设计模式之策略模式

2016年5月9日　　23:49

策略模式定义了一系列的算法，并将每一个算法封装起来，而且使它们还可以相互替换。策略模式让算法独立于使用它的客户而独立变化。

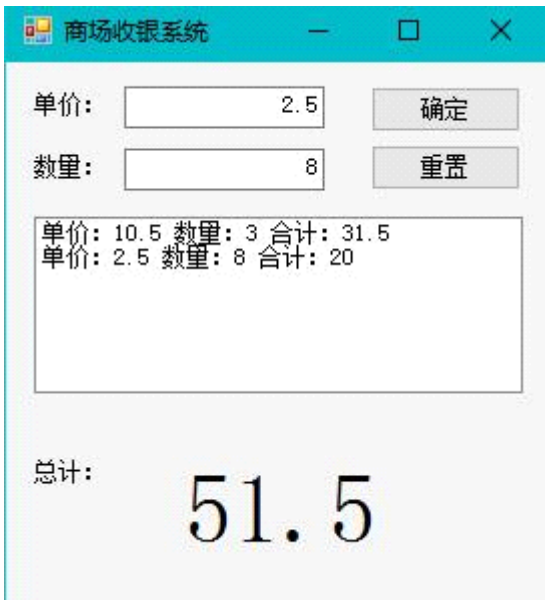
举个例子：

出场人物：小菜（菜鸟级程序员），大鸟（骨灰级程序员）

要求：做一个商场收银软件，营业员根据客户所购买的商品的单价和数量，向客户收费。

商场收银系统v1.0运行截图以及关键代码

```
 public partial class Form1 : Form    {
public Form1()
        {
            InitializeComponent();
        }
//声明一个double变量total来计算总计
        double total = 0.0d;
        private void btnOk_Click(object sender, EventArgs e)
        {
            //声明一个double变量totalPrices来计算每个商品的单价（txtPrice）*数量(txtNum)后的合计
double totalPrices=Convert.ToDouble(txtPrice.Text) * Convert.ToDouble(txtNum.Text);
            total = total + totalPrices;//将每个商品合计计入总计
lbxList.Items.Add("单价："+txtPrice.Text+" 数量："+txtNum.Text+" 合计："+totalPrices.ToString());//在列表框中显示信息
lblResult.Text = total.ToString();//在lblResult标签上显示总计数
}
    }
}
```



现在问题来了：

大鸟：昨天妈妈节商场的所有商品打八折，那么上面的代码需要怎样修改呢？

小菜：上面的totalPrices乘以0.8不就可以了，没什么大不了的。

大鸟：但是过了妈妈节商场老板决定恢复原来的价格，那你是不是又要再改回来？不过再过不久也就六一儿童节了，商场要是打算打七折呢，你怎么办？

小菜：这样呐，我就加一个combox，里面给出打折选项就可以啦。

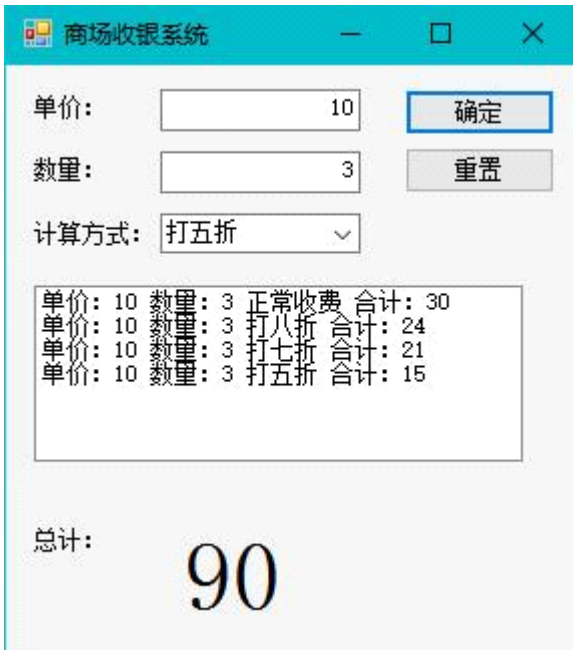
大鸟笑而不语。

商场收银系统v1.1运行截图以及关键代码

```
double total = 0.0d;

        private void Form1_Load(object sender, EventArgs e)
        {
            cbxType.Items.AddRange(new object[] {"正常收费","打八折","打七折","打五
折"});//在combox中添加下拉选项
            cbxType.SelectedIndex = 0;
        }
private void btnOk_Click(object sender, EventArgs e)
        {
            double totalPrices=0d;
            switch(cbxType.SelectedIndex)
            {
                case 0:
                    totalPrices = Convert.ToDouble(txtPrice.Text) *
Convert.ToDouble(txtNum.Text);
                    break;
                case 1:
                    totalPrices = Convert.ToDouble(txtPrice.Text) *
Convert.ToDouble(txtNum.Text) * 0.8;
                    break;
                case 2:
                    totalPrices = Convert.ToDouble(txtPrice.Text) *
Convert.ToDouble(txtNum.Text) * 0.7;
                    break;
                case 3:
                    totalPrices = Convert.ToDouble(txtPrice.Text) *
Convert.ToDouble(txtNum.Text) * 0.5;
                    break;

}
            total = total + totalPrices;
            lbxList.Items.Add("单价：" + txtPrice.Text + " 数量：" + txtNum.Text
                + " "+cbxType.SelectedItem+ " 合计：" + totalPrices.ToString());
            lblResult.Text = total.ToString();
}
```



小菜：这样就可以了吧。

大鸟：比之前灵活了些，但是问题也不少：首先Convert.ToDouble()在这里就写了8遍，另外商场决定加大活动力度，满300返100这样的促销算法，你觉得该怎么办？

小菜：那意思就是满300返100，700的话就返200了？写函数就可以了吧？

大鸟：看来之前的简单工厂模式是白学了。

小菜：哦哦哦，这样子呐。那就先写一个父类，在继承它实现多个打折和返利的子类，使用多态，是这样子吧？

大鸟：这样你准备写多少个子类？

小菜：根据商店需求呀，比如9折，8折，6折，满300返100，满400返180.....要多少写多少O(∩_∩)O~~

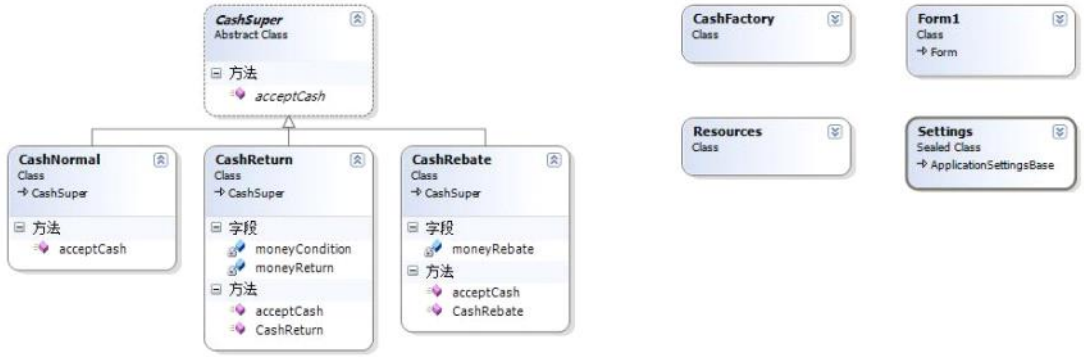
大鸟：小菜没有认真考虑这些促销活动之间的相同点和不同点呐

小菜：(⊙o⊙)哦，好像是有相同的地方，可以分为打折促销，返利促销，正常收费三个模式。这样可以写一个**父类，包含抽象方法：收钱()，然后写三个子类（正常收费，返利收费，打折收费）**分别实现这个收钱的函数，然后使用简单工厂模式，创建**一个收费工厂类**，在里面进行收费的选择，创建不同的收费实例。

大鸟：**面向对象的编程，并不是类越多越好，类的划分是为了封装，但分类的基础是抽象，具有相同属性和功能的对象的抽象集合才是类。**

商场收银系统v1.1运行截图以及关键代码

代码结构图：



现金收费子类：

```
using System;
using System.Collections.Generic;
using System.Text;

namespace 商场管理软件
{
//现金收取父类
abstract class CashSuper
{
    //抽象方法：收取现金，参数为原价，返回为当前价
    public abstract double acceptCash(double money);
}

}
```

正常收费子类：

```
using System;
using System.Collections.Generic;
using System.Text;

namespace 商场管理软件
{
    //正常收费，继承CashSuper
    class CashNormal : CashSuper
    {
```

```
        public override double acceptCash(double money)
        {
            return money;
        }
    }
}
```

打折收费子类：

```
using System;
using System.Collections.Generic;
using System.Text;

namespace 商场管理软件
{
    //打折收费，继承CashSuper
    class CashRebate : CashSuper
    {
        private double moneyRebate = 1d;
        //初始化时，必需要输入折扣率,如八折，就是0.8
        public CashRebate(string moneyRebate)
        {
            this.moneyRebate = double.Parse(moneyRebate);
        }
public override double acceptCash(double money)
        {
            return money * moneyRebate;
        }
    }
}
```

返利收费子类：

```
using System;
using System.Collections.Generic;
using System.Text;

namespace 商场管理软件
{
    //返利收费，继承CashSuper
    class CashReturn : CashSuper
    {
        private double moneyCondition = 0.0d;
        private double moneyReturn = 0.0d;
        //初始化时必须要输入返利条件和返利值，比如满300返100，则moneyCondition为300，
moneyReturn为100
        public CashReturn(string moneyCondition, string moneyReturn)
        {
            this.moneyCondition = double.Parse(moneyCondition);
            this.moneyReturn = double.Parse(moneyReturn);
        }
public override double acceptCash(double money)
        {
            double result = money;
            //若大于返利条件，则需要减去返利值
            if (money >= moneyCondition)
                result = money - Math.Floor(money / moneyCondition) * moneyReturn;
return result;
        }
    }
}
```

现金收费工厂类：

```
using System;
using System.Collections.Generic;
using System.Text;

namespace 商场管理软件
{
//现金收取工厂
class CashFactory
{
    //根据条件返回相应的对象
    public static CashSuper createCashAccept(string type)
    {
        CashSuper cs = null;
        switch (type)
        {
            case "正常收费":
                cs = new CashNormal();
                break;
            case "满300返100":
                CashReturn cr1 = new CashReturn("300", "100");
                cs = cr1;
                break;
            case "打8折":
                CashRebate cr2 = new CashRebate("0.8");
                cs = cr2;
                break;
        }
        return cs;
    }
}
}
```

客户端程序主要部分：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace 商场管理软件
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
//客户端窗体程序（主要部分）
        double total = 0.0d;
        private void btnOk_Click(object sender, EventArgs e)
        {
            //利用简单工厂模式根据下拉选择框，生成相应的对象
            CashSuper csuper =
CashFactory.createCashAccept(cbxType.SelectedItem.ToString());
            double totalPrices = 0d;
            //通过多态，可以得到收取费用的结果
            totalPrices = csuper.acceptCash(Convert.ToDouble(txtPrice.Text) *
Convert.ToDouble(txtNum.Text));
            total = total + totalPrices;
            lbxList.Items.Add("单价：" + txtPrice.Text + " 数量：" + txtNum.Text + "
"
                + cbxType.SelectedItem + " 合计：" + totalPrices.ToString());
            lblResult.Text = total.ToString();
        }
private void btnClear_Click(object sender, EventArgs e)
        {
            total = 0d;
```

```
            txtPrice.Text = "0.00";
            txtNum.Text = "0";
            lbxList.Items.Clear();
            lblResult.Text = "0.00";
        }
    }
}
```

小菜：大鸟，这样不管怎样修改，我都可以简单处理了。

大鸟：那我要实现打五折和满500返200的返利活动，你怎么修改？

小菜：在现金工厂中添加打五折和满500返200的case语句，然后在下拉选择框中添加两个选项就可以了。

大鸟：现金工厂？应该是收费对象生成工厂才准确。但是如果促销修改为满100积分加10，当积分达到一定时候就可以领取奖品怎么做？

小菜：有了工厂，何难？添加一个积分算法，构造方法里面有两个参数：条件和返点，继承CashSuper，然后在现金工厂，哦，不对，是收费对象生成工厂里面增加满100积分加10的分支条件，然后在下拉选择框里面添加这个选项就可以了。

大鸟：但是在以后的学习中我们会发现这样的设计模式并不是最好的选择，因为每次维护或者扩展都要修改工厂这个类，以至于代码需要重新编译部署，这样的处理很糟糕，自己去研究一下看看那个设计模式可以用在其中。

小菜（陷入沉思.....）

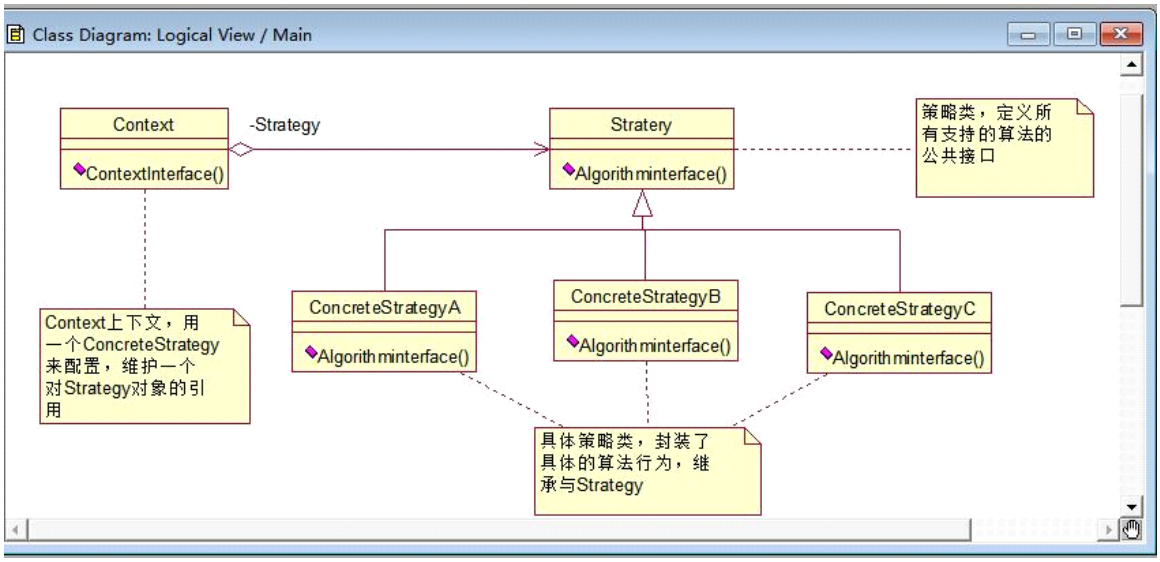
次日

小菜：大鸟我知道该使用哪种设计模式了，选择策略模式就很好地解决了这个问题。

策略模式(Strategy):它定义了算法家族，分别封装起来，让它们之间可以相互替换，此模式让算法的变化，不会影响到使用算法的客户。

大鸟：你说说看。

小菜：这里是我画的策略模式结构图




```
using System;
using System.Collections.Generic;
using System.Text;

namespace 策略模式
{
    class Program
    {
        static void Main(string[] args)
        {
            Context context;
context = new Context(new ConcreteStrategyA());
            context.ContextInterface();
context = new Context(new ConcreteStrategyB());
            context.ContextInterface();
context = new Context(new ConcreteStrategyC());
            context.ContextInterface();
Console.Read();
        }
    }
//抽象算法类
    abstract class Strategy
    {
        //算法方法
        public abstract void AlgorithmInterface();
    }
    //具体算法A
```

```
    class ConcreteStrategyA : Strategy
    {
        //算法A实现方法
        public override void AlgorithmInterface()
        {
            Console.WriteLine("算法A实现");
        }
    }
    //具体算法B
    class ConcreteStrategyB : Strategy
    {
        //算法B实现方法
        public override void AlgorithmInterface()
        {
            Console.WriteLine("算法B实现");
        }
    }
    //具体算法C
    class ConcreteStrategyC : Strategy
    {
        //算法C实现方法
        public override void AlgorithmInterface()
        {
            Console.WriteLine("算法C实现");
        }
    }
    //上下文
    class Context
    {
        Strategy strategy;
public Context(Strategy strategy)
        {
            this.strategy = strategy;
        }
        //上下文接口
        public void ContextInterface()
        {
            strategy.AlgorithmInterface();
        }
    }
}
```

来自 <http://i.cnblogs.com/EditPosts.aspx?postid=5475908>

完全背包问题实例

2016年5月9日　　23:49

题目描述

零崎有很多朋友，其中有一个叫做lfj的接盘侠。

lfj是一个手残，他和零崎一起玩网游的时候不好好打本，天天看拍卖行，没过多久，就成为了一个出色的商人。时间一长，虽然挣了不少钱，却没时间练级了。

作为lfj的友人，零崎实在看不下去，于是他决定帮lfj一把。当然了，零崎肯定不会自己动手，活还得你们来干。

lfj可以提供给你们拍卖行所有能买到物品的价格和利润，由于游戏产出不限，所以可以假定只要有钱，即使是同一种东西，多少个也都能买到手。lfj还会告诉你他初始的成本。虽然零崎想让你们给出一次交易中利润最大的购买方案，但是lfj觉得只要知道最大利润就可以了。

输入

每组数据第一行为两个整数P和N，表示本金和拍卖行物品种类数。

接下来N行，每行两个数据pi,ci代表第i类物品的利润和购买价格。

1<=P<=20000,1<=N<=300,1<=c,p<=200

输出

对于每组数据，输出一行，为能获得的最大利润

输入样例

```
3 1
2 1
2 3
1 1
1 2
2 1
```

输出样例

```
6
4
```

Hint

使用if直接比较不要调用max（）以防超时

完全背包问题：

完全背包和0-1背包的不同之处：完全背包的物品不再是只有一件而是有无数件，所以对于某一件物品也不再是拿（1）不拿（0）。而是变为了拿0件，1件，2件...k件,按照0-1背包问题的状态转移方程同样可以写出完全背包的状态转移方程：

f[i][v]=max{f[i-1][v-k*c[i]]+k*w[i]|0<=k*c[i]<=v}

分析上述的状态转移方程

这跟01背包问题一样有O(N*V)个状态需要求解，但求解每个状态的时间已经不是常数了，求解状态f[i][v]的时间是O(v/c[i])，总的复杂度是超过O(VN)的。

因此我们需要对改状态方程进行改进：

O(VN)的算法：

```
1 for (int i = 1; i <= N; i++)
2
3     for (int v = 0; v <= V; v++)
4
```

```
5         f[v] = max(f[v], f[v - c[i]] + w[i]);
```

或者f[i][v]=max(f[i-1][v],f[i][v-c[i]]+w[i])

可以发现和0-1背包不同的地方只是在于内部for循环的起止改变了顺序，为什么这样可以实现完全背包的要求呢？

首先想想为什么P01中要按照v=V..0的逆序来循环。这是因为要保证第i次循环中的状态f[i][v]是由状态f[i-1][v-c[i]]递推而来。换句话说，这正是为了保证每件物品只选一次，保证在考虑“选入第i件物品”这件策略时，依据的是一个绝无已经选入第i件物品的子结果f[i-1][v-c[i]]。而现在完全背包的特点恰是每种物品可选无限件，所以在考虑“加选一件第i种物品”这种策略时，却正需要一个可能已选入第i种物品的子结果f[i][v-c[i]]，所以就可以并且必须采用v= 0..V的顺序循环。这就是这个简单的程序为何成立的道理。

因此可以得到完全背包的代码实现：

```
1 void CompletePack(int cost , int weight)
2 {
3     for (int i = weight ; i <= W ; ++ i)
4         f[i] = max(f[i],f[i-weight]+cost) ;
5 }
```

下面给出本题的代码实现：

```
 1 #include <bits/stdc++.h>
 2 long long f[20010];
 3 long long c[310];
 4 long long v[310];
 5 using namespace std;
 6
 7 int main()
 8 {
 9     int V,k;
10     while(~scanf("%d%d",&V,&k))
11     {
12         memset(f,0,sizeof(f));
13         memset(c,0,sizeof(c));
14         memset(v,0,sizeof(v));
15         for(int i=1; i<=k; i++)
16             scanf("%lld%lld",&v[i],&c[i]);
17         for(int i=1; i<=k; i++)
18         {
19             for(int j=c[i]; j<=V; j++)
20             {
21                 if(f[j-c[i]]+v[i]>=f[j])
22                     f[j]=f[j-c[i]]+v[i];
23                 else
24                     f[j]=f[j];
25             }
26         }
27         printf("%lld\n",f[V]);
28     }
29 }
```

来自 <http://i.cnblogs.com/EditPosts.aspx?postid=4966335>

背包九讲

2016年5月9日　　23:50

dd大牛的《背包九讲》

P01：01背包问题

题目

有N件物品和一个容量为V的背包。第i件物品的费用是c[i]，价值是w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

基本思路

这是最基础的背包问题，特点是：每种物品仅有一件，可以选择放或不放。

用子问题定义状态：即f[i][v]表示前i件物品恰放入一个容量为v的背包可以获得的最大价值。则其状态转移方程便是：f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]}。

这个方程非常重要，基本上所有跟背包相关的问题的方程都是由它衍生出来的。所以有必要将它详细解释一下：“将前i件物品放入容量为v的背包中”这个子问题，若只考虑第i件物品的策略（放或不放），那么就可以转化为一个只牵扯前i-1件物品的问题。如果不放第i件物品，那么问题就转化为“前i-1件物品放入容量为v的背包中”；如果放第i件物品，那么问题就转化为“前i-1件物品放入剩下的容量为v-c[i]的背包中”，此时能获得的最大价值就是f [i-1][v-c[i]]再加上通过放入第i件物品获得的价值w[i]。

注意f[i][v]有意义当且仅当存在一个前i件物品的子集，其费用总和为v。所以按照这个方程递推完毕后，最终的答案并不一定是f[N] [V]，而是f[N][0..V]的最大值。如果将状态的定义中的“恰”字去掉，在转移方程中就要再加入一项f[i][v-1]，这样就可以保证f[N] [V]就是最后的答案。至于为什么这样就可以，由你自己来体会了。

优化空间复杂度

以上方法的时间和空间复杂度均为O(N*V)，其中时间复杂度基本已经不能再优化了，但空间复杂度却可以优化到O(V)。

先考虑上面讲的基本思路如何实现，肯定是有一个主循环i=1..N，每次算出来二维数组f[i][0..V]的所有值。那么，如果只用一个数组f [0..V]，能不能保证第i次循环结束后f[v]中表示的就是我们定义的状态f[i][v]呢？f[i][v]是由f[i-1][v]和f[i-1] [v-c[i]]两个子问题递推而来，能否保证在推f[i][v]时（也即在第i次主循环中推f[v]时）能够得到f[i-1][v]和f[i-1][v -c[i]]的值呢？事实上，这要求在每次主循环中我们以v=V..0的顺序推f[v]，这样才能保证推f[v]时f[v-c[i]]保存的是状态f[i -1][v-c[i]]的值。伪代码如下：

for i=1..N

for v=V..0

f[v]=max{f[v],f[v-c[i]]+w[i]};

其中的f[v]=max{f[v],f[v-c[i]]}一句恰就相当于我们的转移方程f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]}，因为现在的f[v-c[i]]就相当于原来的f[i-1][v-c[i]]。如果将v的循环顺序从上面的逆序改成顺序的话，那么则成了f[i][v]由f[i][v-c[i]]推知，与本题意不符，但它却是另一个重要的背包问题P02最简捷的解决方案，故学习只用一维数组解01背包问题是十分必要的。

总结

01背包问题是最基本的背包问题，它包含了背包问题中设计状态、方程的最基本思想，另外，别的类型的背包问题往往也可以转换成01背包问题求解。故一定要仔细体会上面基本思路的得出方法，状态转移方程的意义，以及最后怎样优化的空间复杂度。

P02: 完全背包问题

题目

有N种物品和一个容量为V的背包，每种物品都有无限件可用。第i种物品的费用是c[i]，价值是w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

基本思路

这个问题非常类似于01背包问题，所不同的是每种物品有无限件。也就是从每种物品的角度考虑，与它相关的策略已并非取或不取两种，而是有取0件、取1件、取2件……等很多种。如果仍然按照解01背包时的思路，令f[i][v]表示前i种物品恰放入一个容量为v的背包的最大权值。仍然可以按照每种物品不同的策略写出状态转移方程，像这样：f[i][v]=max{f[i-1][v-k*c[i]]+k*w[i]|0<=k*c[i]<= v}。这跟01背包问题一样有O(N*V)个状态需要求解，但求解每个状态的时间则不是常数了，求解状态f[i][v]的时间是O(v/c[i])，总的复杂度是超过O(VN)的。

将01背包问题的基本思路加以改进，得到了这样一个清晰的方法。这说明01背包问题的方程的确是很重要，可以推及其它类型的背包问题。但我们还是试图改进这个复杂度。

一个简单有效的优化

完全背包问题有一个很简单有效的优化，是这样的：若两件物品i、j满足c[i]<=c[j]且w[i]>=w[j]，则将物品j去掉，不用考虑。这个优化的正确性显然：任何情况下都可将价值小费用高得j换成物美价廉的i，得到至少不会更差的方案。对于随机生成的数据，这个方法往往会大大减少物品的件数，从而加快速度。然而这个并不能改善最坏情况的复杂度，因为有可能特别设计的数据可以一件物品也去不掉。

转化为01背包问题求解

既然01背包问题是最基本的背包问题，那么我们可以考虑把完全背包问题转化为01背包问题来解。最简单的想法是，考虑到第i种物品最多选V/c [i]件，于是可以把第i种物品转化为V/c[i]件费用及价值均不变的物品，然后求解这个01背包问题。这样完全没有改进基本思路的时间复杂度，但这毕竟给了我们将完全背包问题转化为01背包问题的思路：将一种物品拆成多件物品。

更高效的转化方法是：把第i种物品拆成费用为c[i]*2^k、价值为w[i]*2^k的若干件物品，其中k满足c[i]*2^k<V。这是二进制的思想，因为不管最优策略选几件第i种物品，总可以表示成若干

个2^k件物品的和。这样把每种物品拆成O(log(V/c[i]))件物品，是一个很大的改进。但我们有更优的O(VN)的算法。 * O(VN)的算法这个算法使用一维数组，先看伪代码： <pre class"example"> for i=1..N for v=0..V f[v]=max{f[v],f[v-c[i]]+w[i]};

你会发现，这个伪代码与P01的伪代码只有v的循环次序不同而已。为什么这样一改就可行呢？首先想想为什么P01中要按照v=V..0的逆序来循环。这是因为要保证第i次循环中的状态f[i][v]是由状态f[i-1][v-c[i]]递推而来。换句话说，这正是为了保证每件物品只选一次，保证在考虑“选入第i件物品”这件策略时，依据的是一个绝无已经选入第i件物品的子结果f[i-1][v-c[i]]。而现在完全背包的特点恰是每种物品可选无限件，所以在考虑“加选一件第i种物品”这种策略时，却正需要一个可能已选入第i种物品的子结果f[i][v-c[i]]，所以就可以并且必须采用v= 0..V的顺序循环。这就是这个简单的程序为何成立的道理。

这个算法也可以以另外的思路得出。例如，基本思路中的状态转移方程可以等价地变形成这种形式：f[i][v]=max{f[i-1][v],f[i][v-c[i]]+w[i]}，将这个方程用一维数组实现，便得到了上面的伪代码。

总结

完全背包问题也是一个相当基础的背包问题，它有两个状态转移方程，分别在“基本思路”以及“O(VN)的算法“的小节中给出。希望你能够对这两个状态转移方程都仔细地体会，不仅记住，也要弄明白它们是怎么得出来的，最好能够自己想一种得到这些方程的方法。事实上，对每一道动态规划题目都思考其方程的意义以及如何得来，是加深对动态规划的理解、提高动态规划功力的好方法。

P03: 多重背包问题

题目

有N种物品和一个容量为V的背包。第i种物品最多有n[i]件可用，每件费用是c[i]，价值是w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

基本算法

这题目和完全背包问题很类似。基本的方程只需将完全背包问题的方程略微一改即可，因为对于第i种物品有n[i]+1种策略：取0件，取1件……取 n[i]件。令f[i][v]表示前i种物品恰放入一个容量为v的背包的最大权值，则：f[i][v]=max{f[i-1][v-k*c[i]]+ k*w[i]|0<=k<=n[i]}。复杂度是O(V*∑n[i])。

转化为01背包问题

另一种好想好写的基本方法是转化为01背包求解：把第i种物品换成n[i]件01背包中的物品，则得到了物品数为∑n[i]的01背包问题，直接求解，复杂度仍然是O(V*∑n[i])。

但是我们期望将它转化为01背包问题之后能够像完全背包一样降低复杂度。仍然考虑二进制的思想，我们考虑把第i种物品换成若干件物品，使得原问题中第i种物品可取的每种策略——取0..n[i]件——均能等价于取若干件代换以后的物品。另外，取超过n[i]件的策略必不能出现。

方法是：将第i种物品分成若干件物品，其中每件物品有一个系数，这件物品的费用和价值均是原来的费用和价值乘以这个系数。使这些系数分别为 1,2,4,...,2^(k-1),n[i]-2^k+1，且k是满足n[i]-2^k+1>0的最大整数。例如，如果n[i]为13，就将这种物品分成系数分别为1,2,4,6的四件物品。

分成的这几件物品的系数和为n[i]，表明不可能取多于n[i]件的第i种物品。另外这种方法也能保证对于0..n[i]间的每一个整数，均可以用若干个系数的和表示，这个证明可以分0..2^k-1和2^k..n[i]两段来分别讨论得出，并不难，希望你自己思考尝试一下。

这样就将第i种物品分成了O(log n[i])种物品，将原问题转化为了复杂度为O(V*∑log n[i])的01背包问题，是很大的改进。

O(VN)的算法

多重背包问题同样有O(VN)的算法。这个算法基于基本算法的状态转移方程，但应用单调队列的方法使每个状态的值可以以均摊O(1)的时间求解。由于用单调队列优化的DP已超出了NOIP的范围，故本文不再展开讲解。我最初了解到这个方法是在楼天成的“男人八题”幻灯片上。

小结

这里我们看到了将一个算法的复杂度由O(V*∑n[i])改进到O(V*∑log n[i])的过程，还知道了存在应用超出NOIP范围的知识的O(VN)算法。希望你特别注意“拆分物品”的思想和方法，自己证明一下它的正确性，并用尽量简洁的程序来实现。

P04: 混合三种背包问题

问题

如果将P01、P02、P03混合起来。也就是说，有的物品只可以取一次（01背包），有的物品可以取无限次（完全背包），有的物品可以取的次数有一个上限（多重背包）。应该怎么求解呢？

01背包与完全背包的混合

考虑到在P01和P02中最后给出的伪代码只有一处不同，故如果只有两类物品：一类物品只能取一次，另一类物品可以取无限次，那么只需在对每个物品应用转移方程时，根据物品的类别选用顺序或逆序的循环即可，复杂度是O(VN)。伪代码如下：

```
for i=1..N
if 第i件物品是01背包
for v=V..0
f[v]=max{f[v],f[v-c[i]]+w[i]};
else if 第i件物品是完全背包
for v=0..V
f[v]=max{f[v],f[v-c[i]]+w[i]};
```

再加上多重背包

如果再加上有的物品最多可以取有限次，那么原则上也可以给出O(VN)的解法：遇到多重背包类型的物品用单调队列解即可。但如果不考虑超过NOIP范围的算法的话，用P03中将每个这类物品分成O(log n[i])个01背包的物品的方法也已经很优了。

小结

有人说，困难的题目都是由简单的题目叠加而来的。这句话是否公理暂且存之不论，但它在本讲中已经得到了充分的体现。本来01背包、完全背包、多重背包都不是什么难题，但将它们简单地组合起来以后就得到了这样一道一定能吓倒不少人的题目。但只要基础扎实，领会三种基本背包问题的思想，就可以做到把困难的题目拆分成简单的题目来解决。

P05：二维费用的背包问题

问题

二维费用的背包问题是指：对于每件物品，具有两种不同的费用；选择这件物品必须同时付出这两种代价；对于每种代价都有一个可付出的最大值（背包容量）。问怎样选择物品可以得到最大的价值。设这两种代价分别为代价1和代价2，第i件物品所需的两种代价分别为a[i]和b[i]。两种代价可付出的最大值（两种背包容量）分别为V和U。物品的价值为w[i]。

算法

费用加了一维，只需状态也加一维即可。设f[i][v][u]表示前i件物品付出两种代价分别为v和u时可获得的最大价值。状态转移方程就是：f [i][v][u]=max{f[i-1][v][u],f[i-1][v-a[i]][u-b[i]]+w[i]}。如前述方法，可以只使用二维的数组：当每件物品只可以取一次时变量v和u采用顺序的循环，当物品有如完全背包问题时采用逆序的循环。当物品有如多重背包问题时拆分物品。

物品总个数的限制

有时，“二维费用”的条件是以这样一种隐含的方式给出的：最多只能取M件物品。这事实上相当于每件物品多了一种“件数”的费用，每个物品的件数费用均为1，可以付出的最大件数费用为M。换句话说，设f[v][m]表示付出费用v、最多选m件时可得到的最大价值，则根据物品的类型（01、完全、多重）用不同的方法循环更新，最后在f[0..V][0..M]范围内寻找答案。

另外，如果要求“恰取M件物品”，则在f[0..V][M]范围内寻找答案。

小结

事实上，当发现由熟悉的动态规划题目变形得来的题目时，在原来的状态中加一纬以满足新的限制是一种比较通用的方法。希望你能从本讲中初步体会到这种方法。

P06：分组的背包问题

问题

有N件物品和一个容量为V的背包。第i件物品的费用是c[i]，价值是w[i]。这些物品被划分为若干组，每组中的物品互相冲突，最多选一件。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

算法

这个问题变成了每组物品有若干种策略：是选择本组的某一件，还是一件都不选。也就是说设f[k][v]表示前k组物品花费费用v能取得的最大权值，则有f[k][v]=max{f[k-1][v],f[k-1][v-c[i]]+w[i]|物品i属于第k组}。

使用一维数组的伪代码如下：

for 所有的组k

for 所有的i属于组k

for v=V..0

f[v]=max{f[v],f[v-c[i]]+w[i]}

另外，显然可以对每组中的物品应用P02中“一个简单有效的优化”。

小结

分组的背包问题将彼此互斥的若干物品称为一个组，这建立了一个很好的模型。不少背包问题的变形都可以转化为分组的背包问题（例如P07），由分组的背包问题进一步可定义“泛化物品”的概念，十分有利于解题。

P07: 有依赖的背包问题

简化的问题

这种背包问题的物品间存在某种“依赖”的关系。也就是说，i依赖于j，表示若选物品i，则必须选物品j。为了简化起见，我们先设没有某个物品既依赖于别的物品，又被别的物品所依赖；另外，没有某件物品同时依赖多件物品。

算法

这个问题由NOIP2006金明的预算方案一题扩展而来。遵从该题的提法，将不依赖于别的物品的物品称为“主件”，依赖于某主件的物品称为“附件”。由这个问题的简化条件可知所有的物品由若干主件和依赖于每个主件的一个附件集合组成。

按照背包问题的一般思路，仅考虑一个主件和它的附件集合。可是，可用的策略非常多，包括：一个也不选，仅选择主件，选择主件后再选择一个附件，选择主件后再选择两个附件……无法用状态转移方程来表示如此多的策略。（事实上，设有n个附件，则策略有2^n+1个，为指数级。）

考虑到所有这些策略都是互斥的（也就是说，你只能选择一种策略），所以一个主件和它的附件集合实际上对应于P06中的一个物品组，每个选择了主件又选择了若干个附件的策略对应于这个物品组中的一个物品，其费用和价值都是这个策略中的物品的值的和。但仅仅是这一步转化并不能给出一个好的算法，因为物品组中的物品还是像原问题的策略一样多。

再考虑P06中的一句话：可以对每组中的物品应用P02中“一个简单有效的优化”。这提示我们，对于一个物品组中的物品，所有费用相同的物品只留一个价值最大的，不影响结果。所以，我们

可以对主件i的“附件集合”先进行一次01背包，得到费用依次为0..V-c[i]所有这些值时相应的最大价值f'[0..V-c[i]]。那么这个主件及它的附件集合相当于V-c[i]+1个物品的物品组，其中费用为c[i]+k的物品的价值为f'[k]+w[i]。也就是说原来指数级的策略中有很多策略都是冗余的，通过一次01背包后，将主件i转化为 V-c[i]+1个物品的物品组，就可以直接应用P06的算法解决问题了。

更一般的问题

更一般的问题是：依赖关系以图论中“森林”的形式给出（森林即多叉树的集合），也就是说，主件的附件仍然可以具有自己的附件集合，限制只是每个物品最多只依赖于一个物品（只有一个主件）且不出现循环依赖。

解决这个问题仍然可以用将每个主件及其附件集合转化为物品组的方式。唯一不同的是，由于附件可能还有附件，就不能将每个附件都看作一个一般的01 背包中的物品了。若这个附件也有附件集合，则它必定要被先转化为物品组，然后用分组的背包问题解出主件及其附件集合所对应的附件组中各个费用的附件所对应的价值。

事实上，这是一种树形DP，其特点是每个父节点都需要对它的各个儿子的属性进行一次DP以求得自己的相关属性。这已经触及到了“泛化物品”的思想。看完P08后，你会发现这个“依赖关系树”每一个子树都等价于一件泛化物品，求某节点为根的子树对应的泛化物品相当于求其所有儿子的对应的泛化物品之和。

小结

NOIP2006的那道背包问题我做得很失败，写了上百行的代码，却一分未得。后来我通过思考发现通过引入“物品组”和“依赖”的概念可以加深对这题的理解，还可以解决它的推广问题。用物品组的思想考虑那题中极其特殊的依赖关系：物品不能既作主件又作附件，每个主件最多有两个附件，可以发现一个主件和它的两个附件等价于一个由四个物品组成的物品组，这便揭示了问题的某种本质。

我想说：失败不是什么丢人的事情，从失败中全无收获才是。

P08: 泛化物品

定义

考虑这样一种物品，它并没有固定的费用和价值，而是它的价值随着你分配给它的费用而变化。这就是泛化物品的概念。

更严格的定义之。在背包容量为V的背包问题中，泛化物品是一个定义域为0..V中的整数的函数h，当分配给它的费用为v时，能得到的价值就是h(v)。

这个定义有一点点抽象，另一种理解是一个泛化物品就是一个数组h[0..V]，给它费用v，可得到价值h[V]。

一个费用为c价值为w的物品，如果它是01背包中的物品，那么把它看成泛化物品，它就是除了h(c)=w其它函数值都为0的一个函数。如果它是完全背包中的物品，那么它可以看成这样一个函数，仅当v被c整除时有h(v)=v/c*w，其它函数值均为0。如果它是多重背包中重复次数最多为n

的物品，那么它对应的泛化物品的函数有h(v)=v/c*w仅当v被c整除且v/c<=n，其它情况函数值均为0。

一个物品组可以看作一个泛化物品h。对于一个0..V中的v，若物品组中不存在费用为v的的物品，则h(v)=0，否则h(v)为所有费用为v的物品的最大价值。P07中每个主件及其附件集合等价于一个物品组，自然也可看作一个泛化物品。

泛化物品的和

如果面对两个泛化物品h和l，要用给定的费用从这两个泛化物品中得到最大的价值，怎么求呢？事实上，对于一个给定的费用v，只需枚举将这个费用如何分配给两个泛化物品就可以了。同样的，对于0..V的每一个整数v，可以求得费用v分配到h和l中的最大价值f(v)。也即f(v)=max{h(k)+l(v-k)|0<=k<=v}。可以看到，f也是一个由泛化物品h和l决定的定义域为0..V的函数，也就是说，f是一个由泛化物品h和 l决定的泛化物品。

由此可以定义泛化物品的和：h、l都是泛化物品，若泛化物品f满足f(v)=max{h(k)+l(v-k)|0<=k<=v}，则称f是h与l的和，即f=h+l。这个运算的时间复杂度是O(V^2)。

泛化物品的定义表明：在一个背包问题中，若将两个泛化物品代以它们的和，不影响问题的答案。事实上，对于其中的物品都是泛化物品的背包问题，求它的答案的过程也就是求所有这些泛化物品之和的过程。设此和为s，则答案就是s[0..V]中的最大值。

背包问题的泛化物品

一个背包问题中，可能会给出很多条件，包括每种物品的费用、价值等属性，物品之间的分组、依赖等关系等。但肯定能将问题对应于某个泛化物品。也就是说，给定了所有条件以后，就可以对每个非负整数v求得：若背包容量为v，将物品装入背包可得到的最大价值是多少，这可以认为是定义在非负整数集上的一件泛化物品。这个泛化物品——或者说问题所对应的一个定义域为非负整数的函数——包含了关于问题本身的高度浓缩的信息。一般而言，求得这个泛化物品的一个子域（例如0..V）的值之后，就可以根据这个函数的取值得到背包问题的最终答案。

综上所述，一般而言，求解背包问题，即求解这个问题所对应的一个函数，即该问题的泛化物品。而求解某个泛化物品的一种方法就是将它表示为若干泛化物品的和然后求之。

小结

本讲可以说都是我自己的原创思想。具体来说，是我在学习函数式编程的 Scheme 语言时，用函数编程的眼光审视各类背包问题得出的理论。这一讲真的很抽象，也许在“模型的抽象程度”这一方面已经超出了NOIP的要求，所以暂且看不懂也没关系。相信随着你的OI之路逐渐延伸，有一天你会理解的。

我想说：“思考”是一个OIer最重要的品质。简单的问题，深入思考以后，也能发现更多。

P09: 背包问题问法的变化

以上涉及的各种背包问题都是要求在背包容量（费用）的限制下求可以取到的最大价值，但背包问题还有很多种灵活的问法，在这里值得提一下。但是我认为，只要深入理解了求背包问题最大价值的方法，即使问法变化了，也是不难想出算法的。

例如，求解最多可以放多少件物品或者最多可以装满多少背包的空间。这都可以根据具体问题利用前面的方程求出所有状态的值（f数组）之后得到。

还有，如果要求的是“总价值最小”“总件数最小”，只需简单的将上面的状态转移方程中的max改成min即可。

下面说一些变化更大的问法。

输出方案

一般而言，背包问题是要求一个最优值，如果要求输出这个最优值的方案，可以参照一般动态规划问题输出方案的方法：记录下每个状态的最优值是由状态转移方程的哪一项推出来的，换句话说，记录下它是由哪一个策略推出来的。便可根据这条策略找到上一个状态，从上一个状态接着向前推即可。

还是以01背包为例，方程为f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]}。再用一个数组g[i][v]，设g[i][v]=0表示推出f[i][v]的值时是采用了方程的前一项（也即f[i][v]=f[i-1][v]），g[i][v]表示采用了方程的后一项。注意这两项分别表示了两种策略：未选第i个物品及选了第i个物品。那么输出方案的伪代码可以这样写（设最终状态为f[N][V]）：

```
i=N
v=V
while(i>0)
if(g[i][v]==0)
print "未选第i项物品"
else if(g[i][v]==1)
print "选了第i项物品"
v=v-c[i]
```

另外，采用方程的前一项或后一项也可以在输出方案的过程中根据f[i][v]的值实时地求出来，也即不须纪录g数组，将上述代码中的g[i] [v]==0改成f[i][v]==f[i-1][v]，g[i][v]==1改成f[i][v]==f[i-1][v-c[i]]+w[i]也可。

输出字典序最小的最优方案

这里“字典序最小”的意思是1..N号物品的选择方案排列出来以后字典序最小。以输出01背包最小字典序的方案为例。

一般而言，求一个字典序最小的最优方案，只需要在转移时注意策略。首先，子问题的定义要略改一些。我们注意到，如果存在一个选了物品1的最优方案，那么答案一定包含物品1，原问题转化为一个背包容量为v-c[1]，物品为2..N的子问题。反之，如果答案不包含物品1，则转化成背包容量仍为V，物品为2..N的子问题。不管答案怎样，子问题的物品都是以i..N而非前所述的1..i的形式来定义的，所以状态的定义和转移方程都需要改一下。但也许更简易的方法是先把物品逆序

排列一下，以下按物品已被逆序排列来叙述。

在这种情况下，可以按照前面经典的状态转移方程来求值，只是输出方案的时候要注意：从N到1输入时，如果f[i][v]==f[i-v]及f[i][v]==f[i-1][f-c[i]]+w[i]同时成立，应该按照后者（即选择了物品i）来输出方案。

求方案总数

对于一个给定了背包容量、物品费用、物品间相互关系（分组、依赖等）的背包问题，除了再给定每个物品的价值后求可得到的最大价值外，还可以得到装满背包或将背包装至某一指定容量的方案总数。

对于这类改变问法的问题，一般只需将状态转移方程中的max改成sum即可。例如若每件物品均是01背包中的物品，转移方程即为f[i][v]=sum{f[i-1][v],f[i-1][v-c[i]]+w[i]}，初始条件f[0][0]=1。

事实上，这样做可行的原因在于状态转移方程已经考察了所有可能的背包组成方案。

最优方案的总数

这里的最优方案是指物品总价值最大的方案。还是以01背包为例。

结合求最大总价值和方案总数两个问题的思路，最优方案的总数可以这样求：f[i][v]意义同前述，g[i][v]表示这个子问题的最优方案的总数，则在求f[i][v]的同时求g[i][v]的伪代码如下：

```
for i=1..N
for v=0..V
f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]}
g[i][v]=0
if(f[i][v]==f[i-1][v])
inc(g[i][v],g[i-1][v]
if(f[i][v]==f[i-1][v-c[i]]+w[i])
inc(g[i][v],g[i-1][v-c[i]])
```

如果你是第一次看到这样的问题，请仔细体会上面的伪代码。

小结

显然，这里不可能穷尽背包类动态规划问题所有的问法。甚至还存在一类将背包类动态规划问题与其它领域（例如数论、图论）结合起来的问题，在这篇论背包问题的专文中也不会论及。但只要深刻领会前述所有类别的背包问题的思路和状态转移方程，遇到其它的变形问法，只要题目难度还属于NOIP，应该也不难想出算法。

触类旁通、举一反三，应该也是一个OIer应有的品质吧。

来自 <http://i.cnblogs.com/EditArticles.aspx?postid=4964317>

双十一-吃土

2016年5月9日　　23:50

吃土

时间限制: 1000 ms 内存限制: 65536 kb

总通过人数: 132 总提交人数: 133

题目描述

输入

多组测试数据（不超过10组），每组数据两行

第一行为一个正整数N（N<=10000），表示排队挖土的总人数

第二行为N个正整数a1,a2,,,an（INT范围内），表示每个人挖土所需的时间

输出

对于每组数据，输出一行，表示等待时间之和的最小值

输入样例

```
5
1 2 3 4 5
```

输出样例

```
20
```

题目来源：http://biancheng.love/contest/17/problem/A/index

解题分析：

按照题目定义，当挖土的时间排序为a1,a2,....an,时，所消耗的时间为T= ∑（i=1->n）（n-i）*ai; 因此当所消耗时间最少时，挖图时间的排序序列应改为非递减排序。因此可以调用库函数的sort，也可以使用前些天提到的优先队列。

代码实现：

```
#include <bits/stdc++.h>
#define max_size 10010
long long a[max_size];
using namespace std;

int main()
{
    int n;
    while(~scanf("%d",&n))
    {
        for(int i=1;i<=n;i++)
            scanf("%lld",&a[i]);
        sort(a+1,a+n+1);
        long long ans=0;
        for(int i=1;i<=n;i++)
        {
            ans+=(n-i)*a[i];
        }
        printf("%lld\n",ans);
    }
}
```

另附优先队列实现：

```
#include <bits/stdc++.h>

using namespace std;

```

```cpp
int main()
{
    int n;
    while(~scanf("%d",&n))
    {
        long long num;
        priority_queue<long long,vector<long long>,greater<long long> >que;
        for(int i=1;i<=n;i++)
        {
            scanf("%lld",&num);
            que.push(num);
        }
        long long kase=1;
        long long ans=0;
        while(!que.empty())
        {
            ans+=(n-kase)*que.top();
            que.pop();
            kase++;
        }
        printf("%lld\n",ans);
    }
}
```

注意事项：计算结果较大请使用long long 数据类型。

来自 <http://i.cnblogs.com/EditPosts.aspx?postid=4963988>

优先队列详解

2016年5月9日　　23:50

优先队列：顾名思义，首先它是一个队列，但是它强调了“优先”二字，所以，已经不能算是一般意义上的队列了，它的“优先”意指取队首元素时，有一定的选择性，即根据元素的属性选择某一项值最优的出队~
百度百科上这样描述的：
　　优先级队列 是不同于先进先出队列的另一种队列。每次从队列中取出的是具有最高优先权的元素
　　优先队列的类定义
　　优先队列是0个或多个元素的集合,每个元素都有一个优先权或值,对优先队列执行的操作有1) 查找;2) 插入一个新元素;3) 删除.在最小优先队列(min priorityq u e u e)中,查找操作用来搜索优先权最小的元素,删除操作用来删除该元素;对于最大优先队列(max priority queue),查找操作用来搜索优先权最大的元素,删除操作用来删除该元素.优先权队列中的元素可以有相同的优先权,查找与删除操作可根据任意优先权进行.
优先队列，其构造及具体实现我们可以先不用深究，我们现在只需要了解其特性，及在做题中的用法，相信，看过之后你会收获不少。
使用优先队列，首先要包函STL头文件"queue"，
以一个例子来解释吧(呃，写完才发现，这个代码包函了几乎所有我们要用到的用法，仔细看看吧)：
view plaincopy to clipboardprint?

```
/*优先队列的基本使用　2010/7/24　dooder*/
#include<stdio.h>
#include<functional>
#include<queue>
#include<vector>
using namespace std;
//定义结构，使用运算符重载,自定义优先级1
struct cmp1{
   bool operator ()(int &a,int &b){
      return a>b;//最小值优先
   }
};
struct cmp2{
   bool operator ()(int &a,int &b){
      return a<b;//最大值优先
   }
};
//定义结构，使用运算符重载,自定义优先级2
struct number1{
   int x;
   bool operator < (const number1 &a) const {
      return x>a.x;//最小值优先
   }
};
struct number2{
   int x;
   bool operator < (const number2 &a) const {
      return x<a.x;//最大值优先
   }
};
int a[]={14,10,56,7,83,22,36,91,3,47,72,0};
number1 num1[]={14,10,56,7,83,22,36,91,3,47,72,0};
number2 num2[]={14,10,56,7,83,22,36,91,3,47,72,0};

int main()
{  priority_queue<int>que;//采用默认优先级构造队列
```

```
priority_queue<int,vector<int>,cmp1>que1;//最小值优先
priority_queue<int,vector<int>,cmp2>que2;//最大值优先

priority_queue<int,vector<int>,greater<int> >que3;//注意“>>”会被认为错误，
                                    //这是右移运算符，所以这里用空格号隔开
priority_queue<int,vector<int>,less<int> >que4;////最大值优先

priority_queue<number1>que5;
priority_queue<number2>que6;

int i;
for(i=0;a[i];i++){
   que.push(a[i]);
   que1.push(a[i]);
   que2.push(a[i]);
   que3.push(a[i]);
   que4.push(a[i]);
}
for(i=0;num1[i].x;i++)
   que5.push(num1[i]);
for(i=0;num2[i].x;i++)
   que6.push(num2[i]);


printf("采用默认优先关系:\n(priority_queue<int>que;)\n");
printf("Queue 0:\n");
while(!que.empty()){
   printf("%3d",que.top());
   que.pop();
}
puts("");
puts("");

printf("采用结构体自定义优先级方式一:\n(priority_queue<int,vector<int>,cmp>que;)\n");
printf("Queue 1:\n");
while(!que1.empty()){
   printf("%3d",que1.top());
   que1.pop();
}
puts("");
printf("Queue 2:\n");
while(!que2.empty()){
   printf("%3d",que2.top());
   que2.pop();
}
puts("");
puts("");
printf("采用头文件\"functional\"内定义优先级:
\n(priority_queue<int,vector<int>,greater<int>/less<int> >que;)\n");
printf("Queue 3:\n");
while(!que3.empty()){
   printf("%3d",que3.top());
   que3.pop();
}
puts("");
printf("Queue 4:\n");
while(!que4.empty()){
   printf("%3d",que4.top());
```

```
        que4.pop();
    }
    puts("");
    puts("");
    printf("采用结构体自定义优先级方式二:\n(priority_queue<number>que)\n");
    printf("Queue 5:\n");
    while(!que5.empty()){
        printf("%3d",que5.top());
        que5.pop();
    }
    puts("");
    printf("Queue 6:\n");
    while(!que6.empty()){
        printf("%3d",que6.top());
        que6.pop();
    }
    puts("");
    return 0;
}
/*
运行结果：
采用默认优先关系:
(priority_queue<int>que;)
Queue 0:
56 47 36 22 14 10  7  3

采用结构体自定义优先级方式一:
(priority_queue<int,vector<int>,cmp>que;)
Queue 1:
14 22 36 47 56 72 83 91
Queue 2:
56 47 36 22 14 10  7  3

采用头文件"functional"内定义优先级:
(priority_queue<int,vector<int>,greater<int>/less<int> >que;)
Queue 3:
14 22 36 47 56 72 83 91
Queue 4:
56 47 36 22 14 10  7  3

采用结构体自定义优先级方式二:
(priority_queue<number>que)
Queue 5:
14 22 36 47 56 72 83 91
Queue 6:
56 47 36 22 14 10  7  3
*/
运行结果：
采用默认优先关系:
(priority_queue<int>que;)
Queue 0:
56 47 36 22 14 10  7  3
采用结构体自定义优先级方式一:
(priority_queue<int,vector<int>,cmp>que;)
Queue 1:
14 22 36 47 56 72 83 91
Queue 2:
56 47 36 22 14 10  7  3
采用头文件"functional"内定义优先级:
```

(priority_queue<int,vector<int>,greater<int>/less<int> >que;)
Queue 3:
14 22 36 47 56 72 83 91
Queue 4:
56 47 36 22 14 10 7 3
采用结构体自定义优先级方式二:
(priority_queue<number>que)
Queue 5:
14 22 36 47 56 72 83 91
Queue 6:
56 47 36 22 14 10 7 3
好了，如果你仔细看完了上面的代码，那么你就可以基本使用优先队列了，下面给出一些我做题中有过的一些应用，希望能给大家带来一些启
示~
1、先来一个我们最近做的题吧，http://acm.hdu.edu.cn/showproblem.php?pid=1242
题意：某人被关在囚笼里等待朋友解救，问能否解救成功，最少需要多少时间~
具体：可同时有几个朋友，每走一格消耗一分钟的时间，地图上还存在着卫兵，卫兵可以解决掉，但是要另外花费一分钟~
分析：从“a”出发，此题可以用回溯法进行深搜，但那样做的话，效率还是不能让人满意，但是广搜的话，由于入队后每次出队时，根据地
图情况的不同，出队元素所记忆的时间并不是层次递增的，因此使用简单广搜的话，同样需要全部搜索才能找到正确答案。有没有一种方法能
让某一步因为遇到士兵而多花时间的结点在队列中向后推迟一层出队呢？答案是肯定的，在这里我们可以用优先队列来实现，总体思想上是，
根据时间进行优先性选择，每次都要出队当前队列元素中记录时间最少的出队，而入队处理时，我们可以按顺序对四个方向上的各种情况按正
常处理入队就行了，出队顺序由优先队列根据预设优先性自动控制。这样，我们就可以从“a”进行基于优先队列的范围搜索了，并且在第一
次抵达有朋友的位置时得到正确结果~具体实现代码：
view plaincopy to clipboardprint?

```
/*HDU 1242  基于优先队列的范围搜索，16ms   dooder*/

#include<stdio.h>
#include<queue>
using namespace std;

#define M 201
typedef struct p{
   int x,y,t;
   bool operator < (const p &a)const
   {
      return t>a.t;//取时间最少优先
   }
}Point;

char map[M][M];
Point start;
int n,m;
int dir[][2]={{1,0},{-1,0},{0,1},{0,-1}};

int bfs()
{
   priority_queue<Point>que;
   Point cur,next;
   int i;

   map[start.x][start.y]='#';
   que.push(start);
```

```
    while(!que.empty()){
        cur=que.top();//由优先队列自动完成出队时间最少的元素
        que.pop();
        for(i=0;i<4;i++){
            next.x=cur.x+dir[i][0];
            next.y=cur.y+dir[i][1];
            next.t=cur.t+1;
            if(next.x<0||next.x>=n||next.y<0||next.y>=m)
                continue;
            if(map[next.x][next.y]=='#')
                continue;
            if(map[next.x][next.y]=='r')
                return next.t;
            if(map[next.x][next.y]=='.'){
                map[next.x][next.y]='#';
                que.push(next);
            }
            else if(map[next.x][next.y]=='x'){
                map[next.x][next.y]='#';
                next.t++;
                que.push(next);
            }
        }
    }
    return -1;
}
int main()
{
    int i,ans;
    char *p;
    while(scanf("%d%d",&n,&m)!=-1){
        for(i=0;i<n;i++){
            scanf("%s",map[i]);
            if(p=strchr(map[i],'a')){
                start.x=i;
                start.y=p-map[i];
                start.t=0;
            }
        }
        ans=bfs();
        printf(ans+1?"%d\n":"Poor ANGEL has to stay in the prison all his life.\n",ans);
    }
    return 0;
}
```

2、http://acm.hdu.edu.cn/showproblem.php?pid=1053

题意：给出一行字符串，求出其原编码需要的编码长度和哈夫曼编码所需的长度，并求其比值

分析：根据哈夫曼生成树的生成过程可知，其生成树的权值是固定的而且这个值是最小的，而且其值根据生成树的顺序，我们可以找出规律而

不需要真的去生成一棵树然后再求出权值，其模拟过程为取出队列中权值最小的两个元素，将其值加入结果中，然后将这两个元素的权值求和

即得出其父节点的权值，将生成元素作为结点入队~~如此循环，直至取出队列中最后两个元素加入结果，实现代码如下：

view plaincopy to clipboardprint?

```
/*HDU 1053  采用广搜求哈夫曼生成树的权值 0ms   dooder*/
#include<stdio.h>
#include<string.h>
#include<ctype.h>
#include<functional>
#include<queue>
```

```
using namespace std;
#define M 1000050
char str[M];
int list[27];

priority_queue< int,vector<int>,greater<int> >que;

int main()
{
   int ans,sum;
   int i,a,b,c;
   while(scanf("%s",str),strcmp(str,"END")){
      memset(list,0,sizeof(list));
      for(i=0;str[i];i++){
         if(isalpha(str[i]))
            list[str[i]-'A']++;
         else
            list[26]++;
      }
      sum=i*8;ans=i;c=0;
      for(i=0;i<27;i++){
         if(list[i]){
            que.push(list[i]);
            c++;
         }
      }
      if(c>1){ans=0;//注意只有一种字符的情况
         while(que.size()!=1){
            a=que.top();
            que.pop();
            b=que.top();
            que.pop();
            ans+=a+b;
            que.push(a+b);
         }
         while(!que.empty())//使用后清空队列
            que.pop();
      }
      printf("%d %d %.1f\n",sum,ans,1.0*sum/ans);
   }
   return 0;
}
```

3、http://acm.pku.edu.cn/JudgeOnline/problem?id=2263

这是第二次练习赛时，我们做过的最后一题，这里采用优先队列进行实现，在《谁说不能这样做题》中已提到这种方法，在这里再次放出代码，~

题意：给出各城市间道路的限制载重量，求出从一个城市到另外一个城市的贷车能够运载的最大货物重量。

分析：采用优先队列，每次取出当前队列中结点的minheavy最大值出队，对它的连接结点搜索入队,这样，从出发点开始就可以在到达终点时求出结果，即最大载货物重，实现代码如下：

view plaincopy to clipboardprint?

```
/*POJ 2263  16ms  dooder*/

#include<stdio.h>
#include<string.h>
#include<queue>
using namespace std;
#define M 201
```

```
typedef struct w{
   int city;
   int mintons;
   bool operator < (const w &a)const {
      return mintons < a.mintons;
   }//优先性定义
}Way;
char citys[M][31];
int map[M][M];
bool mark[M][M];
int n,m,from,to,ans,k;
priority_queue <Way> que;
int min(int a,int b)
{
   return a>b?b:a;
}
void bfs()
{
   Way cur,next;
   int i;
   while(!que.empty()){
      cur=que.top();
      que.pop();
      if(cur.city==to){
         if(cur.mintons>ans)
            ans=cur.mintons;
         while(!que.empty())
            que.pop();
         return ;
      }
      for(i=0;i<n;i++){
         if(map[cur.city][i]&&!mark[cur.city][i]){
            next.city=i;
            next.mintons=min(cur.mintons,map[cur.city][i]);

            mark[cur.city][i]=mark[i][cur.city]=1;
            que.push(next);
         }
      }
   }
}
void run()
{
   int i,temp,index;
   Way cur;
   ans=0;
   memset(mark,0,sizeof(mark));
   temp=0;
   for(i=0;i<n;i++){
      if(map[from][i]>temp){
         temp=map[from][i];
         index=i;
      }
   }
   cur.city=index;
   cur.mintons=temp;
   que.push(cur);
   bfs();
}
int main()
```

```
{
   int k1,k2,tons,t=1;
   char s1[31],s2[31];
   while(scanf("%d%d",&n,&m),n||m){
      k=0;
      while(m--){
         scanf("%s%s%d",s1,s2,&tons);
         for(k1=0;strcmp(s1,citys[k1])&&k1<k;k1++);
         if(k1==k)
            strcpy(citys[k++],s1);
         for(k2=0;strcmp(s2,citys[k2])&&k2<k;k2++);
         if(k2==k)
            strcpy(citys[k++],s2);
         map[k1][k2]=map[k2][k1]=tons;
      }
      scanf("%s%s",s1,s2);
      for(from=0;strcmp(citys[from],s1);from++);
      for(to=0;strcmp(citys[to],s2);to++);
      run();
      printf("Scenario #%d\n",t++);
      printf("%d tons\n\n",ans);
   }
   return 0;
}
```

当然了，优先队列的用法决不是仅仅提到的这些，各种应用还需要大家去发现，给道题大家可以练习一下hdu 2066\

相信大家已经学到不少了，还有一点可以告诉大家，优先队列是启发式搜索的数据结构基础，希望好好理解，并逐步掌握其用法~

加：失策啊，竟然忘了说优先队列的效率了，其时间复杂度为O(logn).n为队列中元素的个数，存取都需要消耗时间~

赫夫曼\哈夫曼\霍夫曼编码 (Huffman Tree)

2016年5月9日　　23:52

哈夫曼树

给定n个权值作为n的叶子结点，构造一棵二叉树，若带权路径长度达到最小，称这样的二叉树为最优二叉树，也称为哈夫曼树(Huffman Tree)。哈夫曼树是带权路径长度最短的树，权值较大的结点离根较近。

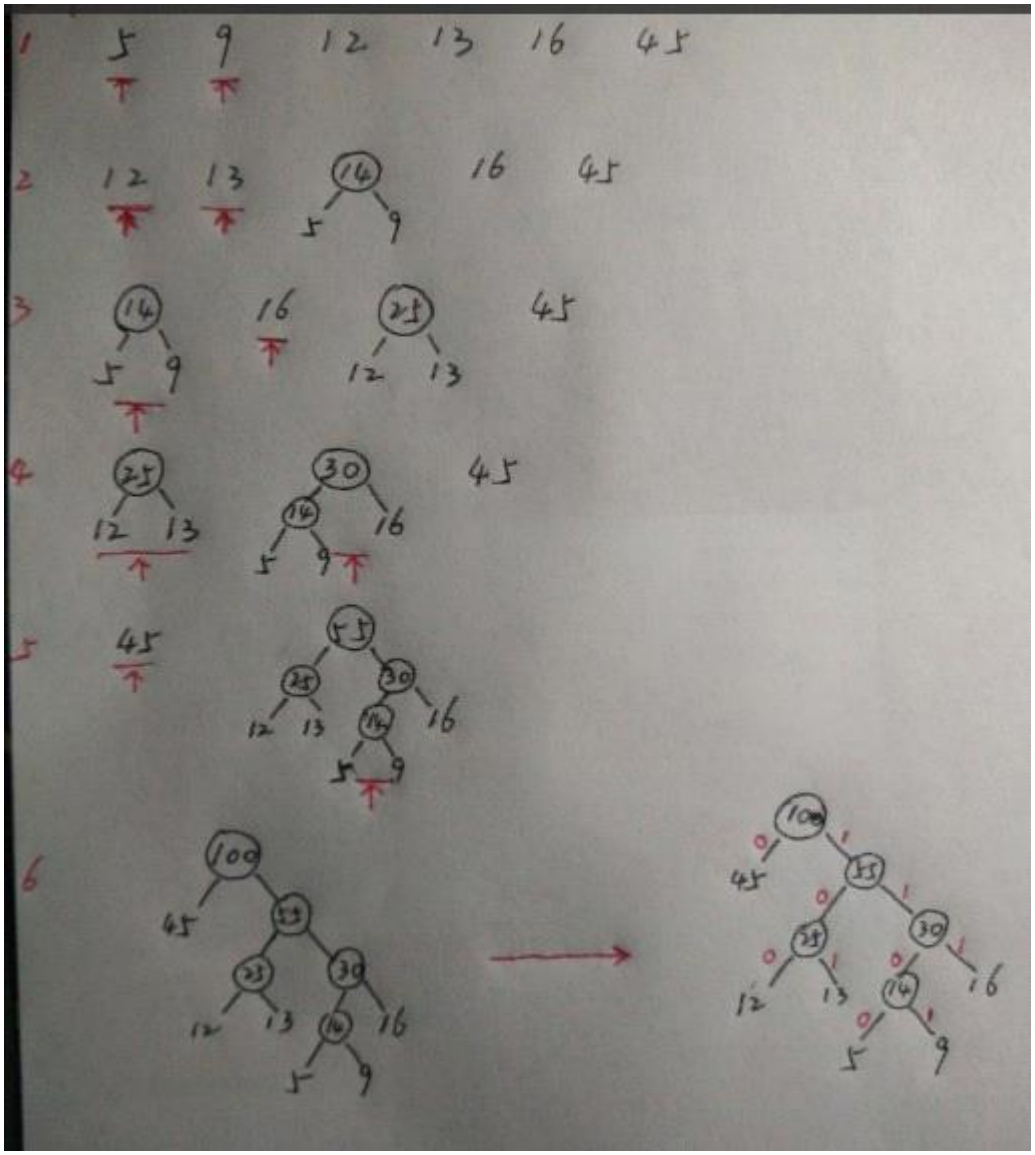
哈夫曼编码(Huffman Coding)

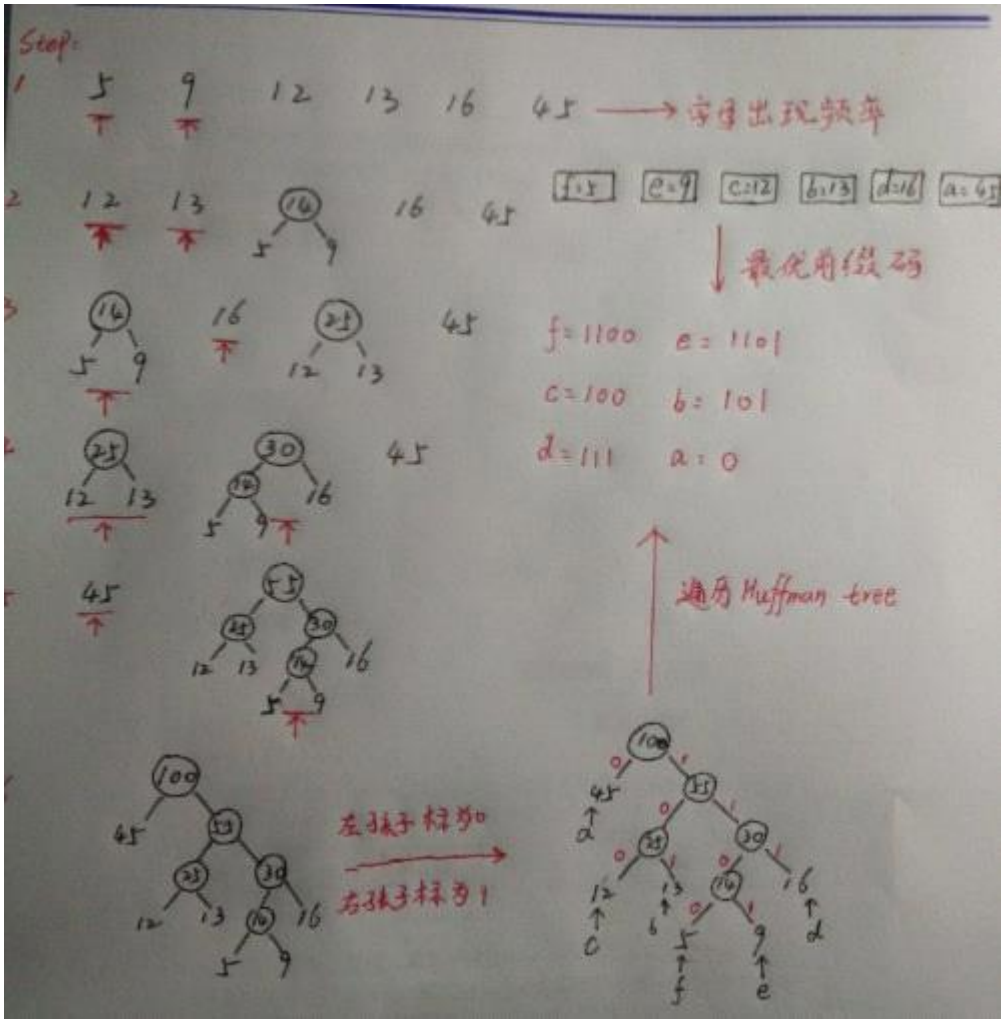
又称霍夫曼编码，是一种编码方式，哈夫曼编码是可变字长编码(VLC)的一种。Huffman于1952年提出一种编码方法，该方法完全依据字符出现概率来构造异字头的平均长度最短的码字，有时称之为最佳编码，一般就叫做Huffman编码（有时也称为霍夫曼编码）。

应用举例

哈夫曼树—即最优二叉树，带权路径长度最小的二叉树，经常应用于数据压缩。 在计算机信息处理中，“哈夫曼编码”是一种一致性编码法（又称“熵编码法”），用于数据的无损耗压缩。这一术语是指使用一张特殊的编码表将源字符（例如某文件中的一个符号）进行编码。这张编码表的特殊之处在于，它是根据每一个源字符出现的估算概率而建立起来的（出现概率高的字符使用较短的编码，反之出现概率低的则使用较长的编码，这便使编码之后的字符串的平均期望长度降低，从而达到无损压缩数据的目的）。

构造Huffman tree


1
5
9
12
13
16
45
2
12
13
14
16
45
5
9
3
14
16
25
45
5
9
12
13
4
25
30
45
12
13
14
16
5
9
5
45
25
30
12
13
14
16
5
9
6
100
45
55
25
30
12
13
14
16
5
9
100
0
1
45
55
0
1
25
30
0
1
0
1
12
13
14
16
0
1
5
9



过程：

1、输入字符关键字以及对应的频率。

2、根据关键字的频率构造最优二叉树

(1)借助于STL的优先队列 ：priority_queue<type, vector<type>,cmp>que; 其中type为优先队列的类型，比如说int, double, 也可以将结构体作为类型。vector它是一个多功能的，能够操作多种数据结构和算法的模板类和函数库。借助于vector可以很轻松的解决一些问题。cmp为定义的优先队列的出队规则，给出博客 http://www.cnblogs.com/buptLizer/archive/2011/09/11/2173708.html大家可以参考来理解优先队列的使用。以及其中常用的操作 que.pop(), que.push(), que.top(), que.size()等...

(2)构造最优二叉搜索树：在有了最优二叉搜索树的帮助之后，对于构建搜索树很容易。上面的图很清楚的讲解了构造的过程，每次取出频率最小的两个形成一个子树，将其频率相加之后再放入优先队列之中，直到队列中元素个

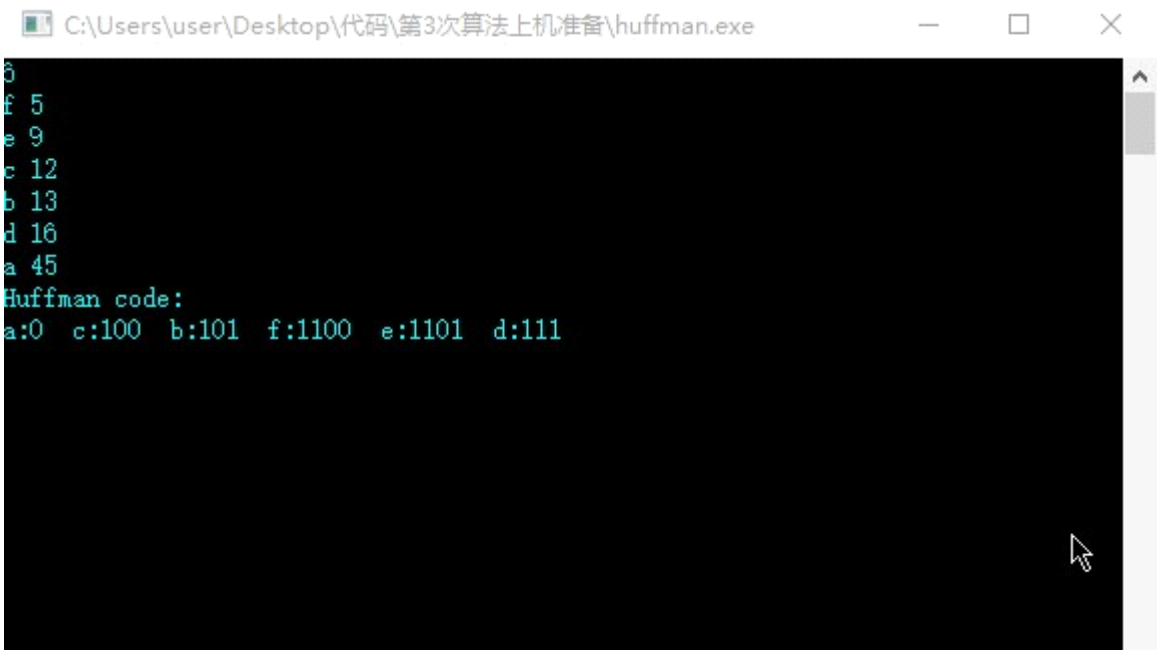
数为0。在操作过程中需要遵守最优二叉搜索树的定义，左孩子小于父节点，右孩子大于父节点。

(3)递归遍历二叉树：遍历中只需要记住，向左遍历那么前缀码为0,向右为1.

亲测代码：

View Code

输入数据以及运行结果：



来自 <http://i.cnblogs.com/EditPosts.aspx?postid=4957691>

OBST（Optimal Binary Tree最优二叉搜索树）

2016年5月9日　　23:52

二叉搜索树

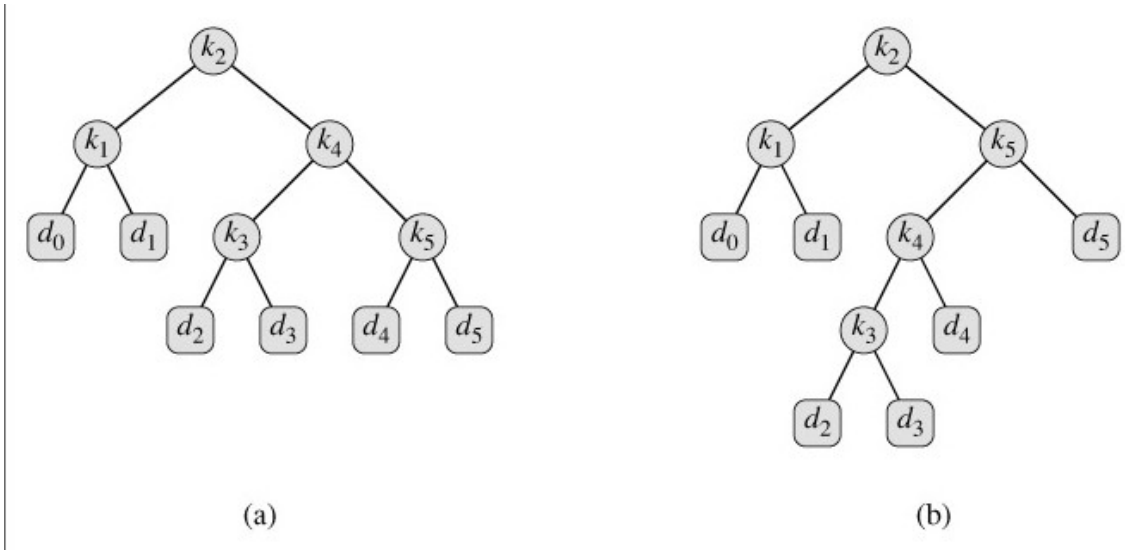
二叉查找树（Binary Search Tree），（又：二叉搜索树，二叉排序树）它或者是一棵空树，或者是具有下列性质的二叉树： 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值； 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值； 它的左、右子树也分别为二叉排序树。

一、什么是最优二叉查找树

最优二叉查找树：

给定n个互异的关键字组成的序列K=<k1,k2,...,kn>，且关键字有序（k1<k2<...<kn），我们想从这些关键字中构造一棵二叉查找树。对每个关键字ki，一次搜索搜索到的概率为pi。可能有一些搜索的值不在K内，因此还有n+1个“虚拟键”d0,d1,...,dn，他们代表不在K内的值。具体：d0代表所有小于k1的值，dn代表所有大于kn的值。而对于i = 1,2,...,n-1,虚拟键di代表所有位于ki和ki+1之间的值。对于每个虚拟键，一次搜索对应于di的概率为qi。要使得查找一个节点的期望代价（代价可以定义为：比如从根节点到目标节点的路径上节点数目）最小，就需要建立一棵最优二叉查找树。

图一显示了给定上面的概率分布pi、qi，生成的两个二叉查找树的例子。图二就是在这种情况下一棵最优二叉查找树。



概率分布：

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

已知每个关键字以及虚拟键被搜索到的概率，可以计算出一个给定二叉查找树内一次搜索的期望代价。假设一次搜索的实际代价为检查的节点的个数，即所发现的节点的深度加1.计算一次搜索的期望代价等式为：

$$\begin{aligned}\mathrm{E}[\text{search cost in } T] &= \sum_{i=1}^{n}(\mathrm{depth}_T(k_i)+1)\cdot p_i + \sum_{i=0}^{n}(\mathrm{depth}_T(d_i)+1)\cdot q_i \\ &= 1 + \sum_{i=1}^{n}\mathrm{depth}_T(k_i)\cdot p_i + \sum_{i=0}^{n}\mathrm{depth}_T(d_i)\cdot q_i\ ,\end{aligned}$$

建立一棵二叉查找树，如果是的上式最小，那么这棵二叉查找树就是**最优二叉查找树**。

而且有下式成立：

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

二、最优二叉查找树的最优子结构

最优子结构：

如果一棵最优二叉查找树T有一棵包含关键字ki,..,kj的子树T'，那么这可子树T'对于关键字Ki,...,kj和虚拟键di-1,...dj的子问题也必定是最优的。可以应用剪贴法证明。

根据最优子结构，寻找最优解：

给定关键字ki,...,kj，假设kr(i<=r<=j)是包含这些键的一棵最优子树的根。其左子树包含关键字ki,...,kr-1和虚拟键di-1,...,dr-1，右子树包含关键字kr+1,...,kj和虚拟键dr,...dj。我们检查所有的候选根kr,就保证可以找到一棵最优二叉查找树。

递归解：

定义e[i,j]为包含关键字ki,...,kj的最优二叉查找树的期望代价，最终要计算的是e[1,n]。

当j = i - 1时，此时子树中只有虚拟键，期望搜索代价为e[i,i - 1] = qi-1.

当j >= i时，需要从ki,...,kj中选择一个根kr，然后分别构造其左子树和右子树。下面需要计算以kr为根的树的期望搜索代价。然后选择导致最小期望搜索代价的kr做根。

现在需要考虑的是，当一棵树成为一个节点的子树时，期望搜索代价怎么变化？子树中每个节点深度都增加1.期望搜索代价增加量为子树中所有概率的总和。

对一棵关键字ki,...,kj的子树，定义其概率总和为：

$$w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l\ .$$

因此，以kr为根的子树的期望搜索代价为：

$$e[i,j] = p_r + (e[i,r-1] + w(i,r-1)) + (e[r+1,j] + w(r+1,j)).$$

而

$$w(i,j) = w(i,r-1) + p_r + w(r+1,j),$$

因此e[i,j]可以进一步写为：

$$e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j) .$$

这样推导出最终的递归公式为：

$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i-1 , \\ \min\limits_{i \le r \le j} \{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{if } i \le j . \end{cases}$$

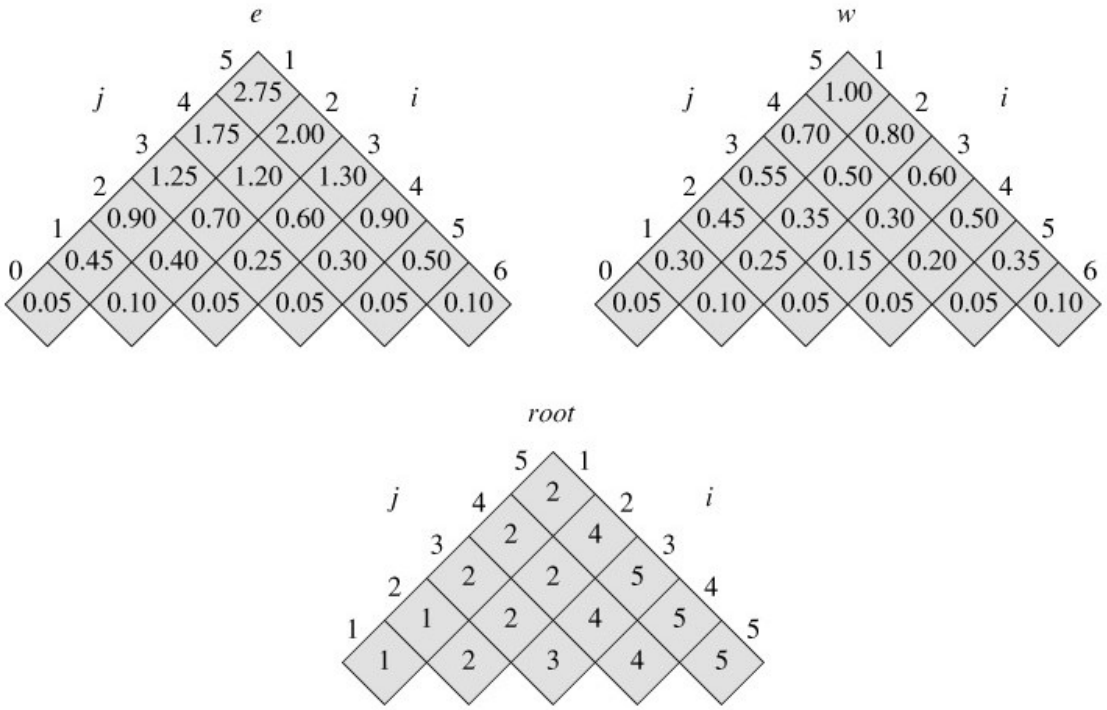
⊞
View Code

参考代码：

```
1  //最优二叉查找树
2
3  #include <iostream>
4
5  using namespace std;
6
7  const int MaxVal = 9999;
8
9  const int n = 5;
10 //搜索到根节点和虚拟键的概率
11 double p[n + 1] = {-1,0.15,0.1,0.05,0.1,0.2};
12 double q[n + 1] = {0.05,0.1,0.05,0.05,0.05,0.1};
13
14 int root[n + 1][n + 1];//记录根节点
15 double w[n + 2][n + 2];//子树概率总和
16 double e[n + 2][n + 2];//子树期望代价
17
18 void optimalBST(double *p,double *q,int n)
19 {
20     //初始化只包括虚拟键的子树
21     for (int i = 1;i <= n + 1;++i)
22     {
23         w[i][i - 1] = q[i - 1];
24         e[i][i - 1] = q[i - 1];
25     }
26
27     //由下到上，由左到右逐步计算
28     for (int len = 1;len <= n;++len)
29     {
30         for (int i = 1;i <= n - len + 1;++i)
31         {
32             int j = i + len - 1;
33             e[i][j] = MaxVal;
34             w[i][j] = w[i][j - 1] + p[j] + q[j];
35             //求取最小代价的子树的根
36             for (int k = i;k <= j;++k)
37             {
38                 double temp = e[i][k - 1] + e[k + 1][j] + w[i][j];
39                 if (temp < e[i][j])
40                 {
41                     e[i][j] = temp;
42                     root[i][j] = k;
43                 }
44             }
45         }
46     }
47 }
48
49 //输出最优二叉查找树所有子树的根
50 void printRoot()
51 {
52     cout << "各子树的根：" << endl;
53     for (int i = 1;i <= n;++i)
54     {
55         for (int j = 1;j <= n;++j)
```

```
56          {
57              cout << root[i][j] << " ";
58          }
59          cout << endl;
60      }
61      cout << endl;
62  }
63
64  //打印最优二叉查找树的结构
65  //打印出[i,j]子树，它是根r的左子树和右子树
66  void printOptimalBST(int i,int j,int r)
67  {
68      int rootChild = root[i][j];//子树根节点
69      if (rootChild == root[1][n])
70      {
71          //输出整棵树的根
72          cout << "k" << rootChild << "是根" << endl;
73          printOptimalBST(i,rootChild - 1,rootChild);
74          printOptimalBST(rootChild + 1,j,rootChild);
75          return;
76      }
77
78      if (j < i - 1)
79      {
80          return;
81      }
82      else if (j == i - 1)//遇到虚拟键
83      {
84          if (j < r)
85          {
86              cout << "d" << j << "是" << "k" << r << "的左孩子" << endl;
87          }
88          else
89              cout << "d" << j << "是" << "k" << r << "的右孩子" << endl;
90          return;
91      }
92      else//遇到内部结点
93      {
94          if (rootChild < r)
95          {
96              cout << "k" << rootChild << "是" << "k" << r << "的左孩子" << endl;
97          }
98          else
99              cout << "k" << rootChild << "是" << "k" << r << "的右孩子" << endl;
100     }
101
102     printOptimalBST(i,rootChild - 1,rootChild);
103     printOptimalBST(rootChild + 1,j,rootChild);
104 }
105
106 int main()
107 {
108     optimalBST(p,q,n);
109     printRoot();
110     cout << "最优二叉树结构：" << endl;
111     printOptimalBST(1,n,-1);
112 }
```

我们将表e、w以及root旋转45°，便于查看上述程序的计算过程。上述代码核心在于函数optimalBST，其计算顺序是从下到上、从左到右。首先是依据概率数组pi、qi初始化：给最下面的一行赋值。然后是三个for循环：从下到上计算表中每一行的值，可以充分利用前面计算出来的结果。如果每当计算e[i][j]的时候都从头开始计算w[i][j]，那么需要O(j-i)步加法，但是将这些值保存在表w[1...n+1][0...n]中，就避免这些复杂的计算。



输出结果：

来自 <http://i.cnblogs.com/EditPosts.aspx?postid=4953898>

LCS（Longest Common Subsequence 最长公共子序列）

2016年5月9日　　23:52

最长公共子序列

英文缩写为LCS（Longest Com*mon Subsequence*）。*其定义是，一个序列 S ，如果分别是两个或多个已知序列的子序列，且是所有符合此条件序列中最长的，则 S 称为已知序列的最长公共子序列。而最长公共子串(要求连续)和最长公共子序列是不同的*

应用

最长公共子序列是一个十分实用的问题，它可以描述两段文字之间的“相似度”，即它们的雷同程度，从而能够用来辨别抄袭。对一段文字进行修改之后，计算改动前后文字的最长公共子序列，将除此子序列外的部分提取出来，这种方法判断修改的部分，往往十分准确。简而言之，百度知道、百度百科都用得上。

动态规划

第一步：先计算最长公共子序列的长度。

第二步：根据长度，然后通过回溯求出最长公共子序列。

现有两个序列X={x_1,x_2,x_3, ...x_i}，Y={y_1,y_2,y_3, ...，y_i}，

设一个C[i,j]：保存X_i与Y_j的LCS的长度。

递推方程为：

$$C[i,j]=\begin{cases}0 & 若\ i=0或j=0\\ C[i-1,j-1]+1 & 若\ i,j>0,x_i=y_j\\ \max\{C[i,j-1],C[i-1,j]\} & 若\ i,j>0,x_i\neq y_j\end{cases}$$

代码亲测：

⊞

```
View Code
```

来自 <http://i.cnblogs.com/EditPosts.aspx?postid=4953865>

活动选择问题 (动态规划、贪心算法)

2016年5月9日　　23:53

问题描述：

设有n个活动的集合E={1,2,…,n}，其中每个活动都要求使用同一资源，如演讲会场等，而在同一时间内只有一个活动能使用这一资源。每个活动i都有一个要求使用该资源的起始时间si和一个结束时间fi,且si <fi。如果选择了活动i，则它在半开时间区间[si, fi)内占用资源。若区间[si, fi)与区间[sj, fj)不相交,则称活动i与活动j是相容的。也就是说，当si≥fj或sj≥fi时，活动i与活动j相容。活动安排问题就是要在所给的活动集合中选出最大的相容活动子集合。

。

活动 i	1	2	3	4	5	6	7	8	9	10	11
开始时间 s	1	3	0	5	3	5	6	8	8	2	12
结束时间 f	4	5	6	7	8	9	10	11	12	13	14

从图中可以看出S中共有11个活动，最大的相互兼容的活动子集为:{a_1，a_4，a_8，a11，}和{a_2，a_4，a_9，a_{11}}。

2、动态规划解决过程

（1）活动选择问题的最优子结构

定义子问题解空间S_{ij}是S的子集，其中的每个获得都是互相兼容的。即每个活动都是在a_i结束之后开始，且在a_j开始之前结束。

$$S_{ij}=\{a_k \in S: f_i \le s_k < f_k \le s_j\}$$

为了方便讨论和后面的计算，添加两个虚构活动a_0和a_{n+1}，其中$f_0=0$，$s_{n+1}=\infty$。

结论：当i≥j时，S_{ij}为空集。

如果活动按照结束时间单调递增排序，子问题空间被用来从S_{ij}中选择最大兼容活动子集，其中0≤i＜j≤n+1，所以其他的S_{ij}都是空集。

最优子结构为：假设S_{ij}的最优解A_{ij}包含活动a_k，则对S_{ik}的解A_{ik}和S_{kj}的解A_{kj}必定是最优的。

通过一个活动a_k将问题分成两个子问题，下面的公式可以计算出S_{ij}的解A_{ij}。

$$A_{ij}=A_{ik}\cup\{a_k\}\cup A_{kj}$$

（2）一个递归解

设c[i][j]为S_{ij}中最大兼容子集中的活动数目，当S_{ij}为空集时，c[i][j]=0；当S_{ij}非空时，若a_k在S_{ij}的最大兼容子集中被使用，则则问题S_{ik}和S_{kj}的最大兼容子集也被使用，故可得到c[i][j] = c[i][k]+c[k][j]+1。

当i≥j时，S_{ij}必定为空集，否则S_{ij}则需要根据上面提供的公式进行计算，如果找到一个a_k，则S_{ij}非空（此时满足$f_i \le s_k$且$f_k \le s_j$），找不到这样的a_k，则S_{ij}为空集。

c[i][j]的完整计算公式如下所示：

$$c[i][j]=\begin{cases}0(if S_{ij}=\varnothing)\\ \max\{c[i][k]+c[k][j]+1\}(i<k<j \text{ 且 } S_{ij}\neq\varnothing)\end{cases}$$

亲测代码：

View Code

下面是贪心法的代码：

View Code

来自 <http://i.cnblogs.com/EditPosts.aspx?postid=4951105>