**实验1、《分治算法实验》**

**一、实验目的**

1. 了解分治策略算法思想

2. 掌握快速排序、归并排序算法

3. 了解其他分治问题典型算法

**二、实验内容**

1．编写一个简单的程序，实现归并排序。

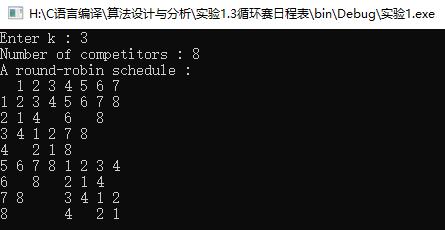
2. 编写一段程序，实现快速排序。

3. 编写程序实现循环赛日程表。设有n=2k个运动员要进行网球循环赛。现要设计一个满足以下要求的比赛日程表：（1）每个选手必须与其它n-1个选手各赛一次（2）每个选手一天只能赛一场（3）循环赛进行n-1天

1. **算法思想分析**
2. 归并排序先是将待排序集合分成两个大小大致相同的集合，分别对每个集合进行排序，递归调用归并排序函数，再是调用合并函数，将两个集合归并为一个排好序的集合。
3. 快速排序先是选择关键数据作为比较量，然后将数组中比它小的数都放到它的左边，比它大的数放大右边，再对左右区间重复上一步，直至各个区间只有一个数。
4. 循环赛日程表先将选手分为两部分，分别排序，再将两部分合并，合并时由于循环赛的规律得知直接将左上角的排序表复制到右下角，左下角的排序表复制到右上角即可。分成两部分时需要利用递归不断分下去直至只剩下一位选手。

**四、实验过程分析**

1. 通过归并算法我对分治算法有了初步的实际操作经验，快速排序与归并算法有很大的相似点，但是在合并时的方法不一样，而循环赛日程表则是思路问题，这个题目编程难点应该在于合并时数组调用的for循环的次数以及起始位置问题。
2. 对于分治算法一般是将大规模问题分解为小问题，通过递归不断分下去，然后对每个小规模用一个函数去求解。适用于小规模独立且易解，可以合并到大问题具有最优子结构的问题。
3. 归并排序和快速排序熟悉书本及PPT基本没有问题，循环赛日程表则是纠结了很久，一开始算法思路并不是十分清晰所以错了很多次，后来想了很久再观察PPT的循环赛日程表得知最终算法，编写代码中遇到了一个小问题，有一部分选手未排序，如图所示：



图中有部分选手未排序，即左下角排序出现了问题，后来直接静态调试，自己按照代码用实际数据去试了一遍，发现是排序时的for循环的次数不对。一开始用最小规模写的代码，即只考虑了两位选手，次数用的是实际数据，但是当规模扩大时该数据是会变化的，所以导致后来的错误。

**五、算法源代码及用户屏幕**

1.（1）算法源码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

归并排序（从小到大）。

codeblock C++

2018/10/16

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

using namespace std;

//函数声明

void merge\_sort(int a[],int left,int right);

void merge(int b[],int left ,int mid, int right);

int main() {

int q;

cout<<"To invoke the default array, enter 1, and the user-defined array input 2 :"<<endl;

cin>>q;

if(1 == q){

//固定数组

int a[9] = {12, 45, 5, 58, 48, 85, 27, 82, 2};

cout<<"The array before the sort ： "<<endl;

for(int i=0; i<9; i++)

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

cout<<endl;

//对数组进行归并排序

merge\_sort(a, 0, 8); //调用排序函数

cout<<"Sorted array ： "<<endl; //输出排序后的数组

for(int i=0; i<9; i++)

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

cout<<endl;

}

else{

//用户自由输入数组

int n; //数组大小

cout<<"Please enter the size of the array : "; //用户定义数组大小

cin>>n;

int a[n];

cout<<"Please enter the array data : "<<endl;

for(int i=0;i<n;i++){ //用户输入数组数据

cin>>a[i];

}

//对数组进行归并排序

merge\_sort(a, 0, n-1); //调用排序函数

cout<<"Sorted array ： "<<endl; //输出排序后的数组

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

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

cout<<endl;

}

return 0;

}

void merge\_sort(int a[],int left,int right){

if(left<right){

int mid = (left + right)/2; //求数组的中间元素标号

merge\_sort(a,left,mid); //对数组左半部分调用归并排序函数

merge\_sort(a,mid+1,right); //对数组右半部分调用归并排序函数

merge(a,left,mid,right); //将左右部分合并到同时排好序到原数组上

}

}

void merge(int a[],int left ,int mid, int right){

//创建动态数组，暂时存储对左右部分排序后的数组

int \*temp = new int[right-left+1];

int p,q,r; //设置三个变量，一次代替left，mid，right

p=left;

q=mid+1;

r=right;

int i=0;

//比较左右部分数据大小，小的先放入动态数组

while(p<=mid && q<=right){

if(a[p]<=a[q]){ //左半部分数据较小，先入temp数组

temp[i] = a[p];

i++;

p++;

}

else{

temp[i] = a[q]; //右半部分数据较小，放入temp数组

i++;

q++;

}

}

//左半部分有剩余，

while(p<=mid){

temp[i] = a[p];

i++;

p++;

}

//右半部分有剩余

while (q<=right){

temp[i] = a[q];

i++;

q++;

}

//将重新排好序的数组归并到原数组上

for(int j=right; j>=left; j--){

i--;

a[r] = temp[i];

r--;

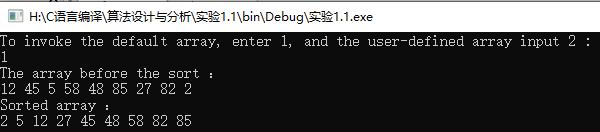
}

delete[] temp; //释放动态数组内存

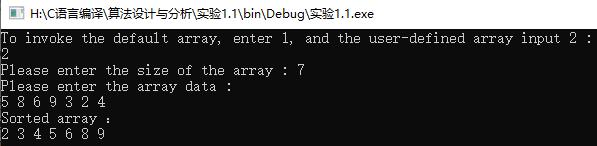
}

1. 用户屏幕

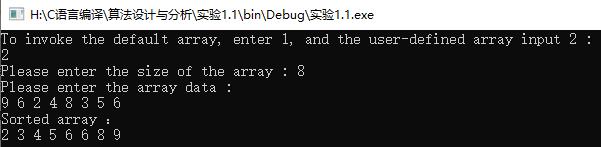
程序采用默认数组数据：12 45 5 58 48 85 17 82 2



用户输入数组大小为7，数据：5 8 6 9 3 2 4



用户输入数组大小8，数据：9 6 2 4 8 3 5 6



2.（1）算法源码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

快速排序。（从小到大）

codeblock C++

2018.10.20

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

using namespace std;

//函数声明

void quicksort(int a[],int left,int right);

int \_partition(int a[],int left,int right);

int main(){

int q;

cout<<"quicksort"<<endl;

cout<<"To invoke the default array, enter 1, and the user-defined array input 2 :"<<endl;

cin>>q;

if(1 == q){

//固定数组

int a[9] = {12, 45, 5, 58, 48, 85, 27, 82, 2};

cout<<"The array before the sort ： "<<endl;

for(int i=0; i<9; i++)

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

cout<<endl;

//对数组进行快速排序

quicksort(a, 0, 8); //调用排序函数

cout<<"Sorted array ： "<<endl; //输出排序后的数组

for(int i=0; i<9; i++)

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

cout<<endl;

}

else{

//用户自由输入数组

int n; //数组大小

cout<<"Please enter the size of the array : "; //用户定义数组大小

cin>>n;

int a[n];

cout<<"Please enter the array data : "<<endl;

for(int i=0;i<n;i++){ //用户输入数组数据

cin>>a[i];

}

//对数组进行快速排序

quicksort(a, 0, n-1); //调用排序函数

cout<<"Sorted array ： "<<endl; //输出排序后的数组

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

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

cout<<endl;

}

return 0;

}

void quicksort(int a[],int left,int right){

int q;

if(left<right){

q=\_partition(a,left,right); //分解

quicksort(a,left,q); //排序

quicksort(a,q+1,right);

}

}

int \_partition(int a[],int left,int right){

//p,q分别代表left，right

int p,q;

p=left;

q=right;

//将关键数据暂时存储到变量中，空出a[p]的位置

int s = a[p];

while(p<q){

//从右端开始找，直到找到一个数小于s,然后赋值给a[p]

while(p<q && a[q]>=s)

q--;

if(p<q)

a[p]=a[q];

//从左端开始找，直到找到一个数大于s，然后将值赋给a[q]

while(p<q && a[p]<=s)

p++;

if(p<q)

a[q]=a[p];

}

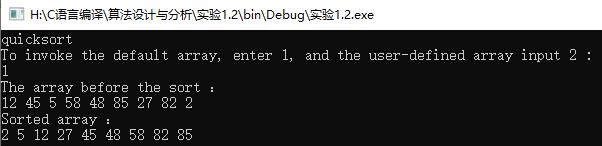
a[p]=s;

return p;

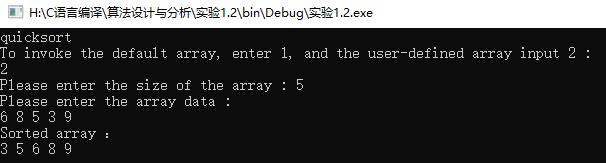
}

（2）用户屏幕

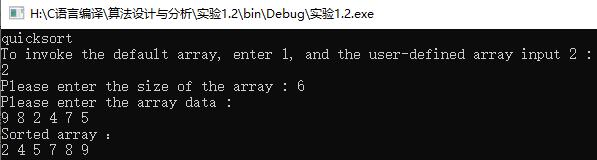
程序采用默认数组数据：12 45 5 58 48 85 17 82 2



自定义输入数组大小：5，数组数据：6 8 5 3 9



自定义输入数组大小：6，数组数据：9 8 2 4 7 5



3.（1）算法源码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

n=个运动员的循环赛日程表。

codeblocks C++

2018.10.20

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include<math.h>

#include<iomanip>

using namespace std;

#define a\_size 200

//声明函数

void cycleScheduleSort(int a[a\_size][a\_size],int k,int startRow);

void cs\_merge(int a[a\_size][a\_size],int k,int startRow);

int main()

{

int k;

cout<<"Enter k : ";

cin>>k;

int m = pow(2,k);

cout<<"Number of competitors : "<<m<<endl;

//创建二维数组，存储日程表信息

int a[a\_size][a\_size];

//初始化第一行为天数

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

a[0][i]=i;

}

//初始化第一列为队伍数

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

a[i][0]=i;

}

//利用循环赛日程表算法排序

cycleScheduleSort(a,k,1);

cout<<"A round-robin schedule :"<<endl;

//输出排序后的日程表

for(int i=0;i<m+1;i++){

for(int j=0;j<m;j++){

if(a[i][j] != 0)

//控制输出格式：左对齐，大小为3

cout<<setiosflags(ios::left)<<setw(3)<<a[i][j]; else

cout<<setiosflags(ios::left)<<setw(3)<<" ";

}

cout<<endl;

}

return 0;

}

void cycleScheduleSort(int a[a\_size][a\_size],int k,int startRow){

if(k>0){

int m = pow(2,k-1); //m记录已排好序的选手数目

//将选手分为两部分，分别排序

cycleScheduleSort(a,k-1,startRow);

cycleScheduleSort(a,k-1,startRow+m);

//将排好序的两部分合并

cs\_merge(a,k,startRow);

}

}

void cs\_merge(int a[a\_size][a\_size],int k,int startRow){

int m = pow(2,k-1);

//将左上角的比赛日程复制到右下角

for(int i=startRow;i<m+startRow;i++){

for(int j=0;j<m;j++){

a[i+m][j+m] = a[i][j];

}

}

//将左下角的比赛日程复制到右上角

for(int i=startRow+m;i<startRow+m\*2;i++){

for(int j=0;j<m;j++){

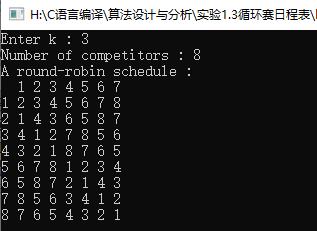
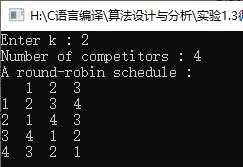
a[i-m][j+m] = a[i][j];

}

}

}

（2）用户屏幕



实验2、《贪心算法实验》

**一、实验目的**

1. 了解贪心算法思想

2. 掌握贪心法典型问题，如背包问题、作业调度问题等。

**二、实验内容**

1. 编写一个简单的程序，实现单源最短路径问题。

2. 编写一段程序，实现找零。

【问题描述】当前有面值分别为2角5分，1角，5分，1分的硬币，请给出找n分钱的最佳方案（要求找出的硬币数目最少）。

3. 编写程序实现多机调度问题

【问题描述】要求给出一种作业调度方案，使所给的n个作业在尽可能短的时间内由m台机器加工处理完成。约定，每个作业均可在任何一台机器上加工处理，但未完工前不允许中断处理。作业不能拆分成更小的子作业。

**三、算法思想分析**

1.初始化将源点设计为红点集，其余点设计为蓝点，重复选择蓝点集中与源点路径最短的点加入红点集，更新剩余的蓝点集路径，直至蓝点集为空或者只剩下没有连通的点，那么源点到其余所有点的最短路径就出来了。

2.找零问题是典型的贪心问题，但是并不代表所有的找零都能用贪心算法找到最优解。只有满足贪心选择性质的找零才能找到最优解，本题满足贪心选择性质，直接先一直选面值最大的硬币，再一次减小即可。

3.先对作业按时长进行重排序，再依次找目前用时最短的机器安排工作并加上对应时长，最后总时长为机器中用时最长的那个时长。

**四、实验过程分析**

1.单源最短路径的算法思想并不难，但是在实际编码过程中还是有很多小问题需要注意，首先，一定要新建数组存储路径变化，因为后面计算路径时会用到原数组，如果直接在原数组上更改后面就找不到原数据了，那么就会出现偏差。其次就是建议先写个伪代码，判断的if-else语句比较多，容易搞混，在代码中一定要及时备注，某些代码的功能是什么，不然再次看代码时需要思考很久甚至忘记。

2.找零问题直接用while循环或者不断取余取模即可解决。

3.作业调度问题大致分为三步，一是排序，二是不断找最短时长的机器安排作业，三是找最长时间为作业完成时间。

**五、算法源代码及用户屏幕**

1.（1）算法源码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

单源最短路径问题。

codeblocks C++

2018.10.26

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include <iomanip>

using namespace std;

int main()

{ //原始带权有向图

int a[5][5] = {{0,10,-1,30,100},

{-1,0,50,-1,-1},

{-1,-1,0,-1,10},

{-1,-1,20,0,60},

{-1,-1,-1,-1,0}};

//单源最短路径图

int b[5][5] = {{0,10,-1,30,100},

{0,0,0,0,0},

{0,0,0,0,0},

{0,0,0,0,0},

{0,0,0,0,0}};

//创建数组红蓝点集

int red[5] = {0,-1,-1,-1,-1};

int red\_N = 0;

int blueToRed = 0;

int blue[5] = {-1, 1, 2, 3, 4};

int blue\_N = 0;

for(int i=0; i<5-1; i++){

//从蓝点集中选出与源点最短路径的蓝点

//初始化随机蓝点

int blue\_min;

for(int j=0; j<5; j++){

if(blue[j]>=0)

blue\_min = j;

}

for(int j=0; j<5; j++){

if(blue[j]>0 && b[i][j]>=0 && b[i][j]<b[i][blue\_min]){

//blue[j]>0表示j是蓝点，a[i][j]>0表示i,j两点连通

blue\_min = j;

}

}

//将蓝点集中与源点最短路径的点加入到红点集

red[++red\_N] = blue\_min;

//将该点从蓝点集中去除

blue[blue\_min] = -1;

//更改加入后的数据

for(int j=0; j<5; j++){

//判断j点是否为蓝点

if(blue[j]>=0){ //j是蓝点

if(a[blue\_min][j]>0){ //新加入的蓝点与j点连通

if(b[i][j]<0){ //上一步i,j两点不连通

b[i+1][j] = b[i][blue\_min]+a[blue\_min][j];

}

//上一步i,j两点连通且加起来路径比原来短

else if(b[i][blue\_min] + a[blue\_min][j] < b[i][j]){

b[i+1][j] = b[i][blue\_min]+a[blue\_min][j];

}

else{ //上一步i,j两点连通但加起来路径比原来长

b[i+1][j] = b[i][blue\_min]+a[blue\_min][j];

}

}

else{ //新加入蓝点与j点不连通

b[i+1][j] = b[i][j];

}

}

else{ //j不是蓝点

b[i+1][j] = b[i][j];

}

}

}

//输出原始带权有向图的邻接矩阵图

cout<<"Original:"<<endl;

//输出第一行

cout<<setw(3)<<" ";

for(int i=0; i<5; i++)

cout<<setw(3)<<i;

cout<<endl;

for(int i=0; i<5; i++){

//输出行号点数

cout<<setw(3)<<i;

for(int j=0; j<5; j++){

cout<<setw(3)<<a[i][j];

}

cout<<endl;

}

cout<<endl;

//输出最终结果

cout<<"Changed:"<<endl;

//输出第一行

cout<<setw(3)<<" ";

for(int i=0; i<5; i++)

cout<<setw(3)<<i;

cout<<endl;

for(int i=0; i<5; i++){

//输出行号点数

cout<<setw(3)<<i;

for(int j=0; j<5; j++){

cout<<setw(3)<<b[i][j];

}

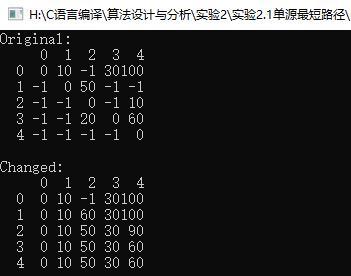
cout<<endl;

}

return 0;

}

（2）用户屏幕



2.（1）算法源码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

实现面值分别为2角5分，1角，5分，1分的硬币找零

codeblocks C++

2018.10.26

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

using namespace std;

int main()

{

int number;

int remain; //记录剩下的数额

cout<<"Please enter the amount of change you want (in cents)：";

cin>>number;

//建立a数组保存已有面值硬币，单位为分

double a[4]={25,10,5,1};

//建立b数组保存找回每个硬币的数量

double b[4]={0,0,0,0};

for(int i=0;i<4;i++){

int x = 0; //记录硬币数量

while(number>=a[i]){

x++;

number -= a[i];

}

b[i] = x;

}

cout<<"25 cents: "<<b[0]<<endl;

cout<<"10 cents: "<<b[1]<<endl;

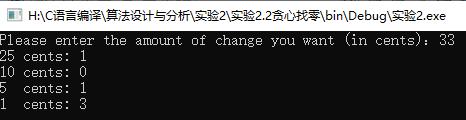
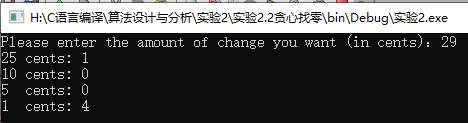
cout<<"5 cents: "<<b[2]<<endl;

cout<<"1 cents: "<<b[3]<<endl;

return 0;

}

（2）用户屏幕



3.（1）算法源码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

多机调度问题。

codeblocks C++

2018.10.26

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

using namespace std;

int shortestTime(int b[],int machines);

int longestTime(int b[],int machines);

void quicksort(int a[],int left,int right);

int \_partition(int a[],int left,int right);

int main()

{

//用户输入作业、机器数量，每个作业完成时长

int n;

cout<<"Please enter the number of jobs：";

cin>>n;

int a[n];

cout<<"Please enter the completion time for each job："<<endl;

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

cin>>a[i];

}

//对作业进行重排序

quicksort(a,0,n);

int machine;

cout<<"Please enter the number of machines：";

cin>>machine;

int b[machine];

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

b[i] = 0;

int shortT;

//找到目前最短时长的机器并安排作业

for(int i=n-1;i>=0;i--){

shortT = shortestTime(b,machine);

b[shortT] += a[i];

}

int longT;

longT = longestTime(b,machine);

cout<<"Total processing time："<<b[longT]<<endl;

return 0;

}

//快速排序算法

void quicksort(int a[],int left,int right){

int q;

if(left<right){

q=\_partition(a,left,right); //分解

quicksort(a,left,q); //排序

quicksort(a,q+1,right);

}

}

int \_partition(int a[],int left,int right){

//p,q分别代表left，right

int p,q;

p=left;

q=right;

//将关键数据暂时存储到变量中，空出a[p]的位置

int s = a[p];

while(p<q){

//从右端开始找，直到找到一个数小于s,然后赋值给a[p]

while(p<q && a[q]>=s)

q--;

if(p<q)

a[p]=a[q];

//从左端开始找，直到找到一个数大于s，然后将值赋给a[q]

while(p<q && a[p]<=s)

p++;

if(p<q)

a[q]=a[p];

}

a[p]=s;

return p;

}

//寻找最短时间

int shortestTime(int b[],int machines){

int flag = 0;

int small = b[flag];

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

if(b[i]<small){

small = b[i];

flag = i;

}

}

return flag;

}

//寻找最长时间

int longestTime(int b[],int machines){

int flag = 0;

int longest = b[flag];

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

if(b[i]>longest){

longest = b[i];

flag = i;

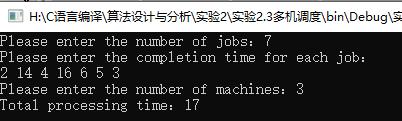
}

}

return flag;

}

（2）用户屏幕



实验3、《动态规划算法实验》

**一、实验目的**

1. 掌握动态规划方法贪心算法思想

2. 掌握最优子结构原理

3. 了解动态规划一般问题

**二、实验内容**

1. 编写一个简单的程序，解决0-1背包问题。设N=5，C=10，w={2，2，6，5，4},v={6,3,5,4,6}

2. 合唱队形安排问题

【问题描述】N位同学站成一排，音乐老师要请其中的(N-K)位同学出列，使得剩下的K位同学排成合唱队形。合唱队形是指这样的一种队形：设K位同学从左到右依次编号为1，2…，K，他们的身高分别为T1，T2，…，TK， 则他们的身高满足T1<...<Ti>Ti+1>…>TK(1<=i<=K)。已知所有N位同学的身高，计算最少需要几位同学出列，可以使得剩下的同学排成合唱队形。

**三、算法思想分析**

1.0-1背包采用动规算法主要是动规方程的思考，之后就是确定边界条件即可。

2.合唱队形问题应用了分治与动态规划的算法，先将所有队员依次做中间最高的同学，将问题分为左右来做，接下来只需要求得左边的最长上升子序列数、右边的最长下降子序列数即可。

**四、实验过程分析**

1.0-1背包问题是背包问题的进一步条件限制，考虑清楚动规方程就不难，编程中对于m(i,j)的含义要清楚，搞混了就容易出错。

2.合唱队形问题的思想并不复杂，特别是如果已经掌握了最长上升子序列数的算法，在分别处理左右最长子序列时需要特别注意数组下标，一开始我用是i，j直接从0到左右的数据长度，但是一直出错，后来发现队员身高数组并不能完全用这些下标，特别是右边的函数，数组起始下标不是0，需要利用函数传递起始下标才能调用对应的数据段。

**五、算法源代码及用户屏幕**

1.（1）算法源码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

0-1背包问题。

codeblocks C++

2018.11.2

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include <iomanip>

using namespace std;

void knapSnack(int v[], int w[], int c, int n, int m[][11]);

int main()

{

int v[] = {6, 3, 5, 4, 6};

int w[] = {2, 2 ,6, 5, 4};

int c = 10;

int n = 5;

int m[5][11];

//初始化数组

for(int i=0; i<5; i++){

for(int j=0; j<11; j++){

m[i][j] = 0;

}

}

knapSnack(v, w, c, n, m);

//输出结果

cout<<setw(3)<<" ";

for(int i=0; i<11; i++){

cout<<setw(3)<<i;

}

cout<<endl;

for(int i=0; i<5; i++){

//输出行号

cout<<setw(3)<<i+1;

for(int j=0; j<11; j++){

cout<<setw(3)<<m[i][j];

}

cout<<endl;

}

return 0;

}

void knapSnack(int v[], int w[], int c, int n, int m[][11]){

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

for(int j=0; j<11; j++){

//边界条件

if(i == 0){

if(w[i] > j)

m[i][j] = 0;

else

m[i][j] = v[i];

}

/\*动规方程

j>w[i]

m(i,j) = max{m(i-1,j), m(i-1,j-w[i])+v[i]}

0<=j<w[i]

m(i,j) = m(i-1,j)

\*/

else{

if(w[i] > j)

m[i][j] = m[i-1][j];

else{

if(m[i-1][j] > (m[i-1][j-w[i]]+v[i]))

m[i][j] = m[i-1][j];

else

m[i][j] = m[i-1][j-w[i]]+v[i];

}

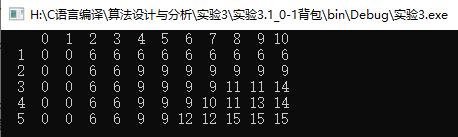
}

}//控制列数的for循环

}//控制行数的for循环

}

（2）用户屏幕



2.（1）算法源码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

合唱队形问题

codeblocks C++

2018.11.2

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include <string.h>

using namespace std;

//计算左端合唱队人数

int leftQueue(int a[], int \_start, int \_end);

//计算右端合唱队人数

int rightQueue(int a[], int \_start2, int \_end2);

int main()

{

cout<<"Please enter total number：";

int number;

cin>>number;

cout<<"Please input the height of each person (cm)："<<endl;

int a[number]; //记录每个人身高

//b数组分别记录当第n个人为合唱队中间人时,合唱队的总人数

int b[number];

int rightNumber[number]; //记录左端合唱队人数

int leftNumber[number]; //记录右端合唱队人数

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

b[i] = 0;

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

cin>>a[i];

int mostQueueNumber = b[0];

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

//设置a[i]为最高的同学

leftNumber[i] = leftQueue(a,0,i);

rightNumber[i] = rightQueue(a,i,number-1);

//计算合唱队总人数

b[i] = leftNumber[i] + rightNumber[i] - 1;

//计算合唱队最多的总人数

if(mostQueueNumber < b[i])

mostQueueNumber = b[i];

}

//计算最少出队人数

int leastDequeueNumber = number - mostQueueNumber;

cout<<"Minimum number of people out: "<<leastDequeueNumber<<endl;

return 0;

}

int leftQueue(int a[], int \_start, int \_end){

int leftMostNumber = 0;

int n = \_end-\_start+1;

//c数组记录i时的最长上升子序列数

int c[n];

int maxN;

//初始化最长上升子序列数为1

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

c[i] = 1;

}

for(int i=\_start; i<\_end+1; i++){

maxN = 0;

for(int j=i-1; j>=\_start; j--){

if(a[j]<a[i] && c[j]>maxN)

maxN = c[j];

c[i] = maxN + 1;

}

}

leftMostNumber = c[n-1];

return leftMostNumber;

}

int rightQueue(int a[], int \_start2, int \_end2){

int rightMostNumber = 0;

int n2 = \_end2-\_start2+1;

//c2数组记录i时的最长下降子序列数

int c2[n2];

int maxN2;

//初始化最长下降子序列数为1

for(int i=0; i<n2; i++){

c2[i] = 1;

}

for(int i=\_end2; i>=\_start2; i--){

maxN2 = 0;

for(int j=i+1; j<=\_end2; j++){

if(a[j]<a[i] && c2[j-\_start2]>maxN2)

maxN2 = c2[j-\_start2];

c2[i-\_start2] = maxN2 + 1;

}

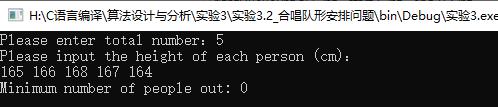
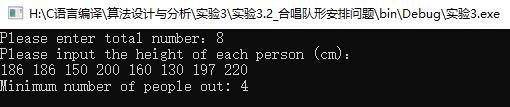
}

rightMostNumber = c2[0];

return rightMostNumber;

}

（2）用户屏幕



实验4、《回溯法实验》

**一、实验目的**

1. 掌握回溯算法思想

2. 掌握回溯递归原理

3. 了解回溯法典型问题

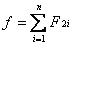
**二、实验内容**

1. 编写一个简单的程序，解决8皇后问题。

2. 批处理作业调度问题

[问题描述]给定n个作业的集合J=(J1, J2, … , Jn)。每一个作业Ji都有两项任务需要分别在2台机器上完成。每一个作业必须先由机器1处理，然后再由机器2处理。作业Ji需要机器i的处理时间为tji，i=1,2, … ,n; j=1,2。

对于一个确定的作业调度，设Fji是作业i在机器i上完成处理的时间。则所有作业在机器2上完成处理的时间和成为该作业调度的完成时间和。



批处理作业调度问题要求对于给定的n个作业，制定一个最佳的作业调度方案，使其完成时间和达到最小。

要求输入：

1）作业数 2）每个作业完成时间表：

|  |  |  |
| --- | --- | --- |
| 作业完成时间 | 机器1 | 机器2 |
| 作业1 | 2 | 1 |
| 作业2 | 3 | 1 |
| 作业3 | 2 | 3 |

要求输出： 1）最佳完成时间 2）最佳调度方案

提示：算法复杂度为O(n!),建议在测试的时候n值不要太大，可以考虑不要超过12。

3. 数字全排列问题

任意给出从1到N的N个连续的自然数，求出这N个自然数的各种全排列。如N=3时，共有以下6种排列方式：123，132，213，231，312，321。

注意：数字不能重复，N由键盘输入（N<=9）。

**三、算法思想分析**

1.八皇后问题是典型的回溯问题，先从空格子起逐行放皇后，如果符合要求即安全则放置，否则返回上一行下一个位置继续，直至最后一行安全放置则为一种放置方式。

2.批处理作业调度的解空间为排列数，不断利用递归函数直至叶节点，剪枝函数为当前用时与最佳用时的比较。关于时间的计算，每次选择作业后先将机器1用时累加，机器2上总用时需要先比较上一个作业完成时间与此时机器1上的总用时，如果机器1上总用时大于上一作业用时，那么机器2上用时则加上机器1上用时与此作业在机器2上的单独用时，反之，则代表此时机器2仍然在处理上一任务，那么机器2上用时则加上上一作业用时与此作业在机器2上的单独用时。

3.数字全排列问题的解空间为排列树，依次向下排列，判断是否继续的条件为该数字是否已经使用，通过一个flag与for循环判断即可，直至到了最后一个数字则输出结果。

**四、实验过程分析**

1.八皇后问题的难点在于安全条件的设计，即同一列，对角线都不能同时放置，对角线的设计比较巧妙，根据PPT设计向右斜为i-j+N,向左斜为i+j，利用二维数组很好的避免了代码的重复与累赘。

2.批处理作业调度的回溯条件即当前时间与最佳时间的比较，本题的难点在于时间的计算，最开始对于机器2上的总用时是用一个整型数据记录，但没办法与上一个作业用时比较，或者说不方便进行下一次回溯条件的改变，后来参考课本PPT采用数组记录就解决了。计算时间的另一个问题即作业进入机器2上时，上一个作业是否已经完成，需不需要等待，这里要加判断条件，否则时间计算会出问题。

3.从数字全排列问题的实验中，我更加熟悉了回溯的基本套路，判断数字是否使用的flag和for循环设计很典型也很实用。

**五、算法源代码及用户屏幕**

1.（1）算法源码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

八皇后问题。

codeblocks C++

2018.11.3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include <iomanip>

using namespace std;

#define N 8 //定义皇后数量

int countN = 0;

int tryN(int n, int a[N][N], int m[], int right[], int left[]); //检测皇后摆放方式

void printN(int a[N][N]); //打印皇后摆放位置

int main()

{

int a[N][N] = {0}; //摆放皇后的数组

int m[N] = {0}; //同一竖排

int right[N\*2] = {0}; //右斜对角线

int left[N\*2] = {0}; //左斜对角线

int countN = tryN(0, a, m, right, left);

cout<<"count = "<<countN<<endl;

return 0;

}

int tryN(int n, int a[N][N], int m[], int right[], int left[]){

for(int j=0; j<N; j++){

if(!m[j] && !right[n-j+N] && !left[n+j]){ //位置安全

//放皇后

a[n][j] = n+1;

m[j] = right[n-j+N] = left[n+j] = 1;

if(n == N-1){ //已经到最后一行

printN(a);

countN++;

}

else{ //检测下一行

tryN(n+1, a, m, right, left);

}

//回溯，去掉该皇后

a[n][j] = 0;

m[j] = right[n-j+N] = left[n+j] = 0;

}

}

return countN;

}

void printN(int a[N][N]){

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

for(int j=0; j<N; j++){

cout<<setw(2)<<a[i][j];

}

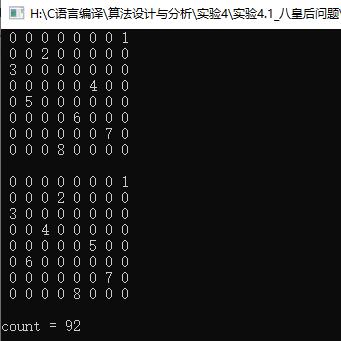
cout<<endl;

}

cout<<endl;

}

（2）用户屏幕



2.（1）算法源码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

批处理作业调度问题

codeblocks C++

2018.11.4

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include <iomanip>

using namespace std;

const int num = 100;

int m1Time[num]; //作业在机器1上花费的时间

int m2Time[num]; //作业在机器2上花费的时间

int workBest[num]; //记录最有作业调度方法

int work[num]; //记录当前作业调度方法

int timeCurrent = 0;

int timeBest = 9999;

int timeM1 = 0; //记录机器1上用时

int timeM2[num] = {0}; //记录每个作业总用时，从下标1开始

void backTrack(int deep, int workNumber);

int main()

{

//用时输入作业数量与在机器1、2上的用时

int workNumber;

cout<<"Enter work numbers : ";

cin>>workNumber;

cout<<"work m1 m2"<<endl;

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

cout<<" "<<i+1<<" ";

cin>>m1Time[i];

cin>>m2Time[i];

}

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

workBest[i] = i; //顺序调度,编号-1

work[i] = i;

}

backTrack(0,workNumber);

cout<<"BestTime : "<<timeBest<<endl;

cout<<"Scheduling sort : "<<endl;

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

cout<<workBest[i]+1<<" ";

cout<<endl;

return 0;

}

void backTrack(int deep, int workNumber){

if(deep >= workNumber){ //到底叶节点

if(timeCurrent < timeBest){

//该调度方法为最佳方法，更新最佳方法的数据

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

workBest[i] = work[i];

}

timeBest = timeCurrent;

}

}

else{

for(int i=deep; i<workNumber; i++){

//从剩余的作业中选择一个与当前作业交换位置

int temp = work[deep];

work[deep] = work[i];

work[i] = temp;

timeM1 += m1Time[work[deep]]; //在机器1上总用时

//作业deep+1在机器2上的完成时间和

//在机器1完成时间如果大于上一个作业时间，则直接进入机器2继续，

//反之，如果小于上一个作业完成时间，则需要继续等待，直到上一个作业完成

timeM2[deep+1] = (timeM2[deep] > timeM1 ? timeM2[deep]:timeM1 ) + m2Time[work[deep]];

//该调度当前总用时

timeCurrent += timeM2[deep+1];

if(timeCurrent < timeBest){

backTrack(deep+1,workNumber);

}

timeM1 -= m1Time[work[deep]];

timeCurrent -= timeM2[deep+1];

temp = work[deep];

work[deep] = work[i];

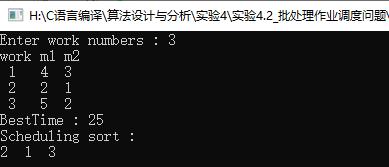
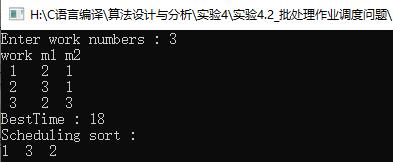
work[i] = temp;

}

}

}

（2）用户屏幕



3.（1）算法源码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

N个自然数全排列问题。

codeblocks C++

2018.11.3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include<iostream>

using namespace std;

void backtrack(int deep, int number, int a[]);

int main(){

int number;

int a[9];

cout<<"Enter a number : ";

cin>>number;

backtrack(0,number,a);

return 0;

}

void backtrack(int deep, int number, int a[]){

if(deep >= number){ //到底叶节点，输出结果

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

cout<<a[i]+1<<" ";

cout<<endl;

}

for(int i=0; i<number; i++){ //继续选择数字排列

bool flag = true;

for(int j=0; j<deep; j++){ //判断该数字是否已使用

if(a[j] == i)

flag = false;

}

if(flag){

a[deep] = i;

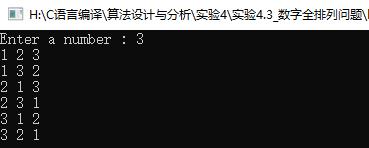
backtrack(deep+1,number,a);

}

}

}

（2）用户屏幕



实验5、《算法综合实验》

**一、实验目的**

1. 理解和复习所学各种算法的概念

2. 掌握和复习所学各种算法的基本要素

3. 掌握各种算法的优点和区别

4. 通过应用范例掌握选择最佳算法的设计技巧与策略

**二、实验内容**

1. 使用贪心算法、回溯法、分支限界法解决0-1背包问题；

2. 通过上机实验进行算法实现；

3. 保存和打印出程序的运行结果，并结合程序进行分析，上交实验报告。

**三、算法思想分析**

1.贪心算法理论上只能解决满足贪心选择性质的问题，而0-1背包并不满足该性质，所以并不能保证能够找到最优解法，只能找到最接近的解，当然如果运气好，也是可以找到最优解的。利用按重量从小到大、按价值从大到小、按价值/重量从大到小三种方式通过贪心算法求得每种方式的最终结果，并比较三种方式的最大价值取最大的那个，即为贪心算法获得的最优解。

2.回溯法解决0-1背包问题的解空间为子集树，利用回溯法的基本代码模版即可，其中左子树为约束条件，即背包能否装下该物品，右子树为限界条件，即当前物品不放入背包，剩余物品是否有可能创造比当前最大价值更大的价值，如果可以则进入右子树，反之，则直接剪去右子树。

3.0-1背包的解空间为子集树，分支界限法是采用广度优先搜索，每次选取队列的最前面的结点为活结点。

1）算法从根结点A即标记结点开始，初始时活结点队列为空，A入队列。

2）A为活结点，A的儿子结点B、C为可行结点。将B、C加入队列，舍弃A。此时队列元素为C-B；

3）B为活结点，B的儿子结点D、E,而D为不可行结点。将E入队列，舍弃B。此时队列元素为E-C；

4)循环以上步骤

按照以上方式扩展到叶节点。

**四、实验过程分析**

1.贪心算法的思路很简单即为一直循环下去，直至不满足指定条件。用于解决0-1背包问题时需要考虑多种放入方式，因为不管哪种方式都不能百分百会得到最优解，只能取多种放入方式中的最优解作为问题的最优解。

这道题目的收获在于贪心算法对于不能保证获得最优解的情况下，如何获得最接近的解，比如0-1背包问题则是采用多种放入方式再进行比较取最优解。

2.回溯法解决0-1背包问题的收获在于限界条件的运用，对于解空间为子集树的问题也有了进一步的了解。

3.从分支界限法解决0-1背包问题中，我更加懂得了队列的运用及标记结点的优势，本题中标记结点不仅代表着深度，而且保证队列不为空，当为空时循环结束。另外此题中的限界函数只有在最后的深度才发挥它的作用，在此之前一直没有到达底层， bestValue的数据一直为初始化数据0。

不过，还有一点不足就是没能实现背包获得最优价值时，所有物品的放入情况。

**五、算法源代码及用户屏幕**

1.（1）算法源码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

贪心算法解决0-1背包问题

codeblocks C++

2018.11.6

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

using namespace std;

//\*

//测试数据一，最大价值50,放置方式0,1，1

const int n = 3; //物品数量

int c = 30; //背包大小

int w[n] = {16, 15, 15}; //背包物品的重量

int v[n] = {45, 25, 25}; //背包物品的价值

//\*/

/\*

//测试数据二，最大价值15，放置方式1,1,0,0,1

const int n = 5; //物品数量

int c = 10; //背包大小

int w[n] = {2, 2, 4, 6, 5}; //背包物品的重量

int v[n] = {6, 3, 6, 5, 4}; //背包物品的价值

\*/

int value[3][n+1] = {0};

//二维数组记录三种放入方式的总价值和是否放入

//第一列为总价值，其余各列依次表示该物品是否放入

void weightG(); //按重量从轻到重重新定义物品放入顺序并求解

void valueG(); //按价值从大到小重新定义物品放入顺序并求解

void valueWeightG(); //按价值/重量重大到小重新定义物品放入顺序并求解

//贪心算法放入物品,number表示是哪种放入方式，indexArray[]表示加入物品的先后顺序

void greedyArrange(int number, int indexArray[]);

int main()

{

//按重量从轻到重定义物品放入顺序并求解

weightG();

//按价值从大到小定义物品放入顺序并求解

valueG();

//按价值/重量从大到小定义物品放入顺序并求解

valueWeightG();

//输出原始数据

cout<<"Greedy algorithm"<<endl;

cout<<"Things number : "<<n<<endl;

cout<<"Bag size : "<<c<<endl;

cout<<"Things weight : ";

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

cout<<w[i]<<" ";

}

cout<<endl;

cout<<"Things value : ";

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

cout<<v[i]<<" ";

}

cout<<endl<<endl;

//输出各种放入方式结果

//按重量从小到大

cout<<"Value(weight) : "<<value[0][0]<<endl;

cout<<"Arrange : ";

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

cout<<value[0][j]<<" ";

}

cout<<endl;

//按价值从大到小

cout<<"Value(value) : "<<value[1][0]<<endl;

cout<<"Arrange : ";

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

cout<<value[1][j]<<" ";

}

cout<<endl;

//按价值/重量从大到小

cout<<"Value(valueWeight) : "<<value[2][0]<<endl;

cout<<"Arrange : ";

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

cout<<value[2][j]<<" ";

}

cout<<endl<<endl;

//确定最大价值的放入方式

int maxValueNumber = 0; //数字记录哪种放入方式价值最大

int maxValue = value[0][0];

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

if(maxValue < value[i][0]){

maxValueNumber = i;

maxValue = value[i][0];

}

}

//输出最大价值放入方式及价值

cout<<"MaxValue : "<<value[maxValueNumber][0]<<endl;

cout<<"Best arrange : ";

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

cout<<value[maxValueNumber][j]<<" ";

}

cout<<endl;

return 0;

}

//按重量从轻到重重新定义物品放入顺序

void weightG(){

int tempSort[n]; //临时保存重量

int a[n]; //记录下标

//初始化数组

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

tempSort[i] = w[i];

a[i] = i;

}

//冒泡排序，按重量从小到大排序

for(int i=0; i<n-1; i++){

for(int j=0; j<n-1; j++){

if(tempSort[j] > tempSort[j+1]){

int temp = tempSort[j];

tempSort[j] = tempSort[j+1];

tempSort[j+1] = temp;

temp = a[j];

a[j] = a[j+1];

a[j+1] = temp;

}

}

}

greedyArrange(0, a);

}

//按价值从大到小重新定义物品放入顺序并求解

void valueG(){

int tempSort[n]; //临时保存价值

int a[n]; //记录下标

//初始化数组

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

tempSort[i] = v[i];

a[i] = i;

}

//冒泡排序，按价值从大到小排序

for(int i=0; i<n-1; i++){

for(int j=0; j<n-1; j++){

if(tempSort[j] < tempSort[j+1]){

int temp = tempSort[j];

tempSort[j] = tempSort[j+1];

tempSort[j+1] = temp;

temp = a[j];

a[j] = a[j+1];

a[j+1] = temp;

}

}

}

greedyArrange(1, a);

}

//按价值/重量重大到小重新定义物品放入顺序并求解

void valueWeightG(){

double tempSort[n]; //临时保存价值/重量

int a[n]; //记录下标

//初始化数组

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

tempSort[i] = (double)v[i]/(double)w[i];

a[i] = i;

}

//冒泡排序，按价值/重量从大到小排序

for(int i=0; i<n-1; i++){

for(int j=0; j<n-1; j++){

if(tempSort[j] < tempSort[j+1]){

int temp = tempSort[j];

tempSort[j] = tempSort[j+1];

tempSort[j+1] = temp;

temp = a[j];

a[j] = a[j+1];

a[j+1] = temp;

}

}

}

greedyArrange(2, a);

}

//贪心算法放入物品

void greedyArrange(int number, int indexArray[]){

int cBag = c;

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

if(cBag >= w[indexArray[i]]){

value[number][indexArray[i]+1] = 1;

cBag -= w[indexArray[i]];

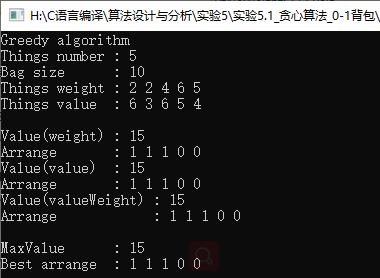
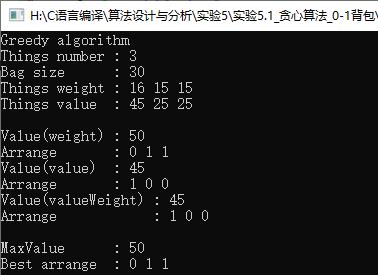
value[number][0] += v[indexArray[i]];

}

}

}

（2）用户屏幕



2.（1）算法源码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

回溯法解决0-1背包问题

codeblocks C++

2018.11.5

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

using namespace std;

void backTrack(int deep, int n, int c, int w[], int v[]); //deep为树的深度，n为物品数量，c为背包大小，w[]为物品重量，v[]为物品价值

int bound(int deep, int n, int c, int w[], int v[]); //限界函数

int arrangCurrent[100] = {0}; //当前放置方式

int bestArrange[100] = {0}; //最佳放置方式

int valueCurrent = 0;

int maxValue = 0;

int main()

{

//\*

//测试数据一，最大价值50,放置方式0,1，1

int n = 3; //物品数量

int c = 30; //背包大小

int w[n] = {16, 15, 15}; //背包物品的重量

int v[n] = {45, 25, 25}; //背包物品的价值

//\*/

/\*

//测试数据二，最大价值15，放置方式1,1,0,0,1

int n = 5; //物品数量

int c = 10; //背包大小

int w[n] = {2, 2, 4, 6, 5}; //背包物品的重量

int v[n] = {6, 3, 6, 5, 4}; //背包物品的价值

\*/

//初始化都不放置

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

arrangCurrent[i] = 0;

bestArrange[i] = 0;

}

backTrack(0, n, c, w, v);

cout<<"Things number : "<<n<<endl;

cout<<"Bag size : "<<c<<endl;

cout<<"Things weight : ";

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

cout<<w[i]<<" ";

}

cout<<endl;

cout<<"Things value : ";

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

cout<<v[i]<<" ";

}

cout<<endl<<endl;

cout<<"MaxValue : "<<maxValue<<endl;

cout<<"Best arrange : ";

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

cout<<bestArrange[i]<<" ";

}

cout<<endl;

return 0;

}

void backTrack(int deep, int n, int c, int w[], int v[]){

if(deep >= n){

if(valueCurrent > maxValue){

//当前价值大于最大价值，更新放置方式及最大价值

maxValue = valueCurrent;

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

bestArrange[i] = arrangCurrent[i];

}

}

}

else{

if(w[deep] <= c){ //搜索左子树，约束条件

arrangCurrent[deep] = 1;

c -= w[deep];

valueCurrent += v[deep];

backTrack(deep+1, n, c, w, v);

arrangCurrent[deep] = 0;

c += w[deep];

valueCurrent -= v[deep];

}

if(bound(deep+1, n, c, w, v) > maxValue){ //搜素右子树

arrangCurrent[deep] = 0;

backTrack(deep+1, n, c, w, v);

}

}

}

int bound(int deep, int n, int c, int w[], int v[]){

int left = c; //剩余重量

int valueTemp = valueCurrent; //当前价值

while(deep<n && w[deep]<=left){

left -= w[deep];

valueTemp += v[deep];

deep++;

}

if(deep<n){

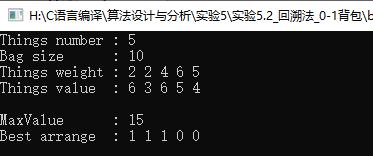
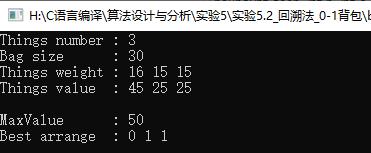
valueTemp += left\*(v[deep]/w[deep]);

}

return valueTemp;

}

（2）用户屏幕



3.（1）算法源码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

队列式分支界限法解决0-1背包问题

codeblocks C++

2018.11.6

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include <queue>

using namespace std;

//每个结点包含当前重量、价值、深度、是否放入

typedef struct treeNode{

int weight;

int value;

int treeDeep;

int flag;

}treeNode;

//创建队列，存放活结点数据

queue<struct treeNode> liveNodeQueue;

//\*

//测试数据一，最大价值50,放置方式0,1，1

const int n = 3; //物品数量

int c = 30; //背包大小

int w[n] = {16, 15, 15}; //背包物品的重量

int v[n] = {45, 25, 25}; //背包物品的价值

//\*/

/\*

//测试数据二，最大价值15，放置方式1,1,0,0,1

const int n = 5; //物品数量

int c = 10; //背包大小

int w[n] = {2, 2, 4, 6, 5}; //背包物品的重量

int v[n] = {6, 3, 6, 5, 4}; //背包物品的价值

//\*/

int bestValue = 0;

//活结点则加入到队列中

void joinQuenen(int weight, int value, int deep, int flag);

void knap(); //广度搜索

int bound(int deep, int cBag, int value); //限界条件

int main()

{

knap();

//输出背包物品相关信息

cout<<"Branch-bound Algorithm"<<endl;

cout<<"Things number : "<<n<<endl;

cout<<"Bag size : "<<c<<endl;

cout<<"Things weight : ";

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

cout<<w[i]<<" ";

}

cout<<endl;

cout<<"Things value : ";

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

cout<<v[i]<<" ";

}

cout<<endl<<endl;

//输出结果

cout<<"MaxValue : "<<bestValue<<endl;

return 0;

}

//活结点则加入到队列中

void joinQuenen(int weight, int value, int deep, int flag){

treeNode node;

node.weight = weight;

node.value = value;

node.treeDeep = deep;

node.flag = flag;

if(deep == n){

//到底最底层,不加入队列

if(node.value > bestValue){

bestValue = node.value;

}

}

else{

//不是最底层，加入队列

liveNodeQueue.push(node);

}

}

//广度搜索

void knap(){

int deep = 1; //表示广度搜索的深度

treeNode tag, liveNode; //tag结点表示为标记结点，表示是否在同一深度，livenode结点表示当前的扩展结点

//队列的起始结点

tag.weight = -1; //用于区别其他结点，判断是否进入下一深度

tag.value = 0;

tag.treeDeep = 0; //表示广度搜索的深度

tag.flag = 0;

//初始化活结点

liveNode.weight = 0;

liveNode.value = 0;

liveNode.treeDeep = 1;

liveNode.flag = 0;

liveNodeQueue.push(tag);

while(1){

//约束条件，判断左子树是否可行

if(liveNode.weight + w[deep-1] <= c){

joinQuenen(liveNode.weight+w[deep-1], liveNode.value+v[deep-1], deep, 1);

}

//限界条件，判断右子树是否可行

if(bound(deep, liveNode.weight, liveNode.value) > bestValue){

joinQuenen(liveNode.weight, liveNode.value, deep, 0);

}

//记录并去掉队列最前面的结点

liveNode = liveNodeQueue.front();

liveNodeQueue.pop();

//判断是否为同一深度

if(liveNode.weight == -1){

//活结点表为空

if(liveNodeQueue.empty()){

break;

}

//获取下一个活结点，并从队列中去掉

liveNode = liveNodeQueue.front();

liveNodeQueue.pop();

//加入标记结点，表明是同一深度

liveNodeQueue.push(tag);

//深度加1

deep++;

}

}

}

//限界条件,当做背包问题求最大价值

int bound(int deep, int cBag, int value){

int left = c - cBag; //剩余重量

int valueTemp = value; //当前价值

while(deep<n && w[deep]<=left){

left -= w[deep];

valueTemp += v[deep];

deep++;

}

if(deep<n){

valueTemp += left\*(v[deep]/w[deep]);

}

return valueTemp;

}

（2）用户屏幕

