Stanford CS106探秘

BigO

1

```
int addInteger(int N) {
    int sum = 0;
    for (int i = 1; i <= N + 2; i++) {
        sum++;
    }
    for (int j = 1; j <= N * 5; j++) {
        sum++;
    }
    return sum;
}
```

代码包含两个并行的 `for` 循环。总操作次数 ≈ $N + 2 + 5N$ 故该函数的时间复杂度为 $O(N)$。

2

```
int showString(string str) {
    int N = str.size();
    string sumString;
    for (int i = 1; i < str.size(); i *= 3) {
        cout << str[i] << endl;
        sumString += str[i];
    }
    return sumString.size();
}
```

代码有一个 `for` 循环。循环变量 `i` 的更新方式是 `i *= 3`。这意味着 `i` 的值呈指数级增长：1, 3, 9, 27, ...相应地时间复杂度为 $O(logN)$。

3

```
int myfunction(Vector<int> vec) {
    int N = vec.size();
    int sum = 0;
    for (int i = 0; i < vec.size(); i += (vec.size() / 6)) {
        cout << vec[i] << endl;
        sum += vec[i];
    }
    return sum;
}
```

代码有一个 `for` 循环。循环的步长（增量）是 `N / 6` (整数除法)。因此，无论 `N` 的值有多大（只要 `N >= 6`），这个循环都只执行大约 6 次。如果 `N < 6`， `N/6` 的结果是 0，这将导致一个无限循环（忽略

这种情况）。由于循环的执行次数是一个固定的常数（约 6 次），它不随输入规模 `N` 的变化而变化。该函数的时间复杂度为 $O(1)$ (常量时间)。

4

```
int myfunction(Vector<int> vec) {
    int N = vec.size();
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < i; j++) {
            cout << vec[j] << endl;
        }
    }
}
```

两个嵌套的 `for` 循环。

1. **外层:** `for (int i = 0; i < N; i++)` 执行 `N` 次。
2. **内层:** `for (int j = 0; j < i; j++)` 的执行次数依赖于外层循环的变量 `i`。
3. **总复杂度:** 结果为 $\frac{(N-1)\times N}{2} = \frac{N^2-N}{2}$。

结论:
该函数的时间复杂度为 $O(N^2)$。

5

```
int myfunction(Vector<int> vec) {
    int N = vec.size();
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < N; j += N/6) {
            cout << vec[j] << endl;
        }
    }
}
```

两个嵌套的 `for` 循环。

1. **外层:** `for (int i = 0; i < 10; i++)` 循环的次数是固定的 10 次，不随 `N` 变化。这是一个常数次操作。
2. **内层:** `for (int j = 0; j < N; j += N/6)` 与第3题一样。它执行大约 6 次，也是一个常数次操作。

该函数的时间复杂度为 $O(1)$ (常量时间)。

6

```
int myfunction(Vector<int> &vec, int key) {
    int N = vec.size();
    for (int i = 1; i < N; i++) {
        if (vec[i] == key) {
            cout << "find " << key << " at " << i;
            return i;
```

```
            }
        }
        return -1;
    }
```

代码有一个 `for` 循环，从索引 1 遍历到 `N-1`。如果要查找的 `key` 就在 `vec[1]`，循环只执行一次就 `return`。时间复杂度是 $O(1)$；如果要查找的 `key` 是向量的最后一个元素，或者 `key` 根本不在向量中，循环会执行 `N-1` 次。时间复杂度是 $O(N)$；平均地，需要检查大约一半的元素，即 `N/2` 次。时间复杂度也是 $O(N)$。故该函数的时间复杂度为 $O(N)$。

7

```
int myfunction(int N) {
    if (N <= 1) {
        // base case
        return 1;
    } else {
        // recursive case
        return 2 * myfunction(N/2);
    }
}
```

这是一个递归函数。每一次递归调用 `myfunction` `N` 的值减小一半；故该函数的时间复杂度为 $O(logN)$。

**总结:

1. **关注执行次数最多的代码块:** 通常是循环或者递归。一个算法的复杂度由最高阶的项决定。
2. **分析循环:**
   - **单层循环:** 循环 `N` 次，复杂度为 $O(N)$。
   - **嵌套循环:** 复杂度是各层循环复杂度的乘积。例如，两个都与 `N` 相关的嵌套循环通常是 $O(N^2)$。
   - **对数循环:** 如果循环的控制变量是按乘法/除法变化的（如 `i *= 2`），那么这个循环的复杂度是对数级的 $O(logN)$。
   - **常数循环:** 如果循环次数是固定的（如 `for (int i = 0; i < 10; i++)` 或者 `for (int i = 0; i < N; i += N/6)`），或者循环次数不随输入规模 `N` 变化，其复杂度为 $O(1)$。
3. **分析递归:**
   - 写出时间复杂度的**递推关系式** `T(N)`。
   - 关注递归函数的传参。
4. **最后注意:**
   - **只保留最高阶项:** 例如，在 $N^2 + N + 100$ 中，只保留 N^2。
   - **忽略所有常数系数:** 例如，$5N^2$ 简化为 N^2，$100N$ 简化为 N。$O(100)$ 就是 $O(1)$。

`random.h`

函数 (Function)	函数原型 (Prototype)	功能说明 (Description)	关键点 / 区间 (Key Point / Range)
整数生成	`int randomInteger(int low, int high);`	生成一个随机整数。	**包含** `low` 和 `high`，即区间为 `[low, high]`。
浮点数生成	`double randomReal(double low, double high);`	生成一个随机浮点数。	**包含** `low`，但**不包含** `high`，即区间为 `[low, high)`。
概率判断	`bool randomChance(double p);`	根据指定的概率 `p` 返回 `true`。	`p` 的取值范围是 `[0.0, 1.0]`。
布尔值	`bool randomBool();`	以50%的概率返回 `true`。	等同于 `randomChance(0.5)`。
种子设置	`void setRandomSeed(int seed);`	设置伪随机数生成器的种子。	用于创建可复现的随机序列，方便调试。

`strlib.h`

功能类别	函数 (Function)	函数原型 (Prototype)	功能说明与关键点
类型转换	`integerToString`	`string integerToString(int n, int radix = 10);`	将整数转换为字符串。可以指定进制（radix），默认为10进制。
	`stringToInteger`	`int stringToInteger(const string& str, int radix = 10);`	将字符串转换为整数。同样可以指定进制。若格式非法会抛出错误。
	`doubleToString`	`string doubleToString(double d);`	将浮点数转换为字符串。
	`stringToDouble`	`double stringToDouble(const string& str);`	将字符串转换为浮点数。
	`boolToString`	`string boolToString(bool b);`	将布尔值 `true` 或 `false` 转换为对应字符串 "true" 或 "false"。
	`stringToBool`	`bool stringToBool(const string& str);`	将字符串 "true" 或 "false" 转换为对应布尔值。
查找与替换	`stringIndexOf`	`int stringIndexOf(const string& s, const string& sub, ...);`	查找子字符串 `sub` **首次出现**的位置，返回索引。未找到则返回-1。
	`stringLastIndexOf`	`int stringLastIndexOf(const string& s, const string& sub, ...);`	查找子字符串 `sub` **最后一次出现**的位置，返回索引。未找到则返回-1。
	`stringReplace`	`string stringReplace(const string& str, const string& old, ...);`	返回一个新字符串，其中所有的 `old` 子串都被替换。
	`stringReplaceInPlace`	`int stringReplaceInPlace(string& str, const string& old, ...);`	**直接在原字符串上**进行替换，更高效。返回值为替

功能类别	函数 (Function)	函数原型 (Prototype)	功能说明与关键点
			换发生的次数。
检查与判断	`startsWith`	`bool startsWith(const string& str, const string& prefix);`	判断字符串 `str` 是否以 `prefix` 开头。
	`endsWith`	`bool endsWith(const string& str, const string& suffix);`	判断字符串 `str` 是否以 `suffix` 结尾。
	`stringContains`	`bool stringContains(const string& s, const string& sub);`	判断字符串 `s` 是否包含子串 `sub`。
	`equalsIgnoreCase`	`bool equalsIgnoreCase(const string& s1, const string& s2);`	判断两个字符串在**忽略大小写**的情况下是否相等。
	`stringIsInteger`	`bool stringIsInteger(const string& str, int radix = 10);`	判断一个字符串能否被成功转换为整数。
	`stringIsReal`	`bool stringIsReal(const string& str);`	判断一个字符串能否被成功转换为浮点数。
修改与操作	`toUpperCase`	`string toUpperCase(const string& str);`	返回一个所有字母都转为**大写**的新字符串。
	`toLowerCase`	`string toLowerCase(const string& str);`	返回一个所有字母都转为**小写**的新字符串。
	`toUpperCaseInPlace`	`void toUpperCaseInPlace(string& str);`	**直接在原字符串上**进行大写转换。
	`toLowerCaseInPlace`	`void toLowerCaseInPlace(string& str);`	**直接在原字符串上**进行小写转换。
	`trim`	`string trim(const string& str);`	返回一个移除了**首尾空白字符**的新字符串。
	`trimInPlace`	`void trimInPlace(string& str);`	**直接在原字符串上**移除首尾空白字符。
	`stringSplit`	`Vector<string> stringSplit(const string& str, const string& delimiter, ...);`	根据分隔符 `delimiter` 将字符串分割成一个 `Vector<string>`。
	`stringJoin`	`string stringJoin(const Vector<string>& v, const string& delimiter);`	将一个 `Vector<string>` 的所有元素用分隔符 `delimiter` 连接成一个新字符串。

`Stream`

操纵符 (Manipulator)	含义 (Meaning)
`endl`	将结束序列插入到输出流，并确保输出的字符能被写到目的地流中。
`setw(n)`	将下一个输出字段的宽度设置为 n 个字符。如果输出值所需空间小于 n，则额外的空间用填充字符填充。这种性质是暂时的，这意味着它只影响下

操纵符 (Manipulator)	含义 (Meaning)
	一个流中的数据值的输出宽度。
`setprecision(digits)`	将输出流的精度设置为 digits。精度设定的解释依赖于其他的设置。如果已经将模式设置为 `fixed` 或 `scientific`， `digits` 会指定小数点后数字的位数。如果你没有设置以上两种模式， `digits` 表示有效数字的总位数，并且不考虑小数点。这种性质是持久的，直到它被明确地改变为止。
`setfill(ch)`	为流设置填充字符 `ch`。默认地，如果需要额外的字符填充到 `setw` 设置的字段宽度中，则空白格作为填充字符输出。调用 `setfill` 使输出流可以改变填充字符。例如，调用 `setfill('0')` 意味着字段将用 0 填充。这种性质是持久的。
`left`	指定输出字段为左对齐，这意味着任何填充字符都在数值之后插入。这种性质是持久的。
`right`	指定输出字段为右对齐，这意味着任何填充字符都在数值之前插入。这种性质是持久的。
`fixed`	指定之后的浮点数输出应该完整地呈现，并且不使用科学计数法。默认地，浮点数应该以最简洁的形式呈现。这种性质是持久的。 [cite:-1]
`scientific`	指定之后的浮点数输出应该以科学计数法的形式呈现。这种性质是持久的。
`showpoint` / `noshowpoint`	这两个流操纵符控制浮点数中是否出现小数点。这种控制同样适用于整数的情况。可以用 `showpoint` 强制要求出现小数点，然后通过 `noshowpoint` 来恢复默认的状况。
`showpos` / `noshowpos`	这两个流操纵符控制在一个正数前是否有应一个正号。默认地，正数前没有正号。这种性质是持久的。
`uppercase` / `nouppercase`	这两个流操纵符控制作为数据转换的一部分所产生的字母的大小写，例如科学计数法中的大写字母 `E`。默认地，字符以小写字母呈现。这种性质是持久的。
`boolalpha` / `noboolalpha`	这两个流操纵符控制布尔值的格式，它一般使用它们在数值的值 `0` 和 `1` 的形式表示呈现。使用 `boolalpha` 操纵符导致它们以 `true` 或 `false` 的形式出现。这种性质是持久的。

输入流操纵符 (`<iostream>`)

操纵符 (Manipulator)	含义 (Meaning)
`skipws` / `noskipws`	这两个流操纵符控制提取操作符 `>>` 在读取一个值之前是否忽略空白字符。如果指定 `noskipws`，提取操作符将所有的字符（包括空白字符）看作是输入字段的一部分。之后可以使用 `skipws` 恢复默认的行为。这个性质是持久的。
`ws`	从输入流中读取字符，直到它不属于空白字符。因此，这个流操纵符的作用是跳过输入中的任何空白字符、制表符和换行符。不像 `skipws` 和 `noskipws` 改变的是流关于之后的输入操作行为， `ws` 流操纵符是立即起作用的。

所有流都支持的方法	
`stream.fail()`	如果流处于失效状态，则返回 `true`。这个条件通常发生在你尝试超出文件的结尾去读取数据的时候，但这也表示数据中出现了不完整性错误。
`stream.eof()`	如果流位于文件的结尾，则返回 `true`。鉴于 C++ 流的语义， `eof` 方法只用在一个 `fail` 调用之后，那时 `eof` 调用允许你判断故障是否是由于

所有流都支持的方法	
	文件的结尾引起的。
`stream.clear()`	重置与流相关的状态位。当一个故障发生后，无论何时需要必须调用这个函数。
`if (stream) ...`	判断流是否有效。就大部分情况而言，这个测试和 `if (!stream.fail())` 的效果相同。
所有文件流都支持的方法	
`stream.open(filename)`	尝试打开文件 `filename` 并将其附加到流中。流的方向由流的类型所决定：输入流用于输入打开，输出流用于输出打开。 `filename` 参数是一个 C 风格的字符串，这意味着你将需要任何 C++ 字符串上调用 `c_str` 。通过调用 `fail` ，可以检测 `open` 方法是否失败。
`stream.close()`	关闭依附于流的文件。
所有输入流都支持的方法	
`stream >> variable`	将格式化数据读入到一个变量中。数据的格式是由变量类型控制的，并且无关流操纵符是什么，它都是有效的。
`stream.get(var)`	将下一个字符读入到字符变量 `var` 中， `var` 是引用参数。返回值是流本身，这使得它有更多的字符去写，设置 `fail` 标志。
`stream.get()`	返回流的下一个字符。返回值是一个整数，它可识别以常量 `EOF` 表示文件结尾。
`stream.unget()`	复制流的内部指针最后读取的一个字符再次被下一个 `get` 调用读取。
`getline(stream, str)`	将流 `stream` 中的下一行读入到字符串变量 `str` 中。 `getline` 函数返回流，它简化了文件结尾的测试。
所有输出流都支持的方法	
`stream << expression`	将格式化数据写入到一个输出流。数据的格式由表达式的类型所控制，并且对于任何输出流操纵符都有效。
`stream.put(ch)`	将字符 `ch` 写入到输出流。

`simpio.h`

函数 (Function)	**摘要 (Summary)**
`getInteger(prompt)`	从cin中读取一个完整的行，并且尝试将它当作一个整数输入。
`getReal(prompt)`	从cin中读取一个完整的行，并且尝试将它当作一个浮点数输入。
`getLine(prompt)`	从cin中读取一个完整的行，并且将该行文本作为一个字符串串返回。

- `int getInteger(string prompt = "");`
  从cin中读取一个完整的行，并且将它当作是一个整数输入。如果输入成功，返回该整数值。如果参数不是一个合法的整数或者字符串中出现无关字符（除了空白），给予用户一次重新输入值的机会。如果提交，可选择的 `prompt` 字符串将会在读取值之前打印。
  用法: `int n = getInteger(prompt);`
- `double getReal(string prompt = "");`
  从cin中读取一个完整的行，并且将它当作是一个浮点数输入。如果输入成功，返回浮点数值。如果参

数不是一个合法的浮点数或者字符串中出现无关字符（除了空白），给予用户一次重新输入值的机会。如果提交，可选择的 `prompt` 字符串将会在读取值之前打印。
用法: `double x = getReal(prompt);`

- `string getLine(string prompt = "");`
从cin中读取一行文本，并且将该行文本作为一个字符串串返回。结束输入的下一行的字符不会作为返回值的一部分进行存储。如果提交，可选择的 `prompt` 字符串将会在读取值之前打印。
用法: `string line = getLine(prompt);`

排序

归并排序

```
void sort(Vector<int> & vec) {
   int n = vec.size();
   if (n <= 1) return;
   Vector<int> v1;
   Vector<int> v2;
   for (int i = 0; i < n; i++) {
      if (i < n / 2) {
         v1.add(vec[i]);
      } else {
         v2.add(vec[i]);
      }
   }
   sort(v1);
   sort(v2);
   vec.clear();
   merge(vec, v1, v2);
}

void merge(Vector<int> & vec, Vector<int> & v1, Vector<int> & v2) {
   int n1 = v1.size();
   int n2 = v2.size();
   int p1 = 0;
   int p2 = 0;
   while (p1 < n1 && p2 < n2) {
      if (v1[p1] < v2[p2]) {
         vec.add(v1[p1++]);
      } else {
         vec.add(v2[p2++]);
      }
   }
   while (p1 < n1) {
      vec.add(v1[p1++]);
   }
   while (p2 < n2) {
      vec.add(v2[p2++]);
   }
}
```

快速排序

```
void sort(Vector<int> & vec) {
   quicksort(vec, 0, vec.size() - 1);
}

void quicksort(Vector<int> & vec, int start, int finish) {
   if (start >= finish) return;
   int boundary = partition(vec, start, finish);
   quicksort(vec, start, boundary - 1);
   quicksort(vec, boundary + 1, finish);
}

int partition(Vector<int> & vec, int start, int finish) {
   int pivot = vec[start];
   int lh = start + 1;
   int rh = finish;
   while (true) {
      while (lh < rh && vec[rh] >= pivot) rh--;
      while (lh < rh && vec[lh] < pivot) lh++;
      if (lh == rh) break;
      int tmp = vec[lh];
      vec[lh] = vec[rh];
      vec[rh] = tmp;
   }
   if (vec[lh] >= pivot) return start;
   vec[start] = vec[lh];
   vec[lh] = pivot;
   return lh;
}
```

`collections`

一般考试试卷最后会给，实在不行翻书，再不济ctrl点击 `#include 'map.h'` 中的 `map.h` 会跳转到头文件

继承

方面 (Aspect)	父类 (Superclass)	子类 (Subclass)
基本关系	提供基础的功能和属性，是一个更通用的概念。	继承父类的行为，并可以添加或特化自己的功能 。子类是父类的一种特殊形式（"is a" 关系）。
声明语法	`class 父类名 { ... };`	`class 子类名 : public 父类名 { ... };`

方面 (Aspect)	父类 (Superclass)	子类 (Subclass)
访问控制 (Access Control)	• `public` 成员：可以被子类和所有客户端访问。 `protected` 成员：可以被子类访问，但不能被外部客户端访问。 `private` 成员：**不能**被子类直接访问 。	• 可以直接访问父类的 `public` 和 `protected` 成员。• **不能**直接访问父类的 `private` 成员。
方法重写 (Method Overriding)	要想让子类重写的方法能通过父类指针或引用被正确调用，父类中的方法**必须**声明为 `virtual` 。 可以将方法声明为纯虚函数 (`virtual ... = 0;`)，这意味着父类不提供实现，强制子类必须提供自己的实现。	子类可以提供一个与父类中 `virtual` 方法签名完全相同的实现，从而“重写”该方法的功能 。当通过父类指针调用该虚方法时，程序会执行子类的版本。
构造函数 (Constructor)	在子类的构造函数被调用时，父类的某个构造函数会先被调用 。	子类的构造函数会**自动调用**父类的默认（无参数）构造函数 。也可以使用**初始化列表**语法 (`: 父类名(参数)`) 来显式调用父类的特定构造函数。
析构函数 (Destructor)	在涉及动态内存分配的继承体系中，父类的析构函数**应当**被声明为 `virtual` 。	如果父类的析构函数是 `virtual` 的，当通过父类指针 `delete` 一个子类对象时，会先调用子类的析构函数，再调用父类的析构函数，确保完全正确的内存清理。
对象赋值与存储	当一个子类对象被赋给一个父类类型的变量时，只会复制父类所包含的部分，子类特有的成员会被切掉，这种现象称为 **slicing**。	为了避免 slicing 问题，处理继承体系中的对象时，应使用**指针**（如 `父类名*` ）而不是对象本身 。例如，`Vector<父类名*>` 是正确的存储方式，而 `Vector<父类名>` 会导致 slicing 。

画图题模板

以 `Assignment3` 为例：

```
/*
 * File: SierpinskiTriangle.cpp
 * Assignment #3.
 */

#include <iostream>
#include <cmath>
#include "gwindow.h"
#include "console.h"
#include "simpio.h"

using namespace std;

/* Function prototypes */

void drawSierpinskiTriangle(GWindow & gw, double x, double y, double size, int
order);
```

```
/* Constants */

const double WINDOW_WIDTH = 700;
const double WINDOW_HEIGHT = 470;
const double COS60 = sqrt(3.0) / 2;
const double SIZE = 520;

/* Main program */

int main() {
    while (1)
    {
        int order = getInteger("Please input the order(>0):");
        if (order <= 0)
        {
            cout << "Program finished !";
            break;
        }
        GWindow gw(WINDOW_WIDTH, WINDOW_HEIGHT);
        gw.setLocation(50, 50);
        gw.setColor("Black");
        double x0 = (WINDOW_WIDTH - SIZE) / 2;
        double y0 = WINDOW_HEIGHT - (WINDOW_HEIGHT - SIZE * COS60) / 2;
        drawSierpinskiTriangle(gw, x0, y0, SIZE, order);
    }
    return 0;
}

/*
 * Function: drawSierpinskiTriangle
 * Usage: drawSierpinskiTriangle(gw, x, y, size, order);
 *
 * Draws a Sierpinski Triangle of the specified size and order.
 * The upper left corner of the triangle is at the point (x, y).
 */

void drawSierpinskiTriangle(GWindow & gw, double x, double y, double size, int order)
{
    if (order == 0) {
        gw.drawLine(x, y, x + size / 2, y - size * COS60);
        gw.drawLine(x + size / 2, y - size * COS60, x + size, y);
        gw.drawLine(x + size, y, x, y);
    } else {
        drawSierpinskiTriangle(gw, x, y, size / 2, order - 1);
        drawSierpinskiTriangle(gw, x + size / 4, y - size * COS60 / 2, size / 2,
order - 1);
        drawSierpinskiTriangle(gw, x + size / 2, y, size / 2, order - 1);
    }
}
```

完整的读写文件

以2023-2024考试第一题为例

```cpp
#include <iostream>
#include <fstream> // 使用 fstream 库进行文件读写

using namespace std;

// --- 为了使代码示例能够独立运行，此处添加了必要的类定义 ---
class Cake {
public:
    virtual ~Cake() {}
    virtual double cakeprice() const = 0;
    double get_density() const { return density; }
    double get_unitPrice() const { return unitPrice; }
protected:
    Cake(double d, double p) : density(d), unitPrice(p) {}
    double density;
    double unitPrice;
};

class CubiodCake : public Cake {
public:
    CubiodCake(double l, double w, double h, double d, double p)
        : Cake(d, p), length(l), width(w), height(h) {}
    // (1) 返回长方体蛋糕价格
    double cakeprice() const override;
private:
    double length, width, height;
};

class CylinderCake : public Cake {
public:
    CylinderCake(double r, double h, double d, double p)
        : Cake(d, p), radius(r), height(h) {}
    // (2) 返回圆柱体蛋糕价格
    double cakeprice() const override;
private:
    double radius, height;
};
// --- 类定义结束 ---


/*(1)请在此处实现 CubiodCake 的成员函数cakeprice(),返回特定的某个长方体
* 形状蛋糕的价格,计算公式:蛋糕价格=体积*密度*单价。*/
double CubiodCake::cakeprice() const {
    return length * width * height * get_density() * get_unitPrice();
}

/* (2)请在此处实现 CylinderCake 的成员函数 cakeprice(),返回特定的某个圆柱体
* 形状蛋糕的价格,计算公式:蛋糕价格=体积*密度*单价。*/
double CylinderCake::cakeprice() const {
    return 3.14 * radius * radius * height * get_density() * get_unitPrice();
}

int main() {
```

```
    // 准备从 "cakes.txt" 文件读取数据
    ifstream inputFile("cakes.txt");

    // (3) 建立 ofstream 类对象 outputFile,并同时打开磁盘文件 totalcost.txt
    ofstream outputFile("totalcost.txt");

    // (4) 判断是否成功打开了文件
    if (!inputFile.is_open() || !outputFile.is_open()) {
        cerr << "Error opening files!" << endl;
        return 1;
    }

    double totalCost = 0;
    char type;

    /* (5) 修改下面while语句中的true条件部分,读取 cakes.txt 中每一行数据的蛋糕类型 type,
     * 提示: 请用操作符>>,该操作若读取成功返回ture,若遇文件结束则返回false. */
    while (inputFile >> type) {
        if (type == 'U') {
            double length = 0, width = 0, height = 0, density = 0, price = 0;
            // (6) 从磁盘文件依次读入 length, width, height, density, price
            inputFile >> length >> width >> height >> density >> price;
            CubiodCake cubiodCake(length, width, height, density, price);
            double cost = cubiodCake.cakeprice();
            totalCost += cost;
        } else if (type == 'Y') {
            double radius = 0, height = 0, density = 0, price = 0;
            // (7) 请从磁盘文件依次读入 height, radius, density, price
            inputFile >> height >> radius >> density >> price;
            CylinderCake cylinderCake(radius, height, density, price);
            double cost = cylinderCake.cakeprice();
            totalCost += cost;
        }
    }

    // (8) 将计算得到的 totalCost 的值写入磁盘文件 totalcost.txt
    outputFile << totalCost;

    // (9) 关闭打开的文件
    inputFile.close();
    outputFile.close();

    return 0;
}
```

把作业解答和书都带上，祝大家好运~