八皇后问题：

```
using System;
class Queen{
const int SIZE = 8;//皇后数
public static void Main()
{
int[] Queen = new int [SIZE];//每行皇后的位置
int y,x,i,j,d,t=0;
y = 0;
Queen[0] = -1;
while( true )
{
for (x=Queen[y]+1; x<SIZE; x++)
{
for (i=0;i<y;i++)
{
j = Queen[i];
d = y-i;
//检查新皇后是否与以前的皇后能相互攻击
if ((j==x)||(j==x-d)||(j==x+d))
break;
}
if (i>=y)
break;//不攻击
}
if (x == SIZE) //没有合适的位置
{
if (0==y)
{
//回朔到了第一行
Console.WriteLine("Done");
break; //结束
}
//回朔
Queen[y]=-1;
y--;
}
else
{
Queen[y]=x;//确定皇后的位置
y++;//下一个皇后
if (y<SIZE)
Queen[y]=-1;
else
```

```
{
//所有的皇后都排完了，输出
Console.WriteLine("\n" + ++t +':');
for(i=0;i<SIZE;i++)
{
for (j=0;j<SIZE;j++)
if(Queen[i] == j)
Console.Write('Q');
else
Console.Write('.');
Console.WriteLine();
}
y = SIZE -1;//回朔
}
}
}
}
}
```
八皇后有解 92 个。

C#算法大全

1.冒泡排序

```
using System；
namespace BubbleSorter
{
public class BubbleSorter
{
public void Sort（int [] list）
{
int i,j,temp；
bool done=false；
j=1；
while（（j＜list.Length）&&（!done））
{
done=true；
for（i=0；i＜list.Length-j；i++）
{
if（list[i]＞list[i+1]）
{
done=false；
temp=list[i]；
list[i]=list[i+1]；
list[i+1]=temp；
```

```
}
}
j++;
}

}
}
public class MainClass
{
public static void Main（）
{
int[] iArrary=new int[]{1,5,13,6,10,55,99,2,87,12,34,75,33,47}；
BubbleSorter sh=new BubbleSorter（）；
sh.Sort（iArrary）；
for（int m=0；m＜iArrary.Length；m++）
Console.Write（"{0} ",iArrary[m]）；
Console.WriteLine（）；
}
}
}
```

2.选择排序

```
using System；

namespace SelectionSorter
{
public class SelectionSorter
{
private int min；
public void Sort（int [] list）
{
for（int i=0；i＜list.Length-1；i++）
{
min=i；
for（int j=i+1；j＜list.Length；j++）
{
if（list[j]＜list[min]）
min=j；
}
int t=list[min]；
list[min]=list[i]；
list[i]=t；
}
```

```
}
}
public class MainClass
{
public static void Main（）
{
int[] iArrary=new int[]{1,5,3,6,10,55,9,2,87,12,34,75,33,47}；
SelectionSorter ss=new SelectionSorter（）；
ss.Sort（iArrary）；
for（int m=0；m＜iArrary.Length；m++）
Console.Write（"{0} ",iArrary[m]）；
Console.WriteLine（）；

}
}
}
```

3.插入排序

```
using System；

namespace InsertionSorter
{
public class InsertionSorter
{
public void Sort（int [] list）
{
for（int i=1；i＜list.Length；i++）
{
int t=list[i]；
int j=i；
while（（j＞0）&&（list[j-1]＞t））
{
list[j]=list[j-1]；
--j；
}
list[j]=t；
}

}
}
public class MainClass
{
```

```
public static void Main（）
{
int[] iArrary=new int[]{1,13,3,6,10,55,98,2,87,12,34,75,33,47}；
InsertionSorter ii=new InsertionSorter（）；
ii.Sort（iArrary）；
for（int m=0；m＜iArrary.Length；m++）
Console.Write（"{0}",iArrary[m]）；
Console.WriteLine（）；
}
}
}
```

4.希尔排序

```
using System；

namespace ShellSorter
{
public class ShellSorter
{
public void Sort（int [] list）
{
int inc；
for（inc=1；inc＜=list.Length/9；inc=3*inc+1）；
for（；inc＞0；inc/=3）
{
for（int i=inc+1；i＜=list.Length；i+=inc）
{
int t=list[i-1]；
int j=i；
while（（j＞inc）&&（list[j-inc-1]＞t））
{
list[j-1]=list[j-inc-1]；
j-=inc；
}
list[j-1]=t；
}
}
}
}
public class MainClass
{
public static void Main（）
```

```
{
int[] iArrary=new int[]{1,5,13,6,10,55,99,2,87,12,34,75,33,47}；
ShellSorter sh=new ShellSorter（）；
sh.Sort（iArrary）；
for（int m=0；m＜iArrary.Length；m++）
Console.Write（"{0} ",iArrary[m]）；
Console.WriteLine（）；
}
}
}
```

C# 2.0 新特性探究之模拟泛型和内置算法

在 C#2.0 中，匿名方法、IEnumerable 接口和匿名方法的合作，使很多的编程任务变得非常的简单，而且写出来的程序非常的优美。

比如，我们可以写出如下的代码：

```
List＜Book＞ thelib = Library.getbooks();

List＜Book＞ found = thelib.FindAll(delegate(Book curbook)
{
    if (curbook.isbn.StartsWith("..."))
        return true;
        return false;
});

foreach (Book b in found)
Console.WriteLine(b.isbn);
```

这段程序非常简单的展示给我们需要查找的信息，代码也非常的直接易懂。内置的数据结构给了我们强大的算法支持，不过，能不能够为自定义的类定义类似的算法呢？

比如，如果我有一个自定义的 Library 类并没有使用 List＜Book＞存储数据，而是使用某种自定义的数据结构，我能不能也让用户使用类似的语法，忽略存储细节的使用匿名委托来实现特定的算法呢？

答案当然是肯定的，而且在 C#中实现这样的功能是非常的简单。

首先让我们看看 FindAll 中用到的匿名委托的原型

```
public delegate bool Predicate＜T＞(T obj);
```

很明显的，上面的代码等于注册了一个搜索的回调，而在 List　　定义了某种遍历的机制，从而实现了一个漂亮的算法结构 Closure。

看到了这些，我们就可以定义自己的算法结构了，首先，我定义了一个如下的类

```
public class MyVec＜T＞
{
    public static MyVec＜T＞ operator +(MyVec＜T＞ a, T b)
    {
       a._list.Add(b);
       return a;
    }
    public override string ToString()
    {
       StringBuilder builder = new StringBuilder();
       foreach (T a in _list)
       {
          builder.Append(a.ToString());
          builder.Append(",");
       }
       string ret = builder.Remove(builder.Length - 1, 1).ToString();
       return ret;
    }

    public MyVec＜T＜＞findAll(Predicate＜T＞ act)
    {
       MyVec＜T：＞t2 = new MyVec＜T＞();
       foreach(T i in _list)
       {
          if (act(i))
          t2._list.Add(i);
       }
       return t2;
    }

    // this is the inner object
    private List＜T＞ _list = new List＜T＞();
}
```

这个类中包含了一个的 List＜T＞结构，主要是为了证实我们的想法是否可行，事实上，任何一个可以支持 foreach 遍历的结构都可以作为内置的数据存储对象，我们会在后面的例子中给出一个更加复杂的实现。

下面是用于测试这个实验类的代码：

```
static void Main(string[] args)
{
    MyVec＜int＞ a = new MyVec＜int＞();
    a += 12;
    a += 15;
    a += 32;
    MyVec＜int＞ b = a.findAll(delegate(int x)
    {
       if (x ＜ 20) return true; return false;
    }
);

    Console.WriteLine("vection original");
    Console.WriteLine(a.ToString());
    Console.WriteLine("vection found");
    Console.WriteLine(b.ToString());
    Console.ReadLine();
}
```

编译，执行，程序输出：

vection original

12,15,32

vection found

32

和我们预期的完全相同。很明显的，List 的算法与我们预期的基本相同。

Predicate＜T＞仅仅是为了仿照系统的实现而采用的一个委托，事实上可以使用自己定义的任何委托作为回调的函数体。

通过使用 IEnumberable 接口，可以实现对任意结构的遍历，从而对任何数据结构定义强大的算法支持。

对 C#泛型中的 new()约束的一点思考

对于 new()约束，大家可能有一个误解，以为使用了 new 约束之后，在创建对象时与非泛型的版本是一致的：