，还测试了对于多于一个 `case` 的情况的处理能力，我们的程序可以处理多个 `case`，当输入 0 的时候，程序终止。满足题目要求。

Case 2

输入：

```
-1
```

期望输出：

```
非法输入，程序应该立即终止
```

实际输出：

```
-1
Input number is invalid!
Program ended with exit code: 0
```

测试通过。

测试目的：虽然题目规定了输入在 1-1000000 之间，不可能小于 0，但是为了检验程序的鲁棒性，对非法输入做了测试。

Case 3

输入：

```
10000001
```

期望输出：

```
Case 1: 144H 181L 180L 143H 179L 142H 141H 139H 174L 135H 144L
```

实际输出：

Case 1: 144H 181L 180L 143H 179L 142H 141H 139H 174L 135H 144L

测试通过。

测试目的：虽然题目规定了输入在 1-1000000 之间，但是对于我们的程序，即使输入超过 1000000 的数，依然可以输出正确的答案，而且计算速度很快。

Case 4

输入：

1

期望输出：

Case 1: impossible

实际输出：

Case 1: impossible

测试通过。

测试目的：常规测试，测试了边界情况，也就是输入的最小值。

Case 5

输入：

1000000

期望输出：

Case 1: 142H 42H 38L

实际输出：

Case 1: 142H 42H 38L

测试通过。

测试目的：常规测试，测试了边界情况，也就是输入的最大值。

Case 6

输入：

500001

期望输出：

Case 1: 95H 106L 25H

实际输出：

Case 1: 95H 106L 25H

测试通过。

测试目的：常规测试，任意取了一个合法的输入值作测试。

测试总结：

首先我们测试了题目所给的样例，测试通过，同时输出的格式也都符合要求。我们又测试了两种非法输入，一种的输入小于0，此时程序应该终止，但是我们的程序对于这个输入，给出了友好的提示—— **Input number is invalid!**，并且程序终止—— **Program ended with exit code: 0**。对于边界情况，即输入是1和1000000时，测试也通过。此外，我们的程序对于大于1000000的输入，也能给出正确的结果，证明其有很强的鲁棒性。

4.2 分析

在这一部分，主要对算法的时间和空间复杂度作分析。

理论分析：

1、Upbound 函数

```
for(i=2; i<=144; i++)
{
    Hightype[i].base=i;
    Hightype[i].sum=i*(i+1)*(2*i+1)/6;    //存储所有high型金字塔的结构信息
    ( 基底为2——144 ) 。
    Hightype[i].type='H';
}

Lowtype[0].sum = Lowtype[1].sum = Lowtype[2].sum = 0;    //地基长度小于3的
Low型为无用金字塔

for(i=3; i<=181; i++)
{
    Lowtype[i].base=i;
    Lowtype[i].sum=i*(i+1)*(i+2)/6;        //存储所有low型金字塔的结构
    信息 ( 基底为3——181 ) 。
    Lowtype[i].type='L';
}
```

```
while(base_h<=144&&base_l<=181)
{
    if(Hightype[base_h].sum>=Lowtype[base_l].sum)
    {
        Aftersort[i]=Lowtype[base_l];
        base_l+=1;
    }
    else
    {
        Aftersort[i]=Hightype[base_h];
        base_h+=1;
    }

    i+=1;
}
```



```

if(base_h==145)
    while(i<=322)
    {
        Aftersort[i]=Lowtype[base_l];
        i++;
        base_l++;
    }
else
    while(i<=322)
    {
        Aftersort[i]=Hightype[base_h];
        i++;
        base_h++;
    }

```

```

for(upbound=1;upbound<=322;upbound++)
{
    total+=Aftersort[upbound].sum;
    if(total>=num)
        break;
}

```

我们在这个函数中的做法是事先计算出最坏输入，也即输入为 1000000 的时候，所需要的储存的信息的大小，因此对于所有输入，时间复杂度和空间复杂度都是一样的。144 次循环和 322 次循环，计算量也比较小，所以时间复杂度和空间复杂度都可以看做是 $O(1)$ 。

2、funct 函数

```

int funct(int pyramids_used, double num, int max)
{
    int flag=0;

    if(num==0) //所有立方体恰好用完，说明这种
                方案符合要求
        return 1;

    if(max<=0) //金字塔已经用完了，说明不符合
                要求
        return 0;
}

```

```

        if(pyramids_used*Aftersort[max].sum<num) //因为这种情况下肯定用不完所有立方体，所以肯定不符合要求。

            return 0;

        while(Aftersort[max].sum>num) //若第max个金字塔的立方体数超过了立方体总数，

        { //那么将max减一来看下一个立方体数较少的金字

            max-=1; //塔，直到找到一个立方体数少于总数的金字塔

            if(max<=0) //为止，若没有这样的金字塔，则说明不符合要求。

                return 0;

        }

        while(max>=1)

        {

            flag=funct(pyramids_used-1, num-Aftersort[max].sum, max-1); //递归

            if(flag==1)

            {

                target[pyramids_used-1]=Aftersort[max];

                return 1;

            }

            max--;

        }

        return 0;

    }

```

这是我们进行计算的主要函数。这里采用了递归的方法，递归表达式为 $T(n) = T(n-1) + c$ ，其中 $T(n)$ 代表输入为 n 时花费的时间， c 代表常数次操作时间。这样不难计算得到其时间复杂度为 $O(n)$ ，由于在这个函数中没有额外分配空间，但是由于用到了递归，在每次递归调用子函数之前，父函数的一些状态需要压到栈中，作最坏的打算，其空间复杂度也为 $O(n)$ （事实上空间花费很少）。

Upbound 和 funct 函数在 main 函数中是依次顺序调用的，并且 main 函数对他们的调用都是常数次，所以程序总的时间复杂度和空间复杂度是两个函数之和，都是 $O(n)$ 。这样看来，我们的程序的计算速度和空间消耗都是很理想的。

实际时间测试：

采用如下方法计算时间：

```
int loops = 1000; //运行相同程序loops次，考虑到单次程序运行太快，时间太少，
会导致计算不精确，循环重复运行loops次，最后时间除以loops次，可以提高精确度

clock_t clock(void); //计时开始
```

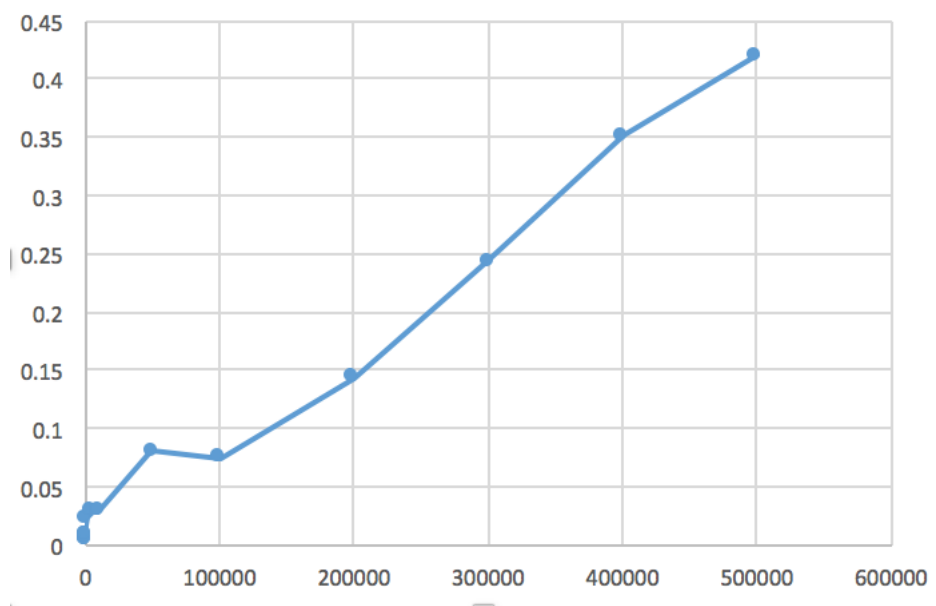
```
for(int i = 0; i < loops; ++i)
```

```
clock_t t = clock(); //计时结束

double mSec = 1000 * (t / CLOCKS_PER_SEC) / loops; //得到运行相同程序
loops次以后消耗的时间，除以loops以后，得到单次程序运行的时间，以毫秒为单位
printf("time: %lf", mSec);
```

测试结果：

input	time
1	0.00491
5	0.005004
10	0.005817
50	0.008947
100	0.009407
1000	0.022673
5000	0.028696
10000	0.028897
50000	0.080866
100000	0.074255
200000	0.143021
300000	0.243738
400000	0.351222
500000	0.419278



由图可知，程序运行消耗的时间线性增长，和理论分析一致。

5 心得体会

李梦圆：

这次主要和丁晨炜同学负责讨论算法设计部分的问题。最先开始的时候没有思路，只能想到用蛮力法来搜索，但其实蛮力法也不特别直接的可以看出来，因为确实这道题目涉及的组合情况太多。后来我们讨论出用 DFS 的方法，但无法找出一个合理的迭代上界。最后，经过讨论，想出了一个利用最差情况下所用金字塔数作为上界的方法，大大降低了最终算法的复杂度。在后期的编程过程中，也由于迭代较为复杂，出现了一些小的 bug，大多是因为数字取得不合理。通过这次大作业，我体会到利用编程之前必要的数学计算来改进算法的复杂度是大有裨益的。

吕珂杰：

在整个编程的过程中，我有很大一部分时间是花费在调试程序上面的。由于刚开始考虑得不是那么的全面，所以初步完成的程序存在着不少 bug，在经过多次修改以及用了大量实例进行测试验证以后，才得到了最后的正确结果。

张雨欣：

在测试过程中，发现 885569 个立方体数的输出出现了问题，本来只需要一个层数为 138 的 High 金字塔，但是输出中用了 11 个金字塔。经过对算法的检查，发现一开始存放在数组中的 High 金字塔置于 70 个，因此出现了差错。经过修改之后，算法能够正确无误地计算。因此，我明白测试在算法设计中是非常重要的，由于 885566 个立方体数目非常大，因此一开始没有想到要用这么大的数来测试，而都是选择比较小的数字测试，测试结果都是正确的导致我们认为这个算法没有问题，然而恰恰是在大的数据上面算法出错了。另外编程时还是应该细心，一些数组大小的设置、上限的设置都是非常重要的。

```
29
885569
0
case1:3H 3L 2H
case2:70H 69H 68H 67H 66H 65H 64H 63H 61H 20H 12L
请按任意键继续. . .
```

王竞康：

这次的编程作业我主要负责算法的改进。对于算法的改进，最为重要的一点就是优化金字塔数目的上界。具体而言就是寻找一个上界，使得任意满足题目的金字塔数目均小于这个上界。一开始我觉得这个上界不会很大，但是我尝试了很久并没有能够证明这个结论，只能作罢。我们最后采取的方法是考虑最差情况得到金字塔数目，虽然算法复杂度有所增加，但是也同样能够完成题目的要求，也不失为一种较优的策略。

丁晨炜：

这次大作业中，我主要是提供了其中一个解题的思路（dfs）以及程序测试分析，一开始觉得这个方法时间空间花销会很大，但是后来仔细分析了问题后，发现通过对问题的剪枝，以及通过李梦圆同学提供的数学证明，问题可以大大被简化。程序测试的主要工作是对各种可能的输入做试验，以检验程序的正确性和鲁棒性，并向 coder 提出优化与改进的建议。程序分析的主要工作是分析算法的时间和空间复杂度，经测试分析后，我们的程序在这两方面都做的还不错。

6 照片展示

（排名不分先后）

姓名	学号	照片
丁晨炜	3140103310	

张雨欣

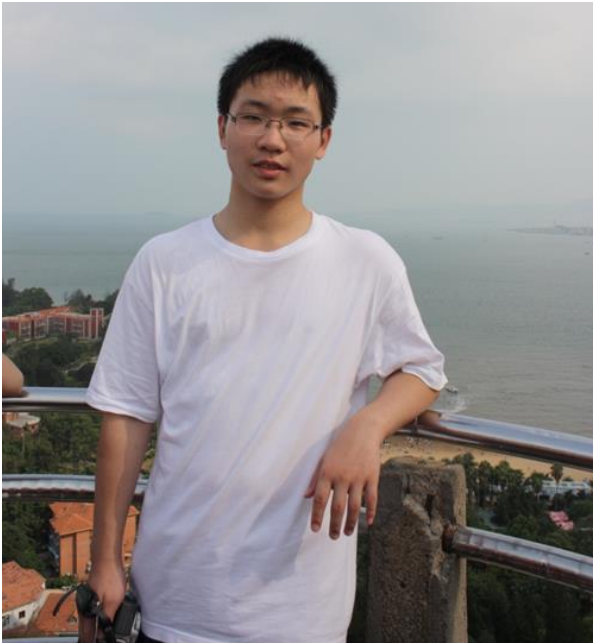

3140102326



李梦圆

3140104107



<p>吕珂杰</p>	<p>3140102486</p>	 A young man with short black hair and glasses, wearing a white t-shirt, stands on a balcony. He is leaning his left arm on a metal railing. In the background, there is a body of water, some distant land, and a building with a red roof.
<p>王竞康</p>	<p>3140104740</p>	 A close-up photograph of a Shiba Inu dog's face. The dog has light brown fur, dark eyes, and a black nose. It is looking directly at the camera. Its front paws are visible, resting on a blue surface.