140、7

a. 对于一个包含100万随机数的数组排序，快速排序比插 入排序快多少倍？

答：1000000=106

插入排序平均效率是n2，快速排序的平均效率是1.39\*n\*log2 n。

计算n2/n\*log2 n=1012/1.39\*106 \*log2 106≈36000倍

b. 是非题：对于n>1的n元素数组，是否存在插入排序比快速排序 更快的情形？

答：插入排序：当n元数组是最优输入，即有序数组的输入效率是n。

快速排序：对最优输入，效率是n\*log2n。

当n=2时，两种排序理论上一样快。当n>2时，插入排序比快速 排序快。所以存在插入排序比快速排序更快的情形。

225、6

切割木棍问题

对于长度为n的木棍，他的递推关系是：profit[n] = max(pi[i] + profit[length - seg[i]]), 其中i = 1,2,3,...n; 切割长度（seg） 销售价格（pi）

解决的主要思路是，先求解长度为1的最大收益，再到2,3.....一直到n，而每次求解的解都储存起来，时间复杂度为O(nm)，空间复杂度为O(n)

#include <iostream>

using namespace std; //自底向上，两个循环，不用递归；

int main()

{

int i=0;

int n = 10;

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

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

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

r[i] = 0; //初始化

for (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；

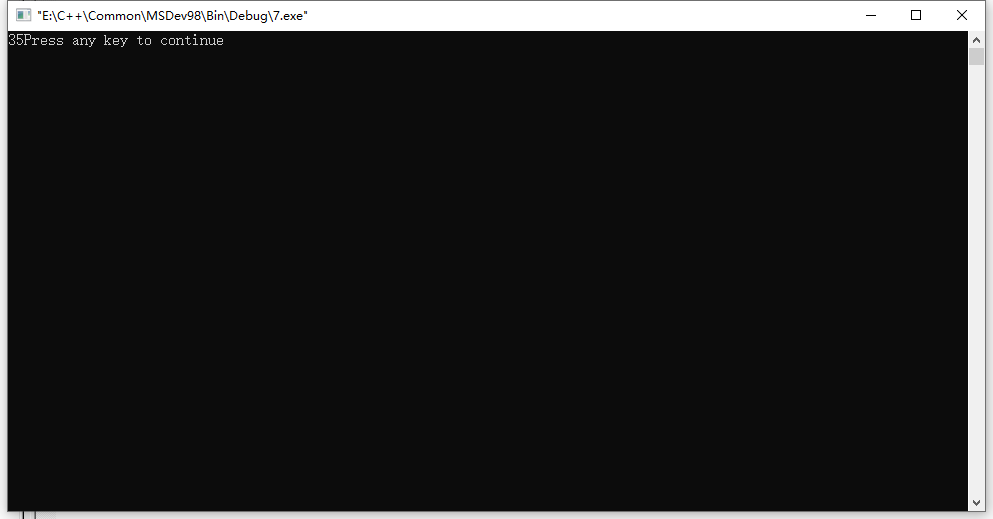
}

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

delete r;

return 0;

}



229、3

对于背包问题的自底向上动态规划算法，请证明：

（1）它的时间效率属于O(nW)。

（2）它的空间效率属于O(nW)。

（3）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)。

答：01背包问题的动态规划解法递归方程为：

F(i, j) = max { F(i-1, j), F(i-1, j-wi) + vi }; j >= wi

F(i, j) = F(i-1, j) j < wi

伪代码

function BeiBao()

for i ⟵ 0 to n do

K[i][0] ⟵ 0

for j ⟵ 1 to W do

K[0][j] ⟵ 0

for i ⟵ 1 to n do

for j ⟵ 1 to W do

if j < wi then

K[i][j] ⟵ K[i-1][j]

else

K[i][j] ⟵ max(K[i-1][j], K[i-1][j-wi]+vi)

return K[n][W]

通过递归方程和伪代码计算F(i, j)值花费常量时间。

（1）（2）显然算法空间复杂度与时间复杂度均为Θ（nW）。其中W为背包容量

（3）为了确定一个最优子集的组成，该算法重复比较前一行中不超过两个单元格的值。因此，其时间效率类为O(n)。因为基本操作是比较“j < wi”。它被执行了n\*w次。

234、11

矩阵连乘 考虑如何使得在计算n个矩阵的乘积A1,A2...An时，总的乘法次数最小，这些矩阵的维度分别为d0\*d1\*,d1\*d1,...,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

1. 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，他们的乘法次数至少相差1000倍

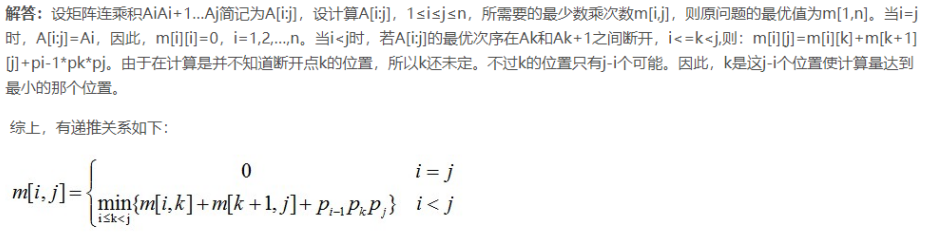
解答：矩阵维数A1为1000\*1，A2为1\*1000，A3为1000\*1

b、有多少种不同的算法来计算n个矩阵的连乘乘积

解答：算法数递推公式为



c、设计一个求n个矩阵乘法最优次数的动态规划算法



249、7

有n个人，每个人都拥有不同的谣言。通过发电子信息，他们相互想共享所有的谣言。假定发送者会在信息中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，保证再给个人都能获得所有谣言的条件下，是发送的信息数最小。

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。

代码如下:

#include<bits/stdc++.h>

using namespace std;

int v[101][101],d[101][101];

int t[101];

int n,m,x,y,N;

int main()

{

memset(v,63,sizeof(v));//预处理

scanf("%d",&n);

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

{ scanf("%d",&N);

for(int j=1;j<=N;j++)

{ scanf("%d%d",&x,&y);

v[i][x]=y;

}

}

for(int k=1;k<=n;k++)

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

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

{ if(v[i][k]+v[k][j]<v[i][j])

v[i][j]=v[i][k]+v[k][j];

}//搜索最短路径

int minn=1e9,sum;

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

{ int maxx=0;

for(int k=1;k<=n;k++)

if(v[i][k]>maxx&&i!=k)

maxx=v[i][k];//找出最晚得知消息的权值，若不能到达则maxx自动赋值为最大值

if(maxx<minn) minn=maxx,sum=i;//与当前最短路径作比较

}

if(minn==1e9)

printf("disjoint");//若搜完后发现不能遍历整个图，则输出"disjoint"

else

printf("%d %d",sum,minn);//否则输出最短路径及开始点

}

264、9

a.写一个程序,为给定的英文文本构造一套哈夫曼编码,并对该哈夫曼编码.

b.写一个程序,对一段用哈夫曼编码的英文文本进行解码.

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的字符的实际出现频率,该程序的压缩率会出现什么样的变化.

#include <iostream>

#include <fstream>

#include <queue>

#include <string>

#include <algorithm>

using namespace std;

class Node

{

public:

char mchar;

int mweight;

Node \*lchild,\*rchild;

Node()

{

mchar='!'; //不是叶子结点就叹号表示

lchild=rchild=NULL;

}

Node(char c,int w)

{

mchar=c;

mweight=w;

lchild=rchild=NULL;

}

//Node(Node &n)

//{

// mchar=n.mchar;

// mweight=n.mweight;

// lchild=n.lchild;

// rchild=n.rchild;

//}

};

bool operator <(Node a,Node b)

{

return a.mweight>b.mweight;

}

class HuffmanTree

{

public:

long mbitlenlastcoding;

int mSumBitLen;

Node \*root;

//queue头中自带有优先队列.如此定义可以小的先出队.注意要如此重载<运算符

priority\_queue<Node> H;

vector<Node\*> V;

vector<string> CodeTable;

HuffmanTree( )

{

mbitlenlastcoding=0;

mSumBitLen=0;

root=NULL;

}

void CreateTree(int \*farray,string chartable,int num)

{

SetQueueH(farray,chartable,num);

for(int k=0;k<num-1;k++)

{

Node \*i=new Node(H.top());

//auto\_ptr<Node> api(i);

H.pop();

Node \*j=new Node(H.top());

//auto\_ptr<Node> apj(j);

H.pop();

Node \*fk=new Node;

//auto\_ptr<Node> apfk(fk);

fk->lchild=i;

fk->rchild=j;

fk->mweight=i->mweight + j->mweight;

H.push(\*fk);

root=fk; //运算到第n-1次时.FK就是root

}

LinearizeTree(); //线性存储huffman树

string str(" ");

CreateCodeTable(root,str,-1); //创建编码表

sort(CodeTable.begin(),CodeTable.end()); //对编码表按字母序排序.方便压缩时查找

}

void Coding(const char\* str) //编码

{

ofstream fout;

fout.open("text.dat",ios::binary);

int bit=0;

int bitcount=0;

for(int i=0;i<strlen(str);i++)

{

int j=0;

if(str[i]>='A' && str[i]<='Z')

j=str[i]-'A'+1;

else if(str[i]>='a' && str[i]<='z')

j=str[i]-'a'+1;

else if(' '==str[i])

j=0;

string code(CodeTable[j].substr(2));

mbitlenlastcoding+=code.length(); //保存实际编码的位数,供解压缩使用;

for(int k=0;k<code.length();k++)

{

if('0'==code.at(k)) //按序把编码放到高位.也可以放到低位

bit=bit<<1;

else

bit=(bit<<1)+1; // 放低位,此步改为与0x80000000相或,bit=(bit|0x80000000);

bitcount=(bitcount+1)%32;

if(0==bitcount)

fout.write((char\*)&bit,sizeof(bit));

}

}

if(0!=bitcount) //将最后不满32位的位移动到最高位

{

bitcount=32-bitcount;

bit=bit<<bitcount;

fout.write((char\*)&bit,sizeof(bit));

}

fout.close();

}

void Decodeing(char \*infilename,char \*outfilename) //解码

{

ifstream fin;

fin.open(infilename,ios::binary);

if(true==fin.fail())

{

cerr<<"file read fail"<<endl;

return;

}

bool overflag=false;

long bitcount=0;

int buffsize=sizeof(int)\*8;

char buff[sizeof(int)\*8+1]; //+1用来保存串结束符

int bit;

Node \*p=root;

string strResult;

while(!overflag)

{

fin.read((char\*)&bit,sizeof(bit));

mbitlenlastcoding=mbitlenlastcoding-buffsize;

\_itoa(bit,buff,2);

//因为\_itoa函数转换二进制数时.若前几位为零时,不会转换位零,要自行添加

string str(buff);

str.insert(0,buffsize-strlen(buff),'0');

if(mbitlenlastcoding<0)

{

mbitlenlastcoding=buffsize+mbitlenlastcoding;

str=str.substr(0,mbitlenlastcoding);

buffsize=str.length();

overflag=true; //置结束标志

}

int i=0;

while(i<buffsize )

{

if('!' !=p->mchar)

{

strResult.push\_back(p->mchar);

p=root;

}

else

{

if('0'==str.at(i++))

p=p->lchild;

else

p=p->rchild;

}

}

}

// cout<<strResult<<endl;

ofstream fout(outfilename);

fout<<strResult;

fout.close();

fin.close();

}

void ShowQueueH()

{

while(!H.empty())

{

Node t=H.top();

H.pop();

cout<<t.mchar<<" "<<t.mweight<<endl;

}

}

void ShowTree()

{

for(int i=0;i<V.size();i++)

cout<<i<<" "<<V[i]->mchar<<" "<<V[i]->mweight<<endl;

}

void ShowCodeTable()

{

for(int i=0;i<CodeTable.size();i++)

cout<<CodeTable.at(i)<<endl;

}

void PrintCodeTable(char \*filename)

{

ofstream fout(filename);

for(int i=0;i<CodeTable.size();i++)

fout<<CodeTable[i]<<endl;

fout.close();

}

~HuffmanTree()

{

for(int i=0;i<V.size();i++)

{

if(NULL!=V[i])

delete V[i];

}

}

private:

void SetQueueH(int \*farray,string chartable,int num)

{

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

{

Node t(chartable.at(i),farray[i]);

H.push(t);

}

}

void LinearizeTree() //线性存储huffman树

{

queue<Node> S;

S.push(\*root);

V.push\_back(root);

while(!S.empty())

{

Node t=S.front();

S.pop();

if(NULL != t.lchild)

{

S.push( (\*t.lchild) );

V.push\_back(t.lchild);

}

if(NULL != t.rchild)

{

S.push( (\*t.rchild) );

V.push\_back(t.rchild);

}

}

}

void CreateCodeTable(Node \*p,string str,int flag)

{

if (NULL == p ) return ;

if(0==flag) str.push\_back('0');//str.append("0");

else if(1==flag) str.push\_back('1');//str.append("1");

if('!' != p->mchar)

{

mSumBitLen+=(str.length()-1); //加上编码的长度

str.insert(str.begin(),p->mchar);

CodeTable.push\_back(str);

return;

}

else

{

if(NULL!=p->lchild)

CreateCodeTable(p->lchild,str,0);

if(NULL!=p->rchild)

CreateCodeTable(p->rchild,str,1);

}

}

};

void main()

{

string chartable(" etaoinshrdlcumwfgypbvkjxqz");

int farray[]={183,102,77,68,59,58,55,51,49,48,35,34,26,24,21,19,18,17,16,16,13,9,6,2,2,1,1};

char text[]="Chapter Graphs surveys the most important graph processing problems including depth first search breadth first search minimum spanning trees and shortest paths ";

HuffmanTree ht;

ht.CreateTree(farray,chartable,chartable.length());

ht.ShowTree();

ht.ShowCodeTable();

ht.PrintCodeTable("CodeTable.txt");

ht.Coding(text);

ht.Decodeing("text.dat","Decoding.txt");

cout<<"平均编码长度:"<<ht.mSumBitLen/27.0<<endl;

}

331、7

用回溯法生成{1，2，3，4}的所有排列

代码如下：

#include<iostream>

#define N 10

using namespace std;

int main(){

int t=1;

int n;//n为给出的一串数中的最大数

cin>>n;

int m;//m为给出的位数

cin>>m;

int a[N];

a[t]=0;

do{

a[t]=a[t]+1;

if(a[t]>n){//超过最大值，回溯

a[t]=0;

t--;

}else{

if(t==m){//已到最后一位，找到一种排列，则输出

for(int i=1;i<=m;i++)

cout<<a[i]<<" ";

cout<<endl;

}else{//继续向前搜索

t++;

a[t]=0;

}

}

}while(t!=0);//do..while的意思是当满足括号里面的条件时，一直执行函数体，最后还有个;

return 0;

}

338、7

写一个程序用分支界限算法对背包问题求解

代码如下：

#include<iostream>

#include<stack>

using namespace std;

#define N 100

class

HeapNode //定义HeapNode结点类

{

public:

double upper,price,weight;//upper为结点的价值上限，price是结点所对应的价值，weight为结点所相应的重量

int level,x[N];//活节点在子集树中所处的层序号

};

double MaxBound(int i);

double Knap();

void AddLiveNode(double up,double cp,double cw,bool ch,int level);

stack<HeapNode>

High;//最大队High

double w[N],p[N];//把物品重量和价值定义为双精度浮点数

double cw,cp,c=30;//cw为当前重量，cp为当前价值，定义背包容量为30

int n=3;//货物数量为3

int main()

{

cout<<"请按照顺序输入3个物品的重量："<<endl;

int i;

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

cin>>w[i];//输入3个物品的重量

cout<<"请按顺序输入3个物品的价值："<<endl;

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

cin>>p[i];//输入3个物品的价值

cout<<"最大价值为：";

cout<<Knap()<<endl;//调用Knap函数，输出最大价值

return 0;

}

double MaxBound(int j)//MaxBound函数求最大上限

{

double

left=c-cw,b=cp;//剩余容量和价值上限

while(j<=n&&w[j]<=left)//以物品单位重量价值递减装填剩余容量

{

left-=w[j];

b+=p[j];

j++;

}

if(j<=n)

b+=p[j]/w[j]\*left;//装填剩余容量装满背包

return b;

}

void AddLiveNode(double up,double cp,double cw,bool ch,int lev)//将一个新的活结点插入到子集数和最大队High中

{

HeapNode be;

be.upper=up;

be.price=cp;

be.weight=cw;

be.level=lev;

if(lev<=n)

High.push(be);//调用stack头文件的push函数

}

double Knap()//优先队列分支界限法，返回最大价值，bestx返回最优解

{

int i=1;cw=cp=0;double bestp=0,up=MaxBound(1);//调用MaxBound求出价值上限，best为最优值

while(1)//非叶子结点

{

double wt=cw+w[i];

if(wt<=c)//左儿子结点为可行结点

{

if(cp+p[i]>bestp) bestp=cp+p[i];

AddLiveNode(up,cp+p[i],cw+w[i],true,i+1);

}

up=MaxBound(i+1);

if(up>=bestp)//右子树可能含最优解

AddLiveNode(up,cp,cw,false,i+1);

if(High.empty()) return bestp;

HeapNode node=High.top();//取下一扩展结点

High.pop();

cw=node.weight;

cp=node.price;

up=node.upper;

i=node.level;

}

}

