**P331 7.用回溯法生成{1,2,3,4}的所有排列**

#include<stdio.h>

#include<stdlib.h>

#define N 4 //最大值为4

int a[N+1],b[N+1],count;//b表示每列的只能去下标下标是从1开始取的

void init\_ary()

{

int i;

for(i=1;i<N+1;i++)

{

b[i]=1; //初始值都是1

}

}

void f()

{

int i,j,l;

init\_ary();

i=1;

j=1;

while(1)

{

if(b[1]==N+1)

{

exit(0);

}

if(j==N&&!a[b[j]]) //表示第N列走成功了 输出信息

{

for(l=1;l<N+1;l++)

{

printf("%d ",b[l]);

}

printf("%d\n",++count);

b[j]=1;

j--; //前面的列

a[b[j]]=0;

b[j]++;

}

else

{

if(b[j]==N+1) //向前倒退 因为走到了最下面

{

b[j]=1; //下次b取值时从1开始取

j--;

a[b[j]]=0; //将该行置为可填

b[j]++; //取下一个数代替之

}

else

{

if(!a[b[j]]) //判断以b[j]为下标的a是否被占用j从1开始取值

{

a[b[j]]=1; //标识该行不可以用了

j++; //主要目的就是取得数组b的从1到N的全排列

} //j表示走到了第几列

else

{

b[j]++; //没有匹配向后走

}

}

}

}

}

int main()

{

f();

return 0;

}

**P264 9 a.写一个程序，为给定的英文文本构造一套哈夫曼编码，并对该文本编码**

**b.写一个程序，对一段用哈夫曼编码的英文文本进行解码**

a+b:

#include<iostream>

#include<string>

#include<cstring>

using namespace std;

const int MaxW=9999;

class HuffNode

{

public:

int weight;

char name;

int parent;

int lchild;

int rchild;

};

class HuffMan

{

private:

void MakeTree()

{

int i,s1,s2;

for(i=lnum+1;i<=len;i++)

{

SelectMin(i-1,&s1,&s2);//找出权值最小的两个节点

huffTree[s1].parent=i;//权值最小的两个节点的父节点从n+1开始

huffTree[s2].parent=i;

huffTree[i].lchild=s1;//父节点左右孩子分别是这两个最小的权值的指针值

huffTree[i].rchild=s2;

huffTree[i].weight=huffTree[s1].weight+huffTree[s2].weight;//父节点的权值是这两个最小的节点的权值之和

}

}

void SelectMin(int pos,int \*s1,int \*s2)//找出最小权值的的两个节点，并把他们的指针值赋给s1,s2

{

int w1,w2,i;

w1=MaxW;w2=MaxW;

\*s1=0;\*s2=0;

for(i=1;i<=pos;i++)

{

if(huffTree[i].weight<w1 && huffTree[i].parent==0)//w1比w2小，所以出现有比w1小的权值时，s1的指针值赋给s2,w1的权值赋给w2

{ //一定要注意父节点的值为0是条件，是为了排除已经找出的最小权值的节点重新进行排序

\*s2=\*s1;w2=w1;

\*s1=i;

w1=huffTree[i].weight;//然后将该指针值赋给s1，并把权值赋给w1

}

else if(huffTree[i].weight<w2 && huffTree[i].parent==0)//出现比w1大比w2小的权值时，直接将该指针值赋给s2，并把权值赋给w2

{

\*s2=i;

w2=huffTree[i].weight;

}

}

}

public:

int len;

int lnum;

HuffNode \*huffTree;

string \*huffCode;//存放编码的string指针

void MakeTree(int n,int wt[],char name[])//传入叶子节点的数量，权值，和名字

{

int i;

lnum=n;

len=2\*n-1;//节点的数量是2\*n-1

huffTree = new HuffNode[2\*n];//给结构体指针huffTree分配大小为2\*n大小的空间

huffCode = new string[lnum+1];//给哈夫曼的编码string指针分配n+1de大小空间

for(i=1;i<=n;i++)//注意是下标是从1开始的

{

huffTree[i].weight=wt[i-1];//初始化叶子结点的权值和名字

huffTree[i].name=name[i-1];

}

for(i=1;i<=len;i++)

{

if(i>n) huffTree[i].weight=0;//初始化化其他节点的权值为0

huffTree[i].parent=0;//初始化所有节点的父节点，左右孩子的指针值为0

huffTree[i].lchild=0;

huffTree[i].rchild=0;

}

MakeTree();//初始化其他节点的父节点，左右孩子，和权值

}

void Coding()

{

char \*cd;

int i,c,f,start;

cd=new char[lnum];//存放编码的字符数组

cd[lnum-1]='\0';

for(i=1;i<=lnum;++i)

{

start=lnum-1;//c是自节点，f是父节点，循环就是子节点变为父节点，父节点变为父节点的父节点，依次类推

for(c=i,f=huffTree[i].parent;f!=0;c=f,f=huffTree[f].parent)//先依次找到叶子节点的父节点，然后再判断是父节点的左孩子，还是右孩子

if(huffTree[f].lchild==c) cd[--start]='0';//直到节点的父节点为0

else cd[--start]='1';

huffCode[i].assign(&cd[start]);//将字符编码存放到字符编码的指针

}

delete []cd;

}

void display(char str[])

{

int len=strlen(str);

for(int i=0;i<len;i++)

{

for(int j=0;j<=lnum;j++)

{

if(str[i]==huffTree[j].name)

cout<<huffCode[j];

}

}

cout<<endl;

}

int Decode(const string codestr,char txtstr[])

{

int i,k,c;

char ch;

c=len;//从最大的父节点开始

k=0;

for(i=0;i<codestr.length();i++)

{

ch=codestr[i];

if(ch=='0') c=huffTree[c].lchild;//如果是0的话，子节点变为当前节点的左节点

if(ch=='1') c=huffTree[c].rchild;//如果是1的话，子节点变为当前节点的右节点

if(ch!='0'&&ch!='1') return -1;

if(huffTree[c].lchild==0 && huffTree[c].rchild==0)//当找到叶子节点时候

{

txtstr[k]=huffTree[c].name;//将叶子节点的名字赋给解编码的数组

k++;

c=len;//每次找到一个叶子节点，都要回到最大的父节点

}

else ch='\0';

}

if(ch=='\0') return -1;

else txtstr[k]='\0';

return 1;

}

void Destroy()

{

len=0;

lnum=0;

delete []huffTree;

delete []huffCode;

}

};

int main()

{

int t,n,i,j;

char name[100];

int wt[800];

string str;

char Name[1000];

HuffMan myHuff,myHuff1;

cin>>t;

for(i=0;i<t;i++)

{

cin>>n;

for(j=0;j<n;j++)

cin>>name[j];

for(j=0;j<n;j++)

cin>>wt[j];

myHuff.MakeTree(n,wt,name);

myHuff.Coding();

cin>>Name;

cin>>str;

char ch[800];

int m=myHuff.Decode(str,ch);

for(j=1;j<=n;j++)

{

cout<<myHuff.huffTree[j].name<<" :";

cout<<myHuff.huffCode[j]<<endl;

}

myHuff.display(Name);

if(m==-1) cout<<"error!"<<endl;

if(m==1) cout<<ch<<endl;

myHuff.Destroy();

}

return 0;

}

**c.做一个实验，测试对包含1000个词的一段英文文本进行哈夫曼编码时，典型的压缩率位于什么样的区间**

**d.对编码程序做一个实验，测试如果用标准的估计频率代替英文文本中字符的实际出现频率，该程序的压缩率会有什么样的变化**

**p338 7.写一个程序用分支界限算法对背包问题求解**

#include<stdio.h>

#include<math.h>

typedef struct {

int no; // 结点标号

int id; // 节点id

int sw; // 背包中物品的重量

int sv; // 背包中物品的价值

}Node;

void branchknap(int \*w,int \*v,int n,int c) {

int bestv = 0;

int f = 0;

int r = 0;

Node que[3000];

int path[15];

que[0].no = 1;

que[0].id = que[0].sv = que[0].sw = 0;

while(f <= r) {

Node node = que[f];

printf("%d %d %d %d\n",node.id+1,node.no,node.sw,node.sv);

if(node.no >= pow(2,n)) {

if(node.sv > bestv) {

bestv = node.sv;

printf("bestv=%d, bestx=[",bestv);

int temp = node.no;

int i = 0;

while(temp > 1) {

if(temp % 2 == 0)

path[i] = 1;

else

path[i] = 0;

temp /= 2;

i++;

}

i--;

while(i >= 0) {

printf(" %d",path[i]);

i--;

}

printf(" ]\n");

}

} else {

if((node.sw + w[node.id + 1]) <= c) {

r++;

que[r].id = node.id + 1;

que[r].no = node.no \* 2;

int id = node.id + 1;

que[r].sv = node.sv + v[id];

que[r].sw = node.sw + w[id];

}

r++;

que[r].id = node.id + 1;

que[r].no = node.no \* 2 + 1;

que[r].sv = node.sv;

que[r].sw = node.sw;

}

f++;

}

}

int main() {

int c,n;

int w[15],v[15];

while(~scanf("%d %d",&c,&n)) {

for(int i = 1; i <= n; i++) {

scanf("%d %d",&w[i],&v[i]);

}

branchknap(w,v,n,c);

}

return 0;

}

**P249 7.谣言传播 有n个人，每个人都拥有不同的谣言。通过发电子信息，他们想互相共享所有的谣言。假定发送者会在信息中心包含他已知的所有谣言，而且一条信息只有一个收信人。设计一个贪心算法，保证在每个人都能获得所有谣言的条件下使发送的信息数最小**

解答：将这n个人标记为1, 2, …, n，按照1发信给2, 2发信给3, 3发信给4，…，n-1发信给n的方式发送谣言，该贪心算法基于每次发信都使得当前收信人掌握的谣言更多，最后由n将所有谣言发送给其他n-1个人。

发送信息总数为2n-2，这是最小的发信息数。因为每增加一个人，至少需要增加两次发送信息，当n=2是，发送信息数为2，归纳法可证明2n-2为最小发信息数。

**P234 11**

**11.矩阵连乘 考虑如何使得在计算n个矩阵的乘积A1A2...An时，总的乘法次数最小，这些矩阵的维度分别为d0×d1、d1×d2、dn-1×dn。假设所有两个矩阵的中间乘积都使用蛮力算法（基于定义）计算。**

**a.给出一个三个矩阵连乘的例子，当分别用（A1A2）A3和A1(A2A3)计算时，它们的乘法次数至少相差1000倍。**

例子：计算三个矩阵连乘{A1，A2，A3}；维数分别为10\*100 , 100\*5 , 5\*50 按此顺序计算需要的次数((A1\*A2)\*A3):10X100X5+10X5X50=7500次，按此顺序计算需要的次数(A1\*(A2\*A3)):100\*5\*50+10\*100\*50=52500次

**b.有多少种不同的方法来计算n个矩阵的连乘乘积？**

穷举法、重叠递归、备忘录递归算法、动态规划方法

**c.设计一个求n个矩阵乘法最优次数的动态规划算法。**

int ra,ca;//矩阵A的行数和列数

int rb,cb;//矩阵B的行数和列数

void matrixMultiply()

{

for(int i=0;i<ra;i++)

{

for(int j=0;j<cb;j++)

{

int sun=0;

for(int k=0;k<=ca;k++)

{

sum+=a[i][k]\*b[k][j];

}

c[i][j]=sum;

}

}

}

**P229 3**

**3.对于背包问题的自底向上动态规划算法，请证明：**

**a.它的时间效率属于θ（nW）**

**b.它的空间效率属于θ（W）**

**c.从一张填好的动态规划表中求得最优子集的组合所用的时间属于O（n）**

a+c:使用动态规划，目标是书包内物品的总价值，而变量是物品和书包的限重，所以可定义状态dp:dp[i][j]表示将前i件物品装进限重为j的背包可以获得的最大价值, 0<=i<=N, 0<=j<=W

那么可以将dp[0][0...W]初始化为0，表示将前0个物品（即没有物品）装入书包的最大价值为0。那么当 i > 0 时dp[i][j]有两种情况：

1. 不装入第i件物品，即dp[i−1][j]；  
   2.装入第i件物品（前提是能装下），即dp[i−1][j−w[i]] + v[i]。  
   即状态转移方程为  
   dp[i][j] = max(dp[i−1][j], dp[i−1][j−w[i]]+v[i]) // j >= w[i]  
   由上述状态转移方程可知，dp[i][j]的值只与dp[i-1][0,...,j-1]有关，所以可以采用动态规划常用的方法（滚动数组）对空间进行优化（即去掉dp的第一维）。需要注意的是，为了防止上一层循环的dp[0,...,j-1]被覆盖，循环的时候 j 只能逆向枚举（空间优化前没有这个限制），伪代码为：

dp[0,...,W] = 0  
for i = 1,...,N  
    for j = W,...,w[i] // 必须逆向枚举  
        dp[j] = max(dp[j], dp[j−w[i]]+v[i])

时间复杂度为O(nW), 空间复杂度为O(W)。

C:每次对于n求出n-1到n时的长度变化，最后得出最长的子集长度，遍历一次即可，算法时间复杂度为O(n)

**P225 6**

1. **切割木棍问题 为下列问题设计一个动态规划算法。已知小木棍的销售价格pi和长度i相关，i=1,2,...,n,如何把长度为n的木棍切割为若干根长度为整数的小木棍，使得所能获得的总销售价格最大？该算法的时间效率和空间效率各是多少？**

解：

长度为n的最大价值 price(n)=MAX(price(i)+price(n-i))

长度为n的价格有两种 第一种：原始长度为n时的价格 第二种：加n分割为个小块 再加起来的价格

设长度1~n长度的木棍价格为p[1…n]

先从最短的长度 1开始找相对应长度可得到的最大价值，因为长度1无法再分，所以maxprice[1] 就为原始长度价格 p[1]

然后长度2的可得到的最大价值maxprice[2]就为maxprice[1] +maxprice[1] 和 p[2]之中最大的那个

3的可得到的最大价值maxprice[3]就为 maxprice[1]+maxprice[2] 、maxprice[2]+maxprice[1]和p[3]中最大的那个（**为什么没有maxprice[1]+maxprice[1]+maxprice[1]？因为1+1=2，长度为2的最大价格maxprice[2]已知，所以不需要再把2分为1+1了，而1+2=3、2+1=3时的长度3的maxprice[3]的价格才是不知道的。也就是每个长度只需要分为两部分就行了**）

因为比当前长度小的所有整数长度的对应的最大价格都是已知的，所以长度为n时只需要找到maxprice[1]+maxprice[n-1]、maxprice[2]+maxprice[n-2]、…、maxprice[i]+maxprice[n-i]、…、maxprice[n-1]+maxprice[1]、p[n]中最大的值，再赋值给maxprice[n]

此算法的时间效率是O(n^2)，空间效率是O (N)

代码：

#include <iostream> using namespace std; //自底向上，两个循环，不用递归； int main() {

int n = 10;

int price[11] = { 0, 1, 7, 8, 9, 10, 17, 17, 20, 23, 24 };

int \*r = new int[n + 1];

for (int i = 0; i <= n; ++i) //r[i]为长度为i时的价值

r[i] = 0; //初始化

for (int i = 1; i <= n; ++i)//规模长度为i

{

int q = INT\_MIN;

for (int j = 1; j <= i; ++j)//计算规模为i的最大收益

{

if (q < (price[j] + r[i - j]))//因为i>i-j，所以当计算r[i]时，r[i-j]已经解决，可以直接用

q = (price[j] + r[i - j]); //迭代q；

}

r[i] = q; //找出i这个位置的最优解；

}

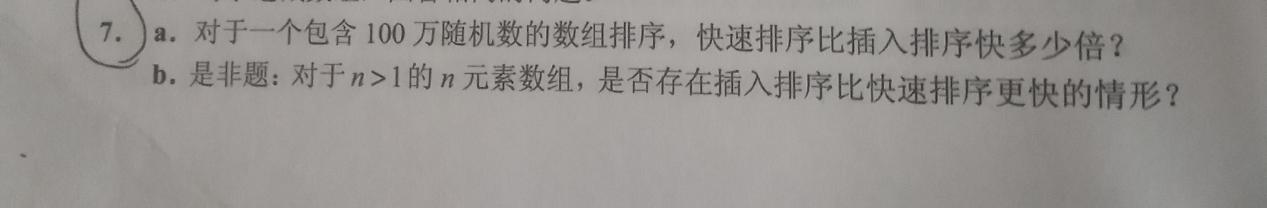
cout << r[n]; //最后是n这个位置，就是n米长的木头的最大价值。

cout <<endl <<INT\_MIN;

delete r;

return 0; }

P140 7



1. 快速排序执行107次，插入排序执行1012次，快速排序比插入排序快105倍。
2. 不存在。我们在对二元数组进行排序是，可以将数组的第一个数字看成是一个有序的数组，而后面的数字是一个无序的数组。遍历后面的无序数组和前面有序的数组进行比较，当发现需要插入时，有序数组向后挪移一位，插入数据。时间复杂度为O（n2）,通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列，时间复杂度nlog(n)。插入排序的效率要比快速排序的效率低。