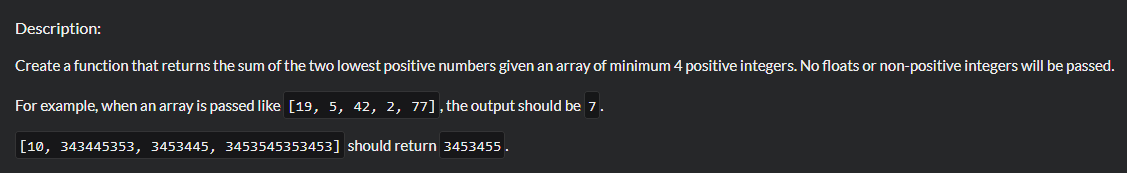
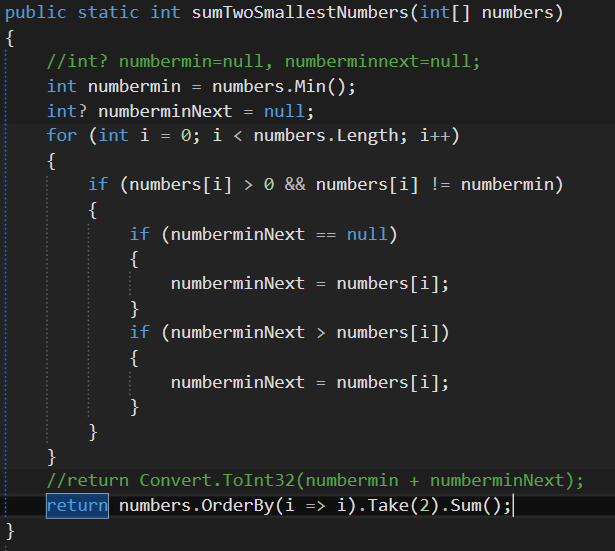
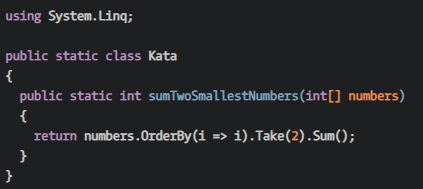
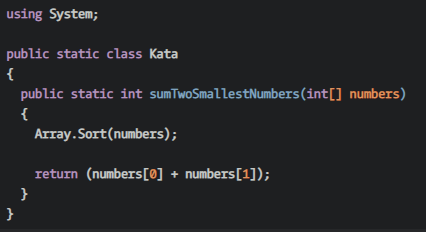
1. 

我的解决方案:



其他人的解决方案:





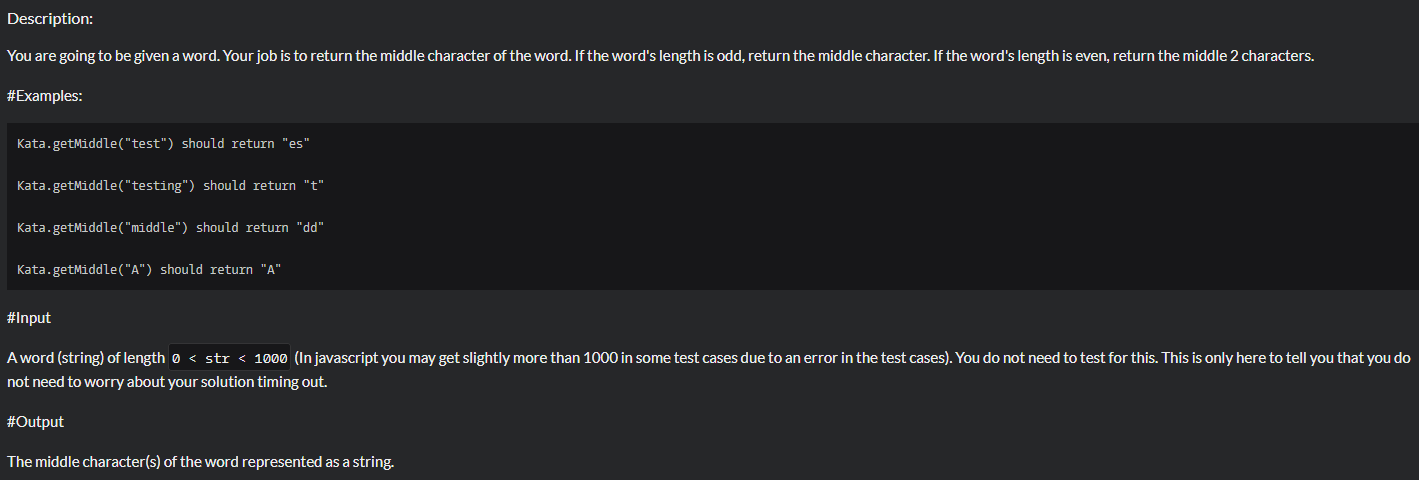
这么一对比我写的真他娘的烂!!

C# 中的int?:表示可空类型,就是一种特殊的值类型,它的值可以为null

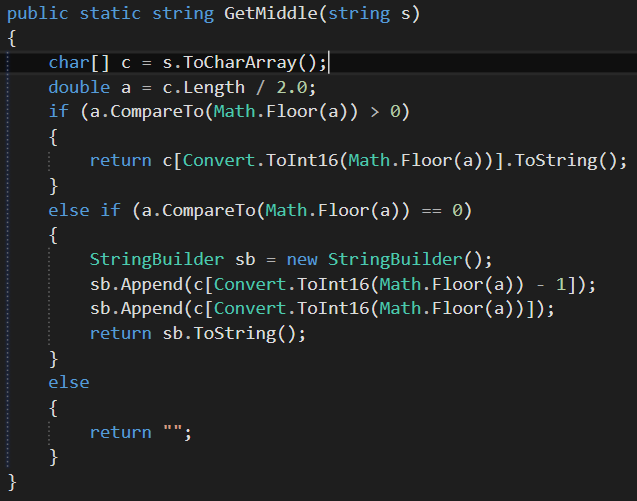
对于数组int[]即Array,我们可以通过Array.Sort()进行排序等操作.int[]即Array

其中i=>i即为Lambda表达式,参数为i,得到i值.并通过OrderBy进行排序

2.

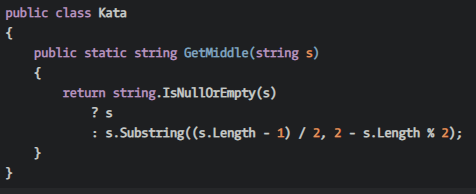


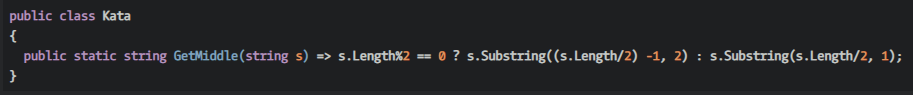
我的解决方案：

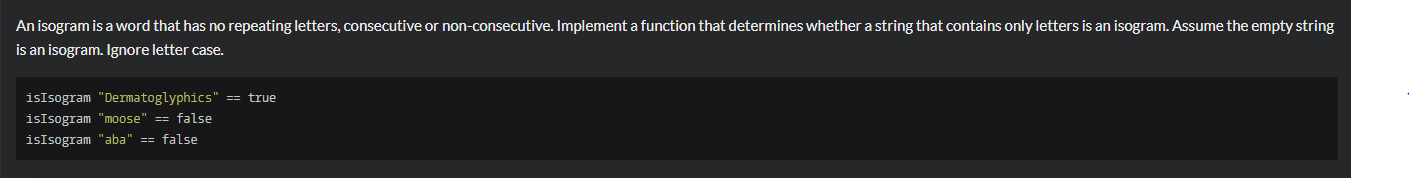


其他人的解决方案：

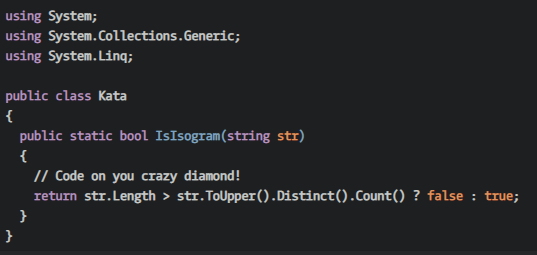
Best One



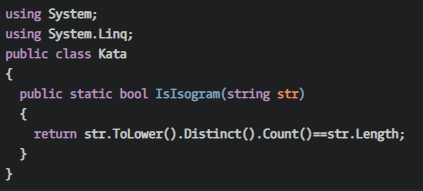


3. 

我的解决方案：

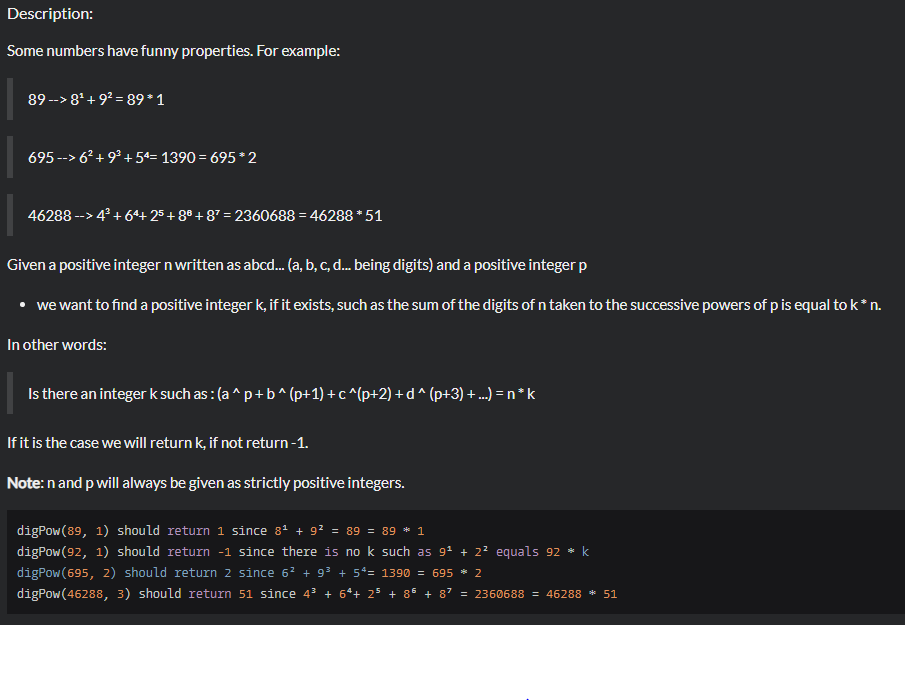


网上的解决方案：

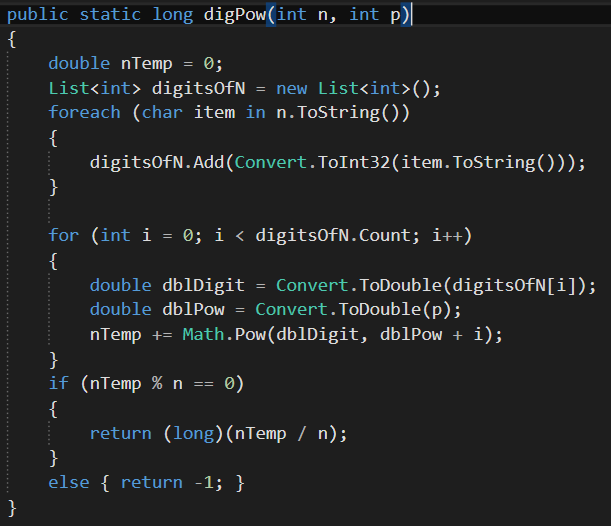


C#集合中的Distinct()方法可以对集合元素去重,如果集合内部元素为值类型,则Distinct方法根据值类型是否相等来判断去重,如果list集合内部元素为引用类型变量,则是判断相同引用的对象来去重.

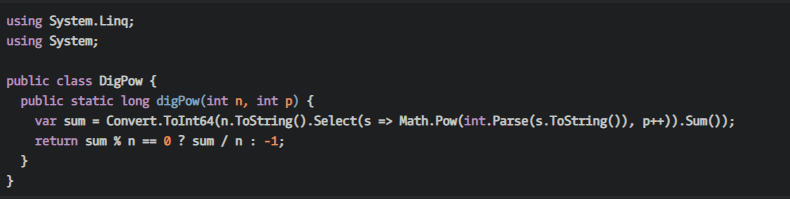
4.



我的解决方案：



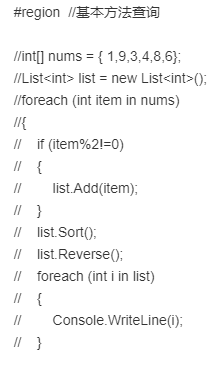
网上的解决方案：

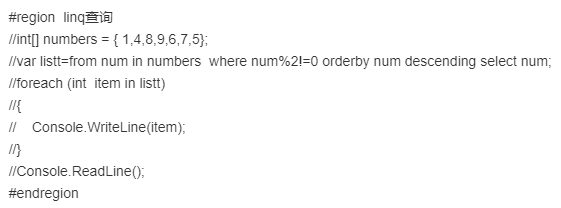


Select()只可用于Ienumberable类型，取出并处理

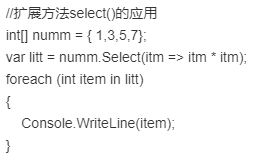
Linq中的Select同Sql中的Select，即输出我们想要的内容。

Linq可以对集合如数组、泛型等操作。

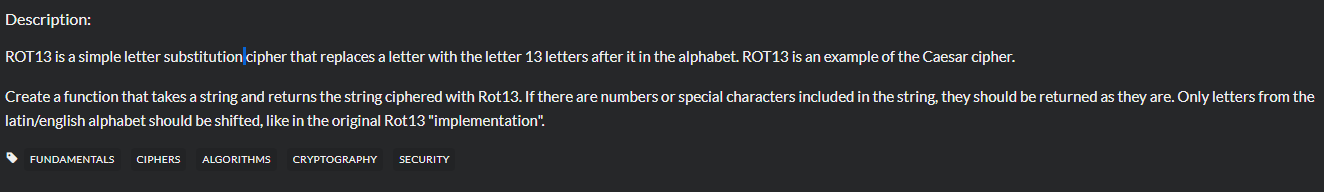




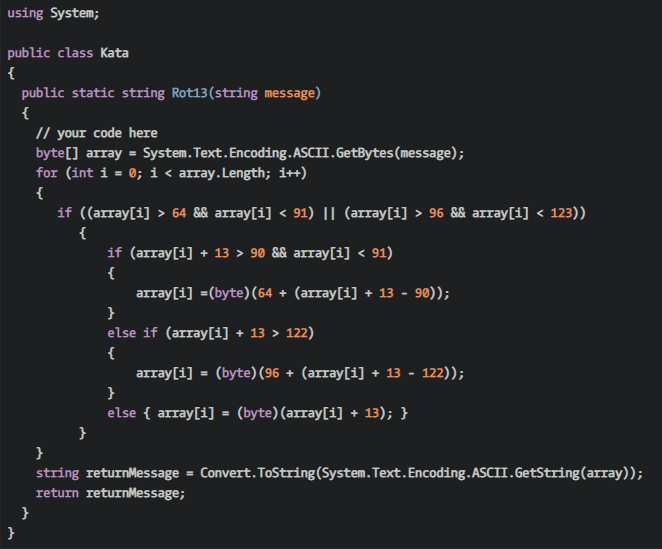
查询过程类似于sql语句



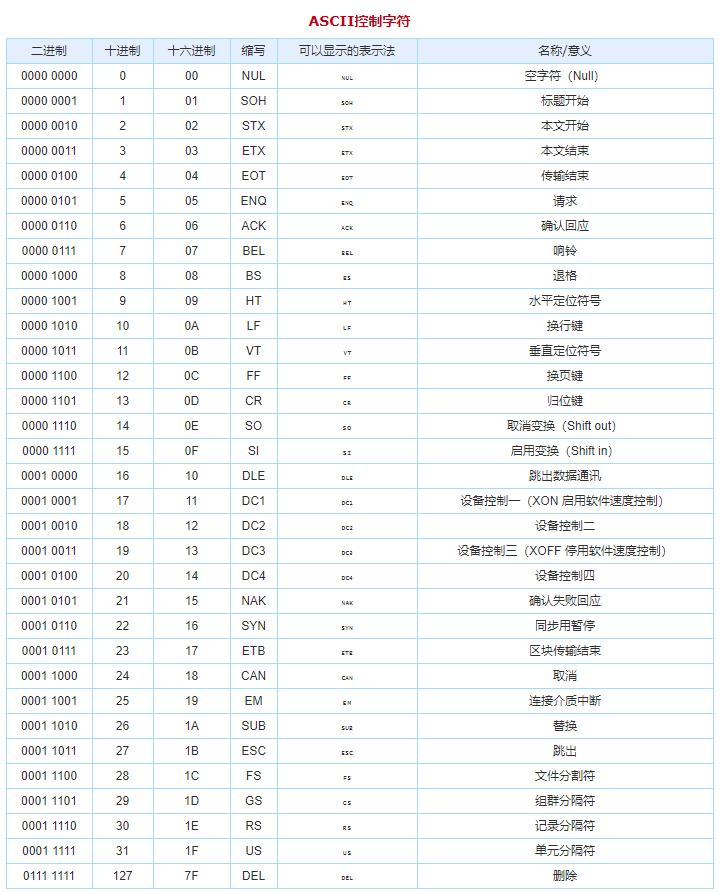
5.

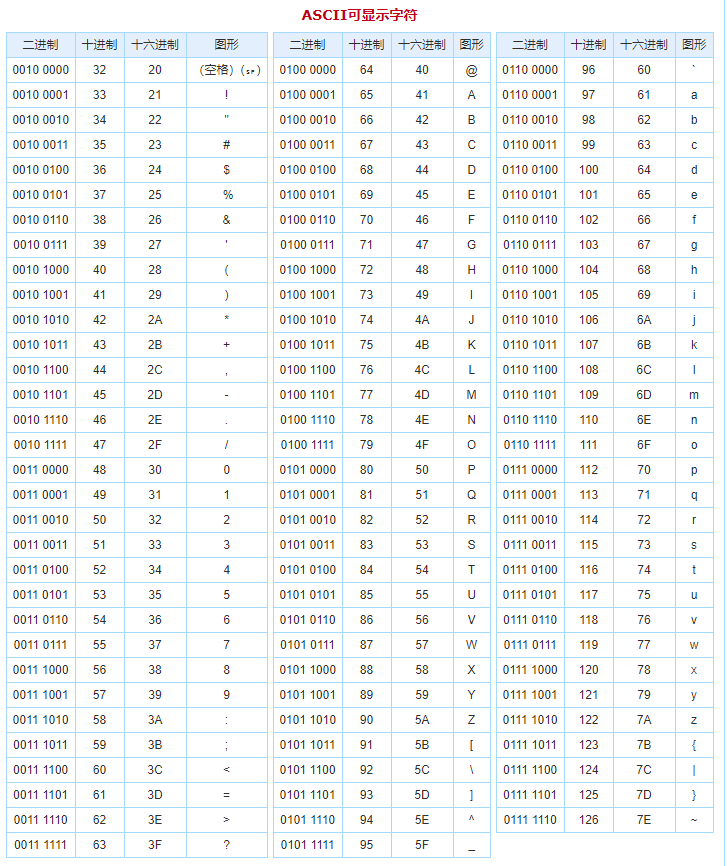


我的解决方案：



我觉得本人的解决方案能够更加深入的了解ASCII码

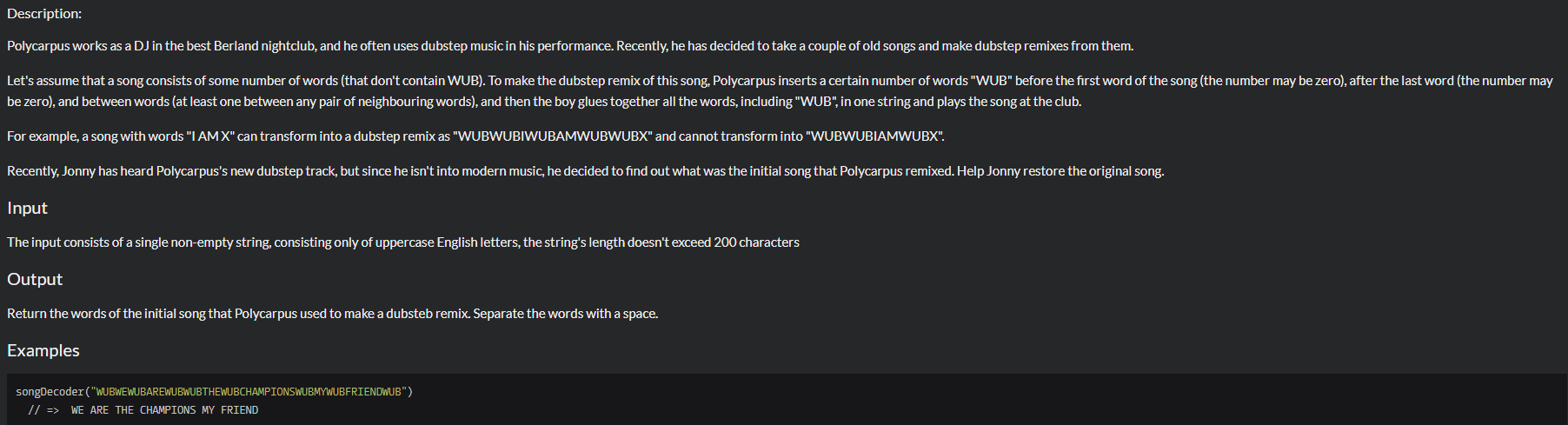




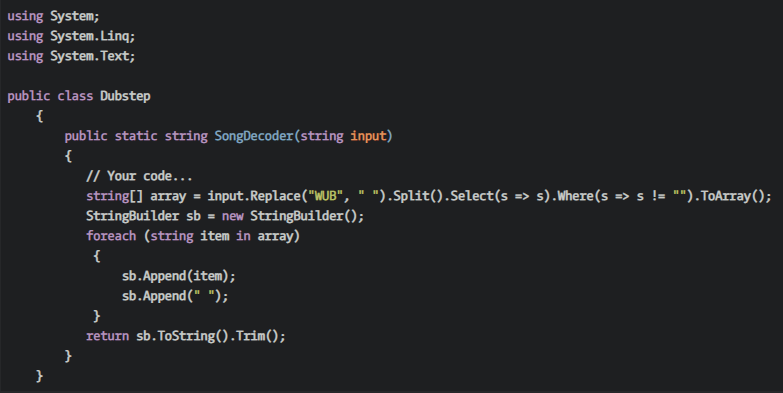
其中65到90为大写字母的ASCII码；

97到122为小写字母的ASCII码。

6.



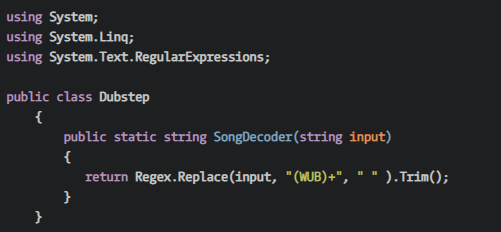
我的解決方案：



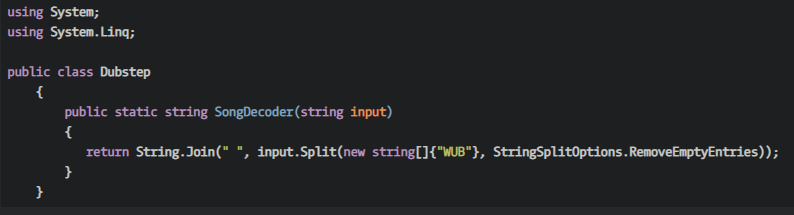
我通過Select()和Where()將各個單詞提取出來組成數組。

在將數組中的單詞構建

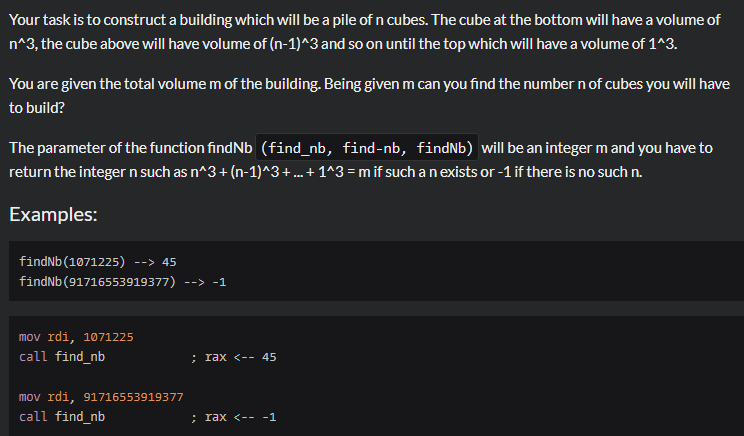
网络上的解决方案：



通过Regex的正则WUB+来替换多个WUB为空格。十分高效

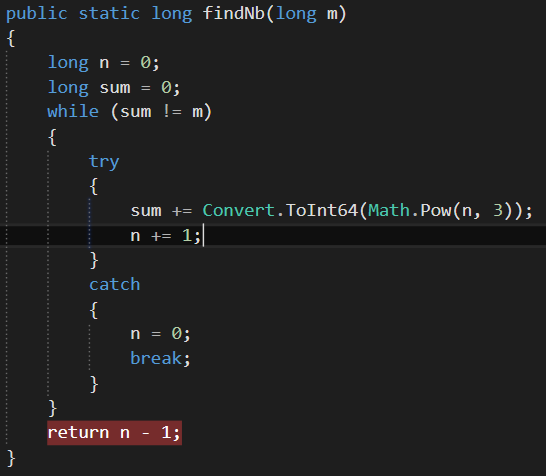


这个方法也很好理解，先根据WUB将字符串分割开来，在以” ”进行拼接.

7．

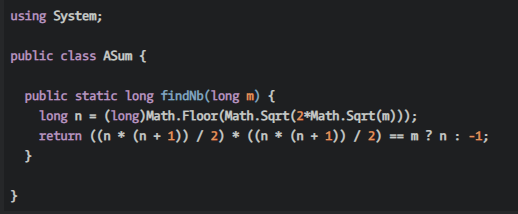
就是一个类似体积累和的公式。

我的解决方案：

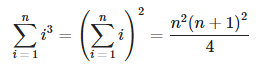


这个问题的难点在于如何在程式计算较长数据时能够不发生超时。

在模板中看到一个有趣的代码如下：

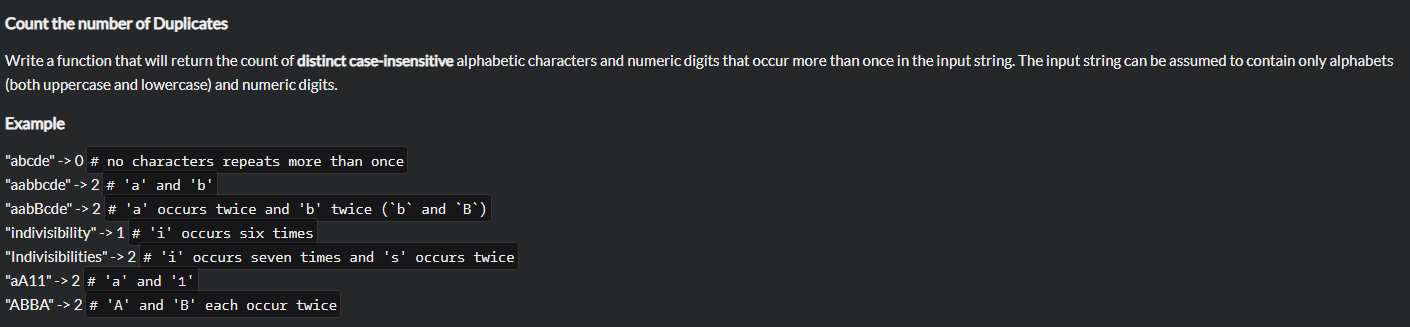


乍一看这个代码跟原问题没有什么关系，实际上是采用了数学中的体积累和公式：

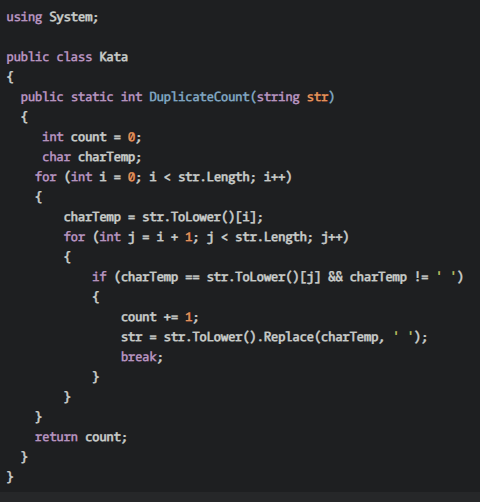


链接如下：

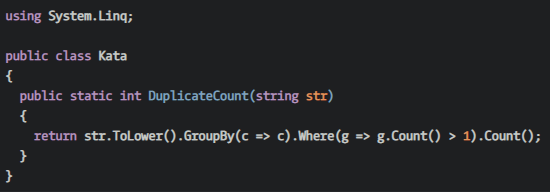
<https://proofwiki.org/wiki/Sum_of_Sequence_of_Cubes>



我的解决方案：



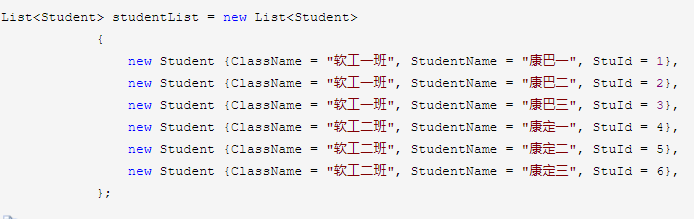
网上的解决方案：



这个方法直接使用了C#中的查询表达式GroupBy来得到结果

C#中实现Ienumerate<T>接口的类提供了很多扩展的方法，其中select，where等为常见的，且几乎和sql语法类似。

其中关于groupby方法的理解：（新建的一个Student类）



假设两个班里的学生总共有六名，现在根据班级分组：



Ienumberable代表返回结果可悲foreach便利，其中Igrouping<string,student>中应该是string代表的是一个key，即classname，那么key对应的应该就是一个student的集合。

IGrouping的key是作为自己的属性来存储，Telement则实现了Ienumerate<TElement>，所以调用foreach遍历Igrouping的时候返回的即是student集合了。

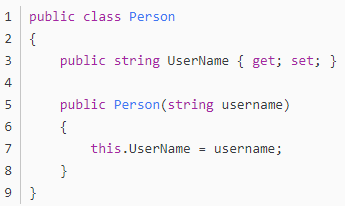
在groupby等分组的查询方法中通常会有相等比较器（IEqualityComparer）的参数，这里我们来了解一下什么是相等比较器：

在数组和集合的使用中，我们通常要在添加数据之前确定该数组或集合是否已经有相同的数据，以保持数组或集合中的数据的唯一。对于普通类型的数组来说，我们只需要使用扩展方法Contains就可以判断是否包含某个元素。



但是上面只是普通类型的判断，如果数组中存储的是一个对象（也就是集合），而不是普通类型的数据，那直接使用Contains方法就无法判断了

有一个对象Person：



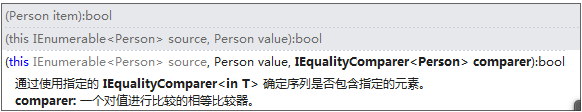
创建一个Person对象的集合，并添加多个该对象：



以上程序得到的结果是false，因为使用集合的Contains方法是对两个对象进行比较，这两个对象虽然UsersName相同，但并不是同一个对象。

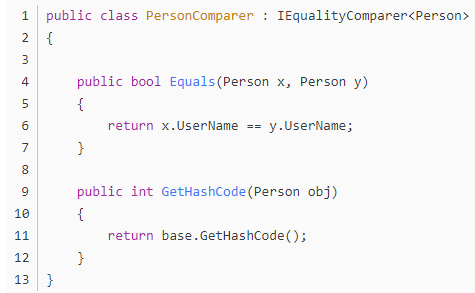
那么要如何检测出集合中有了一个名为“石昊”的Person呢？

我们看到Contains方法并不是一个只有一个参数的方法，它还有另外两个重载的方法：

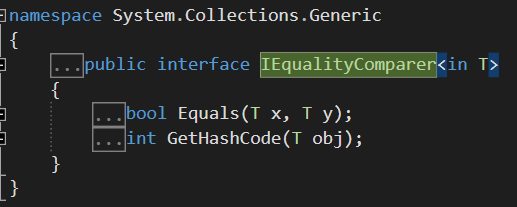


在最后一个重载方法中，我们可以看到，它需要一个相等比较器，其类型是IequalityComparer<Tsource>类型，其中Tsource就是进行比较的数据类型。

这个比较器是一个对象，我们可以针对Person类型创建一个比较器对象的类型：



相等比较器接口：



这个比较器类型需要实现IequalityComparer<Tsource>接口，在Equals方法中对两个对象的UserName进行比较。

这样，我们再来把list中的对象与p6对象进行比较：



此时就能得到TRUE的结果。

同样还有自定义的比较器：

在C#中，要比较两个数组，可以调用System.Array.Sort()方法，list也有Sort()方法。（但是只能用于系统的集合）

有两种方式可以为自己的类提供排序；

1. 类实现接口IComparable
2. 创建比较器类，该类实现接口IComparer

第二种方法的优点是，你可以在不修改已有的类，创建自己的比较器类，而且能实现比第一种方法更加灵活的方式；

namespace AnimalCompare

{

class Program

{

public class Animal : Icomparable // 这里是类实现接口的方式

{

private string name;

private int age;

public string Name

{

get { return name; }

set { name = value; }

}

public int Age

{

get { return age; }

set { age = value; }

}

public Animal(string name, int age)

{

this.name = name;

this.age = age;

}

public int CompareTo(Object obj) // 新增CompareTo方法来实现自定义类的CompareTo方法

{

Animal other = obj as Animal;

int result = name.CompareTo(other.name);

if (result == 0)

{

result = age.CompareTo(other.age);

}

return result;

} // 这里实现的是当name相同时在去比较age的大小

public override string ToString()

{

return string.Format("{0},{1}", name, age);

}

}

public class AnimalComparer : Icomparer // 这里是直接创建新的比较器类

{

public int Compare(object lhs, object rhs) // 需创建Compare方法

{

Animal aniLhs = lhs as Animal;

Animal aniRhs = rhs as Animal;

int result = aniLhs.Age.CompareTo(aniRhs.Age);

if (result == 0)

{

result = aniLhs.Name.CompareTo(aniRhs.Name);

}

return result;

}

} // 这里实现的是当age相同时在比较name的大小

static void Main(string[] args)

{

Animal[] animals = new Animal[]{

new Animal("ElephantBig",5),

new Animal("ElephantBig",3),

new Animal("ElephantNano",1)

};

foreach (Animal animal in animals)

{

Console.WriteLine(animal);

}

Console.WriteLine("/nAfter soring...");

Array.Sort(animals);

foreach (Animal animal in animals)

{

Console.WriteLine(animal);

}

Console.WriteLine("/nUsing IComparer...");

Array.Sort(animals, new AnimalComparer());

foreach (Animal animal in animals)

{

Console.WriteLine(animal);

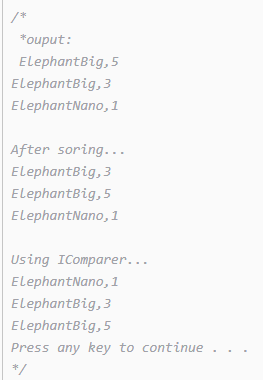
}

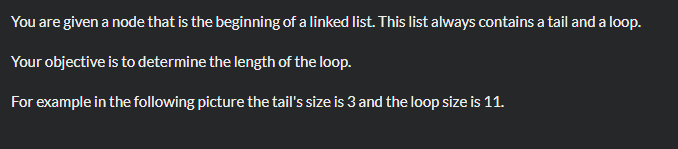
}

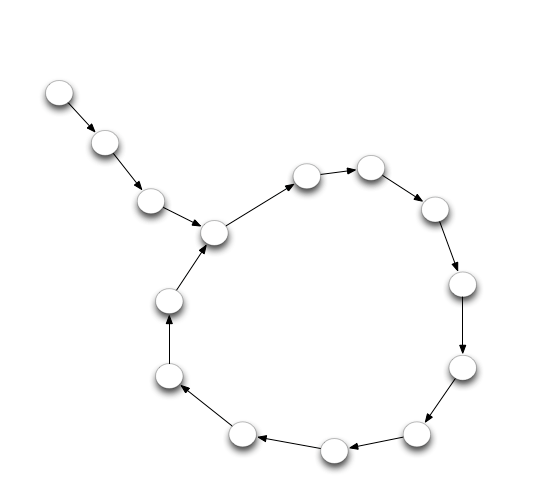
}

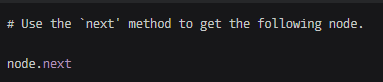
}

程式执行结果如下：



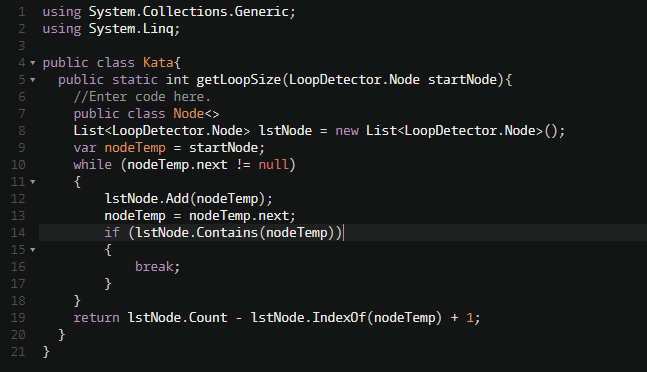




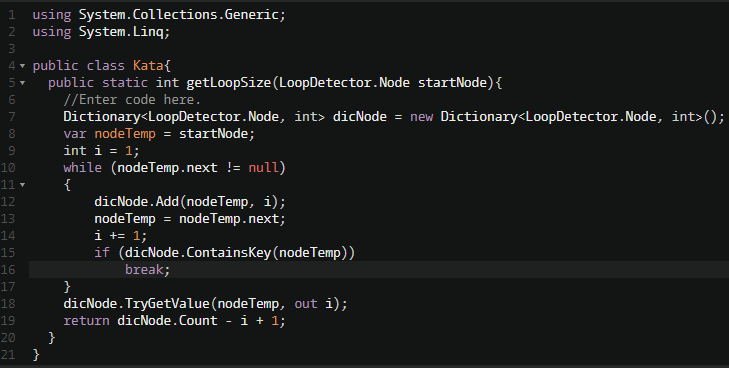


示例代码中的LoopDetector.Node为作者自定义的Node节点，只需要知道其next方法即可

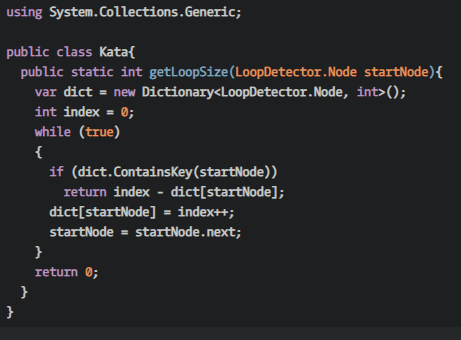
下图为我写的第一种方法，但是因为list的查询效率十分低，所以contains方法的运行效率就不会很高，所以当测试中循环的节点比较多的话，会发生执行timeout的情况。



所以采用Dictionary的集合来实现这里的contains逻辑，结果测试通过



网上的解决方案也是类似的：



还有一种解决方案如下：



这个方法是基于龟兔算法（tortoise and the hare algorithm）即Floyd判圈算法，是一个可以在有限状态机，迭代函数或者链表上判断是否存在环，已经判断环的起点和长度的算法。

1. 如何判断有环？

龟兔解法的基本思想可以用跑步的例子来解释，如果两个人同时出发，如果赛道有环，那么快的一方必定会追上慢的一方。即快的一方比慢的一方至少多跑一圈，即多跑的圈数是圈的长度的倍数。

如上述代码的第一个while，fast应该始终在slow前面，如果两个节点相同的话，那么只能说明这两个节点在一个环上。否则，如果到最后都没有相遇（相等），即没有环。

1. 求环的长度

相遇的时候，一定在环上了，然后两个人只要在环上再跑，再次相遇的时候，跑的快的就会比慢的多跑一圈，即环的长度。

也就是上图代码中的第二个while循环（该代码中只有fast节点在跑）

1. 如何确定环的起点

这也是Floyd解法的第二部分。方法是将其一个指针移到链表起点，两者同时移动，每次移动一步，那么两个相遇的地方就是环的起点。

C#实现十六进制

**任何数据在计算机内部都是以二进制保存的，所以进制与数据的存储无关，只与输入输出有关。。所以，对于进制转换，我们只关心字符串中的结果**。

int d = 10;

十进制转二进制字符串：

Console.Writeline(Convert.ToString(d, 2));

输出：1010

十进制转十六进制字符串

Console.Writeline(Convert.ToString(d, 16));

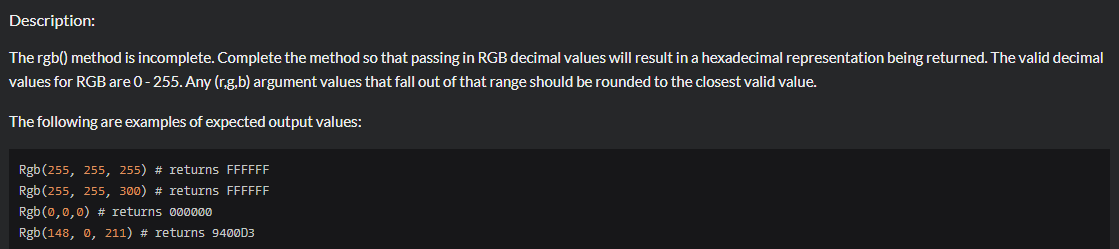
输出：a

二进制字符串转十进制

String bin = “1010”；

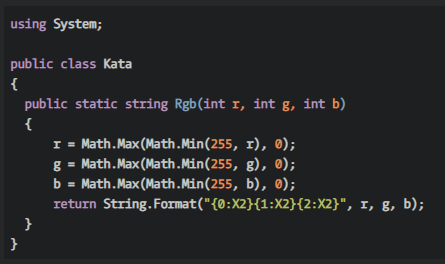
Console.Writeline(Convert.ToInt32(bin, 2));

输出：10



输入0-255之间的数字将其装换为两位的十六进制数。

网上解决方案：



如上图解决方案，直接将得到的int数字以X2（十六进制的两位）进行保存。