**P140**

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：插入比快速排序时间快10倍，忽略常数、误差的平均情况中，快速排序执行约10^7次，插入排序执行约10^12次，大约十万倍

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：存在

**P225**

6.**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p;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?

答：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已知，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)

#include <iostream>

#include <cstring>

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

void main()

{

int n,i;

cin>>n;

int price[11]={0,1,5,8,9,10,17,17,20,23,28};

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。

}

**P229**

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 (nW)。

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)。

该算法用n+1行和W+1列填充一个表格，用Θ（1）时间填充一个单元格。因此，其时间效率和空间效率均在Θ（nW）之间。  
为了确定最佳子集的组成，该算法反复比较前一行中不超过两个单元格的值。因此，其时间效率等级为O（n）。

**P234**

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 x d1 , d1 x d2, .... dn-1 x 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(A1 A2)A3和A1（A2A3）计算时，它们的乘法次数至少相差1 000倍。

答：用基于定义的算法将两个维数为α-β和β-γ的矩阵相乘需要αβγ乘法。（产品中有αγ元素，每个元素都需要计算β乘法。）如果A1、A2和A3的尺寸分别为d0\*d1、d1\*d2和d2\*d3，则（A1\*A2）\*A3需要 d0d1d2+d0d2d3=d0d2（d1+d3）乘法，

而 A1\*（A2\*A3）

需要 d1d3+d0d1d3=d1d3（d0+d2）

乘法。这里有一个简单的特定值选择，例如，第一个值比第二个大1000倍：

d0=d2=103，d1=d3=1。

b.有多少种不同的方法来计算n个矩阵的连乘乘积?

答：设m（n）是计算n个矩阵A1......An的链积的不同方法的数目。链的任何圆括号都将导致前k个矩阵（A1......Ak）和最后n−k矩阵（Ak+1......An）的某些乘积相乘，作为最后一个操作。前者有m（k）种方法，后者有m（n−k）种方法。因此，对于n个矩阵的矩阵链加括号的方法总数，我们有以下递归：

n-1

m(n)= ∑ m(k)m(n−k) for n>1,m(1) = 1.

k=1

因为构造n的递归链与构造n的递归链非常相似，因为它与构造n的递归链非常相似

n-1

b(n)= ∑ b(k)b(n−1−k) for n>1,b(0) = 1,

k=0

对于第8.3节中提到的二叉树的数量。它们的解也很相似，也就不足为奇了： m（n）=b（n−1）n≥1，

其中b（n）是具有n个节点的二叉树的数目。让我们用数学归纳法来证明这个主张。基础立即检查：m（1）=b（0）=1。对于一般情况，让我们假设m（k）=b（k−1）对于不超过某个正整数n的所有正整数（我们使用数学归纳法的强版本）；我们将证明对于n+1的等式成立。的确，

n

m（n+1）=∑ m（k）m（n+1−k）

k=1

n

=[使用归纳法的假设]∑ b（k−1）b（n−k）

k=1

n−1

=[代入l=k−1]∑ b（l）b（n−1−l）

l=0

=[见b（n）的递归] b（n）。

c.设计一个求n个矩阵乘法最优次数的动态规划算法。

答：

//3d1-2 矩阵连乘 动态规划迭代实现

//A1 30\*35 A2 35\*15 A3 15\*5 A4 5\*10 A5 10\*20 A6 20\*25

//p[0-6]={30,35,15,5,10,20,25}

#include <iostream>

using namespace std;

const int L = 7;

int MatrixChain(int n,int \*\*m,int \*\*s,int \*p);

void Traceback(int i,int j,int \*\*s);//构造最优解

int main()

{

int p[L]={30,35,15,5,10,20,25};

int \*\*s = new int \*[L];

int \*\*m = new int \*[L];

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

{

s[i] = new int[L];

m[i] = new int[L];

}

cout<<"矩阵的最少计算次数为："<<MatrixChain(6,m,s,p)<<endl;

cout<<"矩阵最优计算次序为："<<endl;

Traceback(1,6,s);

return 0;

}

int MatrixChain(int n,int \*\*m,int \*\*s,int \*p)

{

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

{

m[i][i] = 0;

}

for(int r=2; r<=n; r++) //r为当前计算的链长（子问题规模）

{

for(int i=1; i<=n-r+1; i++)//n-r+1为最后一个r链的前边界

{

int j = i+r-1;//计算前边界为r，链长为r的链的后边界

m[i][j] = m[i+1][j] + p[i-1]\*p[i]\*p[j];//将链ij划分为A(i) \* ( A[i+1:j] )

s[i][j] = i;

for(int k=i+1; k<j; k++)

{

//将链ij划分为( A[i:k] )\* (A[k+1:j])

int t = m[i][k] + m[k+1][j] + p[i-1]\*p[k]\*p[j];

if(t<m[i][j])

{

m[i][j] = t;

s[i][j] = k;

}

}

}

}

return m[1][L-1];

}

void Traceback(int i,int j,int \*\*s)

{

if(i==j) return;

Traceback(i,s[i][j],s);

Traceback(s[i][j]+1,j,s);

cout<<"Multiply A"<<i<<","<<s[i][j];

cout<<" and A"<<(s[i][j]+1)<<","<<j<<endl;

}

**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。

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。

**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，对一段用哈夫曼码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字符的实际出现频率，该程序的压缩率会有什么样的变化。

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<queue>

using namespace std;

typedef struct node{

char ch; //存储该节点表示的字符，只有叶子节点用的到

int val; //记录该节点的权值

struct node \*self,\*left,\*right; //三个指针，分别用于记录自己的地址，左孩子的地址和右孩子的地址

friend bool operator <(const node &a,const node &b) //运算符重载，定义优先队列的比较结构

{

return a.val>b.val; //这里是权值小的优先出队列

}

}node;

priority\_queue<node> p; //定义优先队列

char res[30]; //用于记录哈夫曼编码

void dfs(node \*root,int level) //打印字符和对应的哈夫曼编码

{

if(root->left==root->right) //叶子节点的左孩子地址一定等于右孩子地址，且一定都为NULL;叶子节点记录有字符

{

if(level==0) //“AAAAA”这种只有一字符的情况

{

res[0]='0';

level++;

}

res[level]='\0'; //字符数组以'\0'结束

printf("%c=>%s\n",root->ch,res);

}

else

{

res[level]='0'; //左分支为0

dfs(root->left,level+1);

res[level]='1'; //右分支为1

dfs(root->right,level+1);

}

}

void huffman(int \*hash) //构建哈夫曼树

{

node \*root,fir,sec;

for(int i=0;i<26;i++) //程序只能处理全为大写英文字符的信息串，故哈希也只有26个

{

if(!hash[i]) //对应字母在text中未出现

continue;

root=(node \*)malloc(sizeof(node)); //开辟节点

root->self=root; //记录自己的地址，方便父节点连接自己

root->left=root->right=NULL; //该节点是叶子节点，左右孩子地址均为NULL

root->ch='A'+i; //记录该节点表示的字符

root->val=hash[i]; //记录该字符的权值

p.push(\*root); //将该节点压入优先队列

}

//下面循环模拟建树过程，每次取出两个最小的节点合并后重新压入队列

//当队列中剩余节点数量为1时，哈夫曼树构建完成

while(p.size()>1)

{

fir=p.top();p.pop(); //取出最小的节点

sec=p.top();p.pop(); //取出次小的节点

root=(node \*)malloc(sizeof(node)); //构建新节点，将其作为fir，sec的父节点

root->self=root; //记录自己的地址，方便该节点的父节点连接

root->left=fir.self; //记录左孩子节点地址

root->right=sec.self; //记录右孩子节点地址

root->val=fir.val+sec.val;//该节点权值为两孩子权值之和

p.push(\*root); //将新节点压入队列

}

fir=p.top();p.pop(); //弹出哈夫曼树的根节点

dfs(fir.self,0); //输出叶子节点记录的字符和对应的哈夫曼编码

}

int main()

{

char text[100];

int hash[30];

memset(hash,0,sizeof(hash)); //哈希数组初始化全为0

printf("请输入想要编码的大写字符:\n");

scanf("%s",text); //读入信息串text

for(int i=0;text[i]!='\0';i++)//通过哈希求每个字符的出现次数

{

hash[text[i]-'A']++; //程序假设运行的全为英文大写字母

}

huffman(hash);

return 0;

}

**P331**

7. 用回溯法生成{1, 2, 3, 4}的所有排列。

一、题目：编写一个输出1,2,3…,n,n个数字所组成的所有排列.

二、分析题意：

第一步，先思考一些例子。

例如:n=3 ,则全排列为：

111 112 113 -> 121 122 123 -> 131 132 133

211 212 213 -> 221 222 223 -> 231 232 233

311 312 313 -> 321 222 223 -> 331 332 333

第二步，分步思考，若仅输出111、112、113：

很容易想到这样：

t=1; a[t]=0;

do{

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

if(t==3) // 找到一种排列，则输出

输出a[1],a[2],a[3]

else

{ ++t; a[t]=0; } //继续向前搜索

}while(a[3]!=3);

第三步，思考如何从 113 过渡到 121？

想到：从113 ->114 当最后一位超过最大数字时，无效。下组数字的最后位要“归零”，同时回溯到前一位“进位”。

发现规律：

（1）每组数中当第t位超过最大值时，下组数的此位要“归零”，同时回溯到 t-1 位“进位”，由此前进，回溯，再前进。

（2）由此，直到回溯到第 t=0 位结束。

三、源代码：

#include<iostream>

#define N 10

#include<cstring>

using namespace std;

int a[N];

int t=1;

int n;//n为给出的一串数中的最大数

int m;//m为给出的位数

//常用的非递归迭代过程

void Iter\_backtrack(){

//第一步，先把每个数组都初始化为0

memset(a,0,sizeof(a));

//第二步，非递归迭代过程

t=1;

while(t!=0){//1.先想到回溯的条件，当回溯到第0位结束

//2.再想什么时候回溯，什么时候继续前进

if(a[t]<n){//这种情况下不回溯

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

if(a[t]<=n){

if(t==m){

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

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

cout<<endl;

}else t++;

}

}else{//这种情况下回溯

a[t]=0;//这个与上面那个回溯规律是一样的

t--;

}

}

}

int main(){

printf("一串数中的最大数为：");

scanf("%d",&n);

printf("位数为：");

scanf("%d",&m);

Iter\_backtrack();

return 0;

}

**P338**

1. **写一个程序用分支界限算法对背包问题求解。**

**beibao.h文件代码如下**

#ifndef BEIBAO\_H

#define BEIBAO\_H

#include <math.h>

//子空间中节点类型

class BBnode{

public:

BBnode\* parent; //父节点

bool leftChild; //左儿子节点标志

BBnode(BBnode\* par,bool ch){

parent=par;

leftChild=ch;

}

BBnode(){

}

};

class HeapNode {

public:

BBnode\* liveNode; // 活结点

double upperProfit; //结点的价值上界

double profit; //结点所相应的价值

double weight; //结点所相应的重量

int level; // 活结点在子集树中所处的层次号

//构造方法

HeapNode(BBnode\* node, double up, double pp , double ww,int lev){

liveNode = node;

upperProfit = up;

profit = pp;

weight = ww;

level = lev;

}

HeapNode(){

}

int compareTo(HeapNode o) {

double xup =o.upperProfit;

if(upperProfit < xup)

return -1;

if(upperProfit == xup)

return 0;

else

return 1;

}

};

class Element {

public:

int id;

double d;

Element(){

}

Element(int idd,double dd){

id=idd;

d=dd;

}

int compareTo(Element x){

double xd=x.d;

if(d<xd)return -1;

if(d==xd)return 0;

return 1;

}

bool equals(Element x){

return d==x.d;

}

};

class MaxHeap{

public:

HeapNode \*nodes;

int nextPlace;

int maxNumber;

MaxHeap(int n){

maxNumber = pow((double)2,(double)n);

nextPlace = 1;//下一个存放位置

nodes = new HeapNode[maxNumber];

}

MaxHeap(){

}

void put(HeapNode node){

nodes[nextPlace] = node;

nextPlace++;

heapSort(nodes);

}

HeapNode removeMax(){

HeapNode tempNode = nodes[1];

nextPlace--;

nodes[1] = nodes[nextPlace];

heapSort(nodes);

return tempNode;

}

void heapAdjust(HeapNode \* nodes,int s,int m){

HeapNode rc = nodes[s];

for(int j=2\*s;j<=m;j\*=2){

if(j<m&&nodes[j].upperProfit<nodes[j+1].upperProfit)

++j;

if(!(rc.upperProfit<nodes[j].upperProfit))

break;

nodes[s] = nodes[j];

s = j;

}

nodes[s] = rc;

}

void heapSort(HeapNode \* nodes){

for(int i=(nextPlace-1)/2;i>0;--i){

heapAdjust(nodes,i,nextPlace-1);

}

}

} ;

#endif

**测试编码**

#include <iostream>

using namespace std;

//子空间中节点类型

#include "beibao.h"

double c=30;

const int n=3;

double \*w;

double \*p;

double cw;

double cp;

int \*bestX;

MaxHeap \* heap;

//上界函数bound计算结点所相应价值的上界

double bound(int i){

double cleft=c-cw;

double b=cp;

while(i<=n&&w[i]<=cleft){

cleft=cleft-w[i];

b=b+p[i];

i++;

}

//装填剩余容量装满背包

if(i<=n)

b=b+p[i]/w[i]\*cleft;

return b;

}

//addLiveNode将一个新的活结点插入到子集树和优先队列中

void addLiveNode(double up,double pp,double ww,int lev,BBnode\* par,bool ch){

//将一个新的活结点插入到子集树和最大堆中

BBnode \*b=new BBnode(par,ch);

HeapNode node =HeapNode(b,up,pp,ww,lev);

heap->put(node);

}

double MaxKnapsack(){

//优先队列式分支限界法，返回最大价值，bestx返回最优解

BBnode \* enode=new BBnode();

int i=1;

double bestp=0;//当前最优值

double up=bound(1);//当前上界

while(i!=n+1){//非叶子结点

//检查当前扩展结点的左儿子子结点

double wt=cw+w[i];

if(wt<=c){

if(cp+p[i]>bestp)

bestp=cp+p[i];

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

}

up=bound(i+1);

if(up>=bestp)

addLiveNode(up,cp,cw,i+1,enode,false);

HeapNode node =heap->removeMax();

enode=node.liveNode;

cw=node.weight;

cp=node.profit;

up=node.upperProfit;

i=node.level;

}

for(int j=n;j>0;j--){

bestX[j]=(enode->leftChild)?1:0;

enode=enode->parent;

}

return cp;

}

double knapsack(double \*pp,double \*ww,double cc,int \*xx){

//返回最大值，bestX返回最优解

c=cc;

//n=sizeof(pp)/sizeof(double);

//定义以单位重量价值排序的物品数组

Element \*q=new Element[n];

double ws=0.0;

double ps=0.0;

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

q[i]=Element(i+1,pp[i+1]/ww[i+1]);

ps=ps+pp[i+1];

ws=ws+ww[i+1];

}

if(ws<=c){

return ps;

}

p=new double[n+1];

w=new double[n+1];

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

p[i+1]=pp[q[i].id];

w[i+1]=ww[q[i].id];

}

cw=0.0;

cp=0.0;

bestX = new int[n+1];

heap = new MaxHeap(n);

double bestp = MaxKnapsack();

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

xx[q[j].id]=bestX[j+1];

return bestp;

}

void main(){

w=new double[4];

w[1]=16;w[2]=15;w[3]=15;

p=new double[4];

p[1]=45;p[2]=25;p[3]=25;

int \*x = new int[4];

double m = knapsack(p,w,c,x);

cout<<"\*\*\*\*\*分支限界法\*\*\*\*\*"<<endl;

cout<<"\*\*\*\*\*物品个数：n="<<n<<endl;

cout<<"\*\*\*\*\*背包容量：c="<<c<<endl;

cout<<"\*\*\*\*\*物品重量数组：w= {"<<w[3]<<" "<<w[1]<<" "<<w[2]<<"}"<<endl;

cout<<"\*\*\*\*\*物品价值数组：v= {"<<p[3]<<" "<<p[1]<<" "<<p[2]<<"}"<<endl;

cout<<"\*\*\*\*\*最优值：="<<m<<endl;

cout<<"\*\*\*\*\*选中的物品是:";

for(int i=1;i<=3;i++)

cout<<x[i]<<" ";

cout<<endl;

}