

答：

1. 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，大约十万倍吧

伪码：Quicksort的平均比较次数

float c(int n)

if (n <= 1) return 0

sum = 0

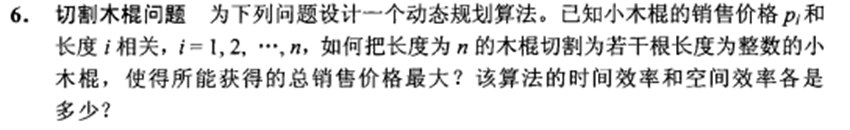
for (m = 1; m <= n; m++)

sum += n-1 + c(m-1) + c(n-m)

return sum/n

如果在输入的数组中最多只有一个元素，那么Quichsort将不会进行比较

1. 不存在



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;

int \_Cut\_Dynamic\_DownToTop(int seg[], int pi[], int arr\_len, int length, int dump[])

{

int tmp;

dump[0] = 0;

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

{

tmp = -1;

for (int j = 0; j < arr\_len; ++j)

{

if (i - seg[j] >= 0)

tmp = max(tmp, pi[j] + dump[i - seg[j]]);

}

dump[i] = tmp;

}

return dump[length];

}

int Cut\_Dynamic\_DownToTop(int seg[], int pi[], int arr\_len, int length)

{

int \*dump = (int \*)malloc(sizeof(int)\*length + 1);

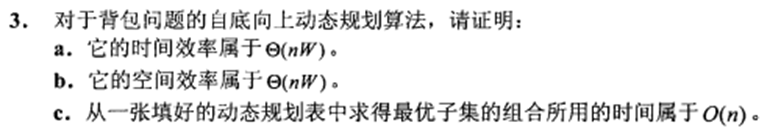
int tmp = \_Cut\_Dynamic\_DownToTop(seg, pi, arr\_len, length, dump);

free(dump);

return tmp;

}

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)。



顺序：将背包物品依次从1,2,...n编号，令i是容量为c共有n个物品的0-1背包问题最优解S的最高编号。则S'=S-{i}一定是容量为c-wi且有1,...,i-1项物品的最优解。如若不是，领S''为子问题最优解，则V(S''+{i})>V(S'+{i})，矛盾。这里V(S)=V(S')+vi.

顺序（从前往后）：设(y1,y2,...,yn)是所给问题的一个最优解。则(y1,...,yn-1)是下面相应子问题的一个最优解：

max ∑vixi

s.t ∑wixi≤c

xi∈{0,1}，1≤i≤n-1

如若不然，设(z1,...,zn-1)是上述子问题的一个最优解，而(y1,...,yn-1)不是它的最优解。由此可知，∑vizi>∑viyi,且∑vizi+wnyn≤c。因此

∑viyi+vnyn>∑viyi(前一个范围是1~n-1，后一个是1~n)

∑vizi+wnyn≤c

这说明(z1,z2,...,yn)是一个所给问题的更优解，从而(y1,y2,...,yn)不是问题的所给问题的最优解，矛盾。

阶段决策过程描述（动态规划的标准形式，这里用顺序法表示）

共有n个阶段，阶段k=1,2,...,n，每个阶段便是需要作出一个决策的子问题部分。这里相当于可以将背包问题分解为n个子问题。

状态与状态变量xk：表示第k阶段背包的容量

决策变量uk：表示k阶段是否取物品k，即uk(xk)∈{0,1}，是一个符号函数。

状态转移方程：xk+1 = xk+uk(xk)\*xk

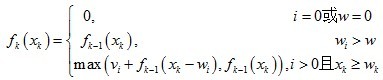
指标函数和最优指标函数，指标函数：衡量决策过程策略的优劣程度、数量指标。定义在全过程上的指标函数记为V1,k(或Vk,n，此时是逆序)，有

V1,k=V1,k(x0,x1,u1,...,xk)

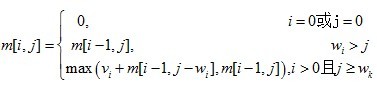
最优指标函数：当指标函数达到最优值时称其为最优指标函数，记为fk(xk).它表示从第初始状态x1起到状态xk过程（或xk→xn，逆‘序)，采取最优策略时得到的指标函数值，如下所示：

fk(xk)=opt V1,k

此处的最优指标函数值fk(xk)：表示共有1,2,...,k个物品背包容量为xk的最优值如下：

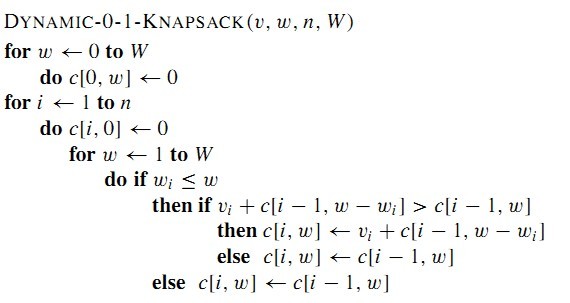


为了便于编程实现，用m[i,j]表示fk(xk)，表示背包容量为j，可选择物品为1,2,...,i时0-1背包问题的最优解。则可得公式如下：

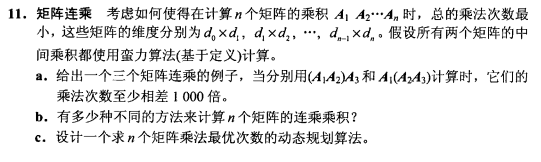


第1个公式表示初始值（边界条件，令其为0）或者背包容量为0则结果也为0，第2个公式表示物品i的重量大于背包容量不能装，第3个公式则是表示是否包含前物品i，前一个包含后一个不包含。

自底向上的非递归算法如下：



如上可证明：其状态数量为n\*W, n为物品数量，W为背包总体积，状态转移复杂度为O(1)，所以综合时间复杂度为O(n\*W)，空间复杂度为O(n\*W)，最优子集的组合所用的时间为O（n）。



1. 矩阵 维数

A1 30\*35

A2 35\*15

A3 15\*5

第一次：我们从最初开始划分，也就是划分两个子矩阵，首先列出所有情况。

A1A2 30\*35\*15=15750

A2A3 35\*15\*5=2625

第二次：划分三个子矩阵

A1A2A3：此时注意A1A2A3有两种子结构划分，由于不管使用哪种子结构，划分后都归类到A1A2A3，也就是说A1A2A3无论有多少子结构，无论如何划分，划分后他都成为（A1A2A3）这一个整体。因此此时我们可以进行pk。（A1（A2A3））2625+30\*35\*5=7875或（（A1A2）A3）15750+30\*15\*5=18000，显然选择7875

总共加括号方法的一个下界是2^n，一个更紧的下界：Ω(4^n/n^(3/2))。  
当n=1时，只有一个矩阵，因此只有一种方法，当n>=2时一个全加括号的矩阵乘积就是连个全部加括号的子矩阵的乘积，而这两个子矩阵分裂的地方可能在第k个和第k+1个矩阵之间，其中k=1,2......,n-1.因此得到递归式(P(n)表示加括号的方法数）:  
 p(n)=1(n=1)或=∑p(k)p(n-k)(k从1到n-1)  
由上面的递归式可以用替换法证明其一个下界确实是2^n

#include<iostream>

#include<cstring>

using namespace std;

const int size=100;

int p[size];

int m[size][size],s[size][size];

int n;

void matrixchain()

{

int i,r,j,k;

memset(m,0,sizeof(m));

memset(s,0,sizeof(s));//初始化数组

for(r=2;r<=n;r++)//矩阵连乘的规模为r

{

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

{

j=i+r-1;

m[i][j]=m[i+1][j]+p[i-1]\*p[i]\*p[j];//对m[][]开始赋值

s[i][j]=i;//s[][]存储各子问题的决策点

for(k=i+1;k<j;k++)//寻找最优值

{

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;

}

}

}

}

}

void print(int i,int j)

{

if(i == j)

{

cout<<"A["<<i<<"]";

return;

}

cout<<"(";

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

print(s[i][j]+1,j);//递归1到s[1][j]

cout<<")";

}

int main()

{

cout<<"请输入矩阵的个数n : "<<endl;

cin>>n;

int i,j;

cout<<"请依次输入每个矩阵的行数和最后一个矩阵的列数："<<endl;

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

cin>>p[i];

matrixchain();

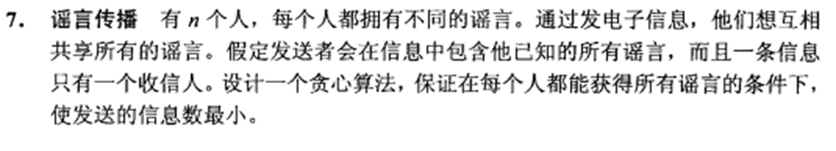
print(1,n);

cout<<endl;

cout<<"最小计算量的值为："<<m[1][n]<<endl;

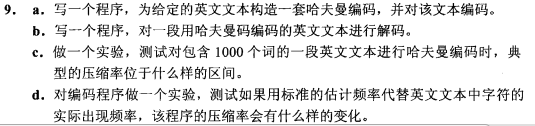
return 0;

}



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。



a.

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

typedef struct HfTreeNode \*HfTree;

typedef struct HNode \*Heap;

struct HNode {

HfTree \*Data;

int Size;

int Capacity;

};

struct HfTreeNode {

char ch;

int weight;

HfTree Left;

HfTree Right;

};

void PrintCode(HfTree root,int \*arr,int top);

void PrintArr(int \*arr,int top);

HfTree DeleteMin( Heap H );

Heap CreateHeap(int size);

void Adjust(Heap H, int p);

void BuildHeap(Heap H);

Heap Read(void);

HfTree Huffman(Heap heap);

int main(int argc, const char \* argv[]) {

int arr[100]={0},top=0;

HfTree Root;

Heap heap;

heap=Read();

Root=Huffman(heap);

PrintCode(Root, arr, top);

return 0;

}

Heap Read(void) {

int num=0,dataNum=1;

int frequency[58]={0};//用于统计次数的数组

Heap heap;

char c;

while((c=getchar())!='\n') {

frequency[c-65]++;

}

for (int i=0; i<58; i++) {

if(frequency[i]!=0)

num++;

}//找出输入的有效字母数量

heap=CreateHeap(num);//建堆

for (int i=0; i<58; i++) {

//找到有效字母

if(frequency[i]!=0) {

//放入节点的过程

HfTree p=(HfTree)malloc(sizeof(struct HfTreeNode));

p->Left=NULL;

p->Right=NULL;

p->weight=frequency[i];

p->ch=i+65;

heap->Data[dataNum++]=p;

heap->Size++;

}

}

return heap;

}

Heap CreateHeap(int MaxSize) {

HfTree p=(HfTree)malloc(sizeof(struct HfTreeNode));

p->Left=NULL;

p->Right=NULL;

p->weight=-1;

/\*

设置哨兵

\*/

Heap H = (Heap)malloc(sizeof(struct HNode));

H->Data = (HfTree \*)malloc((MaxSize+1)\*sizeof(struct HfTreeNode));//data存储的是指向HfTree的指针

H->Size = 0;//初始化堆的大小为0

H->Capacity = MaxSize;//设定堆的容量

/\*

将除data的指针全设为NULL

\*/

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

H->Data[i]=NULL;

H->Data[0]=p;//设置哨兵

return H;

}

void Insert( Heap H, HfTree X ) {

int i=++H->Size;

for (; H->Data[i/2]->weight>X->weight; i/=2 )

H->Data[i] = H->Data[i/2];

H->Data[i] = X;

}

HfTree Huffman(Heap heap) {

BuildHeap(heap);

HfTree p;

for (int i=1; i<heap->Capacity; i++) {

p=(HfTree)malloc(sizeof(struct HfTreeNode));

p->Left=DeleteMin(heap);

p->Right=DeleteMin(heap);

p->weight=p->Left->weight+p->Right->weight;

Insert(heap, p);

}

p=DeleteMin(heap);

return p;

}

void Adjust(Heap H, int p) {

int Parent, Child;

HfTree X;

X = H->Data[p];

for( Parent=p; Parent\*2<=H->Size; Parent=Child ) {

Child = Parent \* 2;

if( (Child!=H->Size) && (H->Data[Child]->weight>H->Data[Child+1]->weight) )

Child++;

if( X->weight <= H->Data[Child]->weight ) break;

else

H->Data[Parent]= H->Data[Child];

}

H->Data[Parent] = X;

}

void BuildHeap( Heap H ) {

/\*

从最后一个节点的父节点开始将一个子单元调整成堆，不断循环直到所有都调成堆

\*/

for(int i = H->Size/2; i>0; i-- )

Adjust(H, i);

}

HfTree DeleteMin(Heap H) {

int Parent, Child;

HfTree MinItem, X;

MinItem = H->Data[1];

X = H->Data[H->Size--];

for( Parent=1; Parent\*2<=H->Size; Parent=Child ) {

Child = Parent \* 2;

if( (Child!=H->Size) && (H->Data[Child]->weight>H->Data[Child+1]->weight) )

Child++;

if( X->weight <= H->Data[Child]->weight ) break;

else

H->Data[Parent]= H->Data[Child];

}

H->Data[Parent] = X;

return MinItem;

}

void PrintCode(HfTree tree,int \*arr,int top) {

if (tree->Left) {

arr[top] = 0;

PrintCode(tree->Left, arr, top + 1);

}

if (tree->Right) {

arr[top] = 1;

PrintCode(tree->Right, arr ,top + 1);

}

// 如果是叶节点就打印

if (tree->Left==NULL&&tree->Right==NULL) {

printf("%c:", tree->ch);

PrintArr(arr, top);

}

}

void PrintArr(int \*arr,int top) {

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

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

printf("\n");

}



#include<iostream>

using namespace std;

int n,a[100],v[100];

//a数组用于保存每一次的排列，v数组用于判断数字是不是已经被选过

void dfs(int dp)//dp从1到4，执行4次，每次选择一个数

{

if(dp>4)//dp为>4的时候，就说明已经选了4个，可以输出了

{

for(int i=1;i<=4;i++)

{

cout<<a[i]<<" ";//先输出a[i]再输出空格

}

cout<<endl;//输出结束

return;

}

for(int i=1;i<=4;i++)

{

if(!v[i])//判断是不是选择过

{

a[dp]=i;//保存当前选择

v[i]=1;//标记这个数字，防止同一个排列选择相同的数字

dfs(dp+1);

v[i]=0;//回溯回来的时候一定要清楚标记，不然下一个排列就能选择了，这也是最关键的地方，要注意

}

}

}

int main()

{

int n=4;

dfs(1);

}



/\*7.写一个程序用分支界限算法对背包问题求解。\*/

#include<iostream>

#include<algorithm>

using namespace std;

#define N 100 //最多可能物体数

struct goods //物品结构体

{

int sign; //物品序号

int w; //物品重量

int p; //物品价值

}a[N];

bool m(goods a,goods b)

{

return (a.p/a.w)>(b.p/b.w);

}

int max(int a,int b)

{

return a<b?b:a;

}

int n,C,bestP=0,cp=0,cw=0;

int X[N],cx[N];

struct KNAPNODE //状态结构体

{

bool s1[N]; //当前放入物体

int k; //搜索深度

int b; //价值上界

int w; //物体重量

int p; //物体价值

};

struct HEAP //堆元素结构体

{

KNAPNODE \*p;//结点数据

int b; //所指结点的上界

};

//交换两个堆元素

void swap(HEAP &a, HEAP&b)

{

HEAP temp = a;

a = b;

b = temp;

}

//堆中元素上移

void mov\_up(HEAP H[], int i)

{

bool done = false;

if(i!=1){

while(!done && i!=1){

if(H[i].b>H[i/2].b){

swap(H[i], H[i/2]);

}else{

done = true;

}

i = i/2;

}

}

}

//堆中元素下移

void mov\_down(HEAP H[], int n, int i)

{

bool done = false;

if((2\*i)<=n){

while(!done && ((i = 2\*i) <= n)){

if(i+1<=n && H[i+1].b > H[i].b){

i++;

}

if(H[i/2].b<H[i].b){

swap(H[i/2], H[i]);

}else{

done = true;

}

}

}

}

//往堆中插入结点

void insert(HEAP H[], HEAP x, int &n)

{

n++;

H[n] = x;

mov\_up(H,n);

}

//删除堆中结点

void del(HEAP H[], int &n, int i)

{

HEAP x, y;

x = H[i]; y = H[n];

n --;

if(i<=n){

H[i] = y;

if(y.b>=x.b){

mov\_up(H,i);

}else{

mov\_down(H, n, i);

}

}

}

//获得堆顶元素并删除

HEAP del\_top(HEAP H[], int&n)

{

HEAP x = H[1];

del(H, n, 1);

return x;

}

//计算分支节点的上界

void bound( KNAPNODE\* node,int M, goods a[], int n)

{

int i = node->k;

float w = node->w;

float p = node->p;

if(node->w>M){ // 物体重量超过背包载重量

node->b = 0; // 上界置为0

}else{

while((w+a[i].w<=M)&&(i<n)){

w += a[i].w; // 计算背包已装入载重

p += a[i++].p; // 计算背包已装入价值

}

if(i<n){

node->b = p + (M - w)\*a[i].p/a[i].w;

}else{

node -> b = p;

}

}

}

//用分支限界法实现0/1背包问题

int KnapSack4(int n,goods a[],int C, int X[])

{

int i, k = 0; // 堆中元素个数的计数器初始化为0

int v;

KNAPNODE \*xnode, \*ynode, \*znode;

HEAP x, y, z, \*heap;

heap = new HEAP[n\*n]; // 分配堆的存储空间

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

a[i].sign=i; //记录物体的初始编号

}

sort(a,a+n,m); // 对物体按照价值重量比排序

xnode = new KNAPNODE; // 建立父亲结点

for( i=0; i<n; i++){ // 初始化结点

xnode->s1[i] = false;

}

xnode->k = xnode->w = xnode->p = 0;

while(xnode->k<n) {

ynode = new KNAPNODE; // 建立结点y

\*ynode = \*xnode; //结点x的数据复制到结点y

ynode->s1[ynode->k] = true; // 装入第k个物体

ynode->w += a[ynode->k].w; // 背包中物体重量累计

ynode->p += a[ynode->k].p; // 背包中物体价值累计

ynode->k ++; // 搜索深度++

bound(ynode, C, a, n); // 计算结点y的上界

y.b = ynode->b;

y.p = ynode;

insert(heap, y, k); //结点y按上界的值插入堆中

znode = new KNAPNODE; // 建立结点z

\*znode = \*xnode; //结点x的数据复制到结点z

znode->k++; // 搜索深度++

bound(znode, C, a, n); //计算节点z的上界

z.b = znode->b;

z.p = znode;

insert(heap, z, k); //结点z按上界的值插入堆中

delete xnode;

x = del\_top(heap, k); //获得堆顶元素作为新的父亲结点

xnode = x.p;

}

v = xnode->p;

for( i=0; i<n; i++){ //取装入背包中物体在排序前的序号

if(xnode->s1[i]){

X[a[i].sign] =1 ;

}else{

X[a[i].sign] = 0;

}

}

delete xnode;

delete heap;

return v; //返回背包中物体的价值

}

/\*测试以上算法的主函数\*/

int main()

{

goods b[N];

printf("物品个数n: ");

scanf("%d",&n); //输入物品个数

printf("背包容量C: ");

scanf("%d",&C); //输入背包容量

for (int i=0;i<n;i++) //输入物品i的重量w及其价值v

{

printf("物品%d的重量w[%d]及其价值v[%d]: ",i+1,i+1,i+1);

scanf("%d%d",&a[i].w,&a[i].p);

b[i]=a[i];

}

int sum4=KnapSack4(n,a,C,X);//调用分支限界法求0/1背包问题

printf("分支限界法求解0/1背包问题:\nX=[ ");

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

cout<<X[i]<<" ";//输出所求X[n]矩阵

printf("] 装入总价值%d\n",sum4);

return 0;

}