西安交通大学

# 算法设计与分析实验报告

班级： 计算机72班 姓名： 徐子越 学号： 2174410455

**１＞必做题**

问题描述

设有n个互不相同的元素x1,x2,…, xn，每个元素xi带有一个权值wi，且。若元素xk满足 且，则称元素xk为x1,x2,…, xn的带权中位数。请编写一个算法，能够在最坏情况下用O(n)时间找出n个元素的带权中位数。

（１）问题分析

本题目采用分治思想的线性时间选择中位数算法可以解决。只不过从单纯的元素大小中位数变成了要考虑元素权值的中位数。

（２）算法设计

每个元素有一个value以及一个weight。

设置find = 1/2，权值总和total = 1;

首先将元素分为ｘ个一组，一共ｎ/ｍ＋１组，在组中使用快速排序进行排序，再将第ｉ组（ｉ∈[０～ｎ/m+1]）中的中位数（元素序号为ｉx＋x/2+1）与整个元素的第ｉ个进行交换。然后再对整个元素数组的前n/m+1进行如上相同操作，直到剩下一个数，就是要找的元素大小的中位数。

找到中位数之后（记为ｍ），位置此时应该在第０位。再对整个元素数组遍历，将不比ｍ的value小的元素排在ｍ的前面，比ｍ的value大的排在ｍ后面。

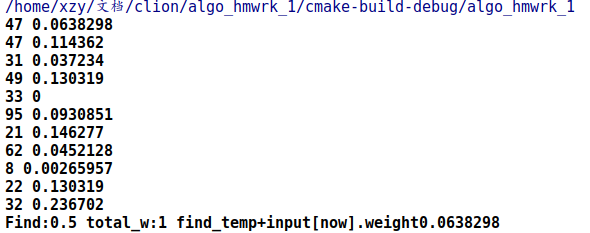
排完序后设ｍ的位置位now。

计算∑(ｍ左侧元素.weight)，如果大于find，则将元素数组ｍ之后（包括ｍ）删掉,find不变，total = ∑(ｍ左侧元素.weight)，然后从头开始；否则，计算｛∑(m.value左侧元素.weight)＋ｍ.weight｝，如果小于find，则将ｍ之前，包括ｍ都删掉，从头开始，find =find -｛∑(m.value左侧元素.weight)＋ｍ.weight｝；否则如果∑(ｍ左侧元素.weight)<=find&&｛∑(m.value左侧元素.weight)＋ｍ.weight｝>=find,则返回当前的now值。这就是答案。

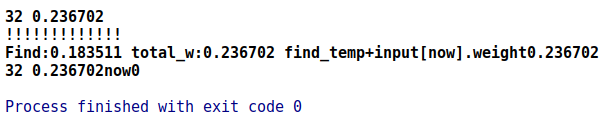
（３）算法实现

#include <iostream>  
#include <vector>  
#include <cstdlib>  
#include <ctime>  
#include <algorithm>  
using namespace std;  
double Find = 0.5;  
int n = 11;  
double total\_w = 1;  
int area = 5;  
class encapsulation{  
public:  
 int value;  
 double weight;  
};  
vector<encapsulation> input(n);  
template <typename T> void init(vector<T>& temp){  
 int i = 0;  
 srand((unsigned)time(NULL));  
 double total = 0, t = 0;  
 for(i = 0; i<n; i++) {  
 temp[i].value = rand() % 101;  
 temp[i].weight = rand() % 97;  
 total += temp[i].weight;  
 }  
 for(i = 0; i<n-1; i++){  
 temp[i].weight = temp[i].weight/total;  
 t += temp[i].weight;  
 }  
 temp[n-1].weight = 1-t;  
 /\*if(is\_same<T,double>::value){ //如果检测为权值数组，则赋值为０～１之间的double  
 T t = 0; //is\_same<T,double>::value判断两种类型是否相同，返回１  
 for(i = 0; i<n; i++)  
 totle += temp[i];  
 for(i = 0; i<n-1; i++){  
 temp[i] = temp[i]/totle;  
 t += temp[i];  
 }  
 temp[n-1] = 1-t;  
 }\*///原本用来初始化两个vector，现在取消了，直接用自建class  
}  
bool Less(const encapsulation& a, const encapsulation& b){  
 return a.value < b.value;  
}//用于自建类的比较函数  
template <typename T> int n\_sort(vector<T>& temp, int t){  
 if(t == n/area){  
 sort(temp.begin()+t\*area,temp.end(), Less);  
 return 0;  
 }//对数组末尾不满area排序  
 sort(temp.begin()+t\*area,temp.begin()+(t+1)\*area, Less);  
}  
  
template <typename T> int linearset(vector<T>& temp){  
 int sign = 0;  
 for(int i = 1; i<n; i++){  
 if(temp[i].value<=temp[sign].value){  
 swap(temp[i],temp[sign+1]);  
 swap(temp[sign],temp[sign+1]);  
 sign++;  
 cout<<"\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;  
 for(int i = 0;i < n;i++)  
 cout<<input[i].value<<"这是ｌｉｎｅａｒｓｅｔ"<<input[i].weight<<endl;  
 }  
 }  
 return sign;  
}  
  
int main() {  
 init(input);  
 for(int i = 0;i < n;i++)  
 cout<<input[i].value<<" "<<input[i].weight<<endl;  
 //线性时间选择带权中位数  
 double find\_temp;  
 int x = 10;  
 int j, now = 0;//记录当前中间value所在位置  
 cout<<"Find:"<<Find<<" total\_w:"<<total\_w<<" find\_temp+input[now].weight"  
 <<find\_temp+input[now].weight<<endl;  
 while(true){  
 j = n/area;  
 cout<<"shit"<<n<<endl;  
 find\_temp = 0;  
 while(j>0){  
 for(int i = 0; i <= j; i++){  
 n\_sort(input, i);  
 if(i<n/area)  
 swap(input[i],input[i\*area+2]);  
 else swap(input[i], input[(i\*area+n)/2]);  
 j /= area;  
 }  
 }  
 cout<<"\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;  
 for(int i = 0;i < n;i++)  
 cout<<input[i].value<<" "<<input[i].weight<<endl;  
 now = linearset(input);  
 cout<<"\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;  
 for(int i = 0;i < n;i++)  
 cout<<input[i].value<<" "<<input[i].weight<<endl;  
 for(int i =0; i < now; i++) {  
 find\_temp += input[i].weight;  
 }  
 if(find\_temp > Find){  
 Find = Find;  
 total\_w = find\_temp;  
 copy(input.begin(),input.begin()+now-1,input.begin());  
 input.resize(now-1);  
 cout<<"\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;  
 for(int i = 0; i < now; i++){  
 cout<<input[i].value<<">"<<endl;  
 }  
 n = now;  
 cout<<"Find:"<<Find<<" total\_w:"<<total\_w<<" find\_temp"  
 <<find\_temp<<"now"<<now<<endl;  
 }  
 else if (find\_temp + input[now].weight < Find){  
 Find = Find - find\_temp - input[now].weight;  
 total\_w = total\_w-find\_temp-input[now].weight;  
 copy(input.begin()+now+1,input.end()+1,input.begin());  
 input.resize(n-now-1);  
 cout<<"\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;  
 for(int i = 0; i < input.size(); i++){  
 cout<<input[i].value<<"<"<<endl;  
 }  
 n = n-now-1;  
  
 cout<<"Find:"<<Find<<" total\_w:"<<total\_w<<"find\_temp"  
 <<find\_temp<<"now"<<now<<endl;  
 }  
 else if((find\_temp <= Find) && (find\_temp + input[now].weight >= Find)){  
 cout<<"!!!!!!!!!!!!!"<<endl;  
 cout<<"Find:"<<Find<<" total\_w:"<<total\_w<<" find\_temp+input[now].weight"  
 <<find\_temp+input[now].weight<<endl;  
 cout<<input[now].value<<" "<<input[now].weight<<"now"<<now<<endl;  
 break;  
 }  
 }  
 return 0;  
}

（４）运行结果

输入：

结果：



**２＞必做题**

设有一个长度为L的钢条，在钢条上标有n个位置点（p1,p2,…,pn）。现在需要按钢条上标注的位置将钢条切割为n+1段，假定每次切割所需要的代价与所切割的钢条长度成正比。请编写一个算法，能够确定一个切割方案，使切割的总代价最小。

1. 问题分析

显然，这道题符合最优子问题的性质。一个大纲条的最优切割是由两个小钢条最优组合而成的。

可以采用分解子问题的方法，将重复数据记录下来，避免重复计算，这就是动态规划。将最短子钢条两个两个地组合判断相邻两个子钢条的最佳组合情况，自下而上地递归计算上层最佳组合，循环ｎ次。途中记录组合的选择情况。

1. 算法设计

输入：vector<double>cut\_p[n];

过程vector：vector<int>rst[n][n]; vector<double>dp[n][n];

算法过程：

①i = 0，　ｘ＝０;

构建dp[i]层，当ｉ为０的时候dp层为最小子钢条。

构建rst[i]层，当ｉ为０的时候rst层为断点序号。

②

构建rst[i]层，构建方法是：dp[i][x] = min{dp[i][x-1],dp[i+1][x-1]}

如果选择左边，则rst[i].insert(rst[i].end(), i+dif);

如果选择右边，则rst[i] = rst[i+1];

rst[i].insert(rst[i].end(),i);

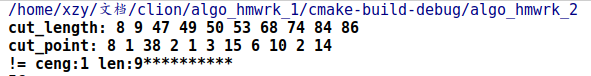
③循环①～②步骤，知道ｉ＝＝ｎ；输出rst与dp

1. 算法实现

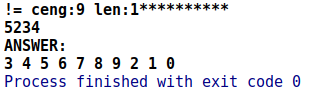
#include <iostream>  
#include <vector>  
#include <algorithm>  
using namespace std;  
  
double L = 100;  
int n = 10;  
vector<double> cut\_p;  
void init(vector<double>& temp){  
 //给切割点  
 srand((unsigned)time(NULL));  
 for(int i = 0; i<n; i++)  
 cut\_p.push\_back(rand()%100);  
 sort(cut\_p.begin(),cut\_p.end());  
 temp.push\_back(cut\_p[0]-0);  
 for(int i = 1; i<n; i++)  
 temp.push\_back(cut\_p[i]-cut\_p[i-1]);  
 temp.push\_back(L-cut\_p[n-1]);  
 cout<<"cut\_length: ";  
 for(int i = 0; i<n; i++){  
 cout<<cut\_p[i]<<" ";  
 }  
 cout<<endl;  
 cout<<"cut\_point: ";  
 for(int i = 0; i<n; i++){  
 cout<<temp[i]<<" ";  
 }  
 cout<<temp[n]<<endl;  
 }  
void setdp(vector<double>& temp, vector<double>& temp\_source, vector<double>& temp0, vector<double> \*rst){  
 int source\_len = temp\_source.size(), len = source\_len-1;  
 int dif = n - len;  
 if(len == n){  
 for(int i = 0; i<len; i++){  
 temp.push\_back(temp\_source[i]+temp\_source[i+1]);  
 rst[i].push\_back(i);  
 }  
 /\*cout<<"!="<<" ceng:"<<0<<"\*\*\*\*\*\*\*\*\*\*"<<endl;  
 for(int i = 0; i < temp.size();i++)  
 cout<<temp[i]<<endl;\*/  
 return;  
 }  
 for(int i = 0; i<len; i++){  
 if(temp\_source[i]<temp\_source[i+1]){  
 temp.push\_back(temp\_source[i]\*2+temp0[i+dif+1]);  
 rst[i].insert(rst[i].end(), i+dif);  
  
 }  
 else if(temp\_source[i]>=temp\_source[i+1]){  
 temp.push\_back(temp0[i]+temp\_source[i+1]\*2);  
 rst[i] = rst[i+1];  
 rst[i].insert(rst[i].end(),i);  
 }  
 }  
 /\*cout<<"SSSSSSSSSSSSSSSSS"<<endl;  
 for(int i = 0;i<rst[0].size();i++)  
 cout<<rst[0][i]<<endl;\*/  
 cout<<"!="<<" ceng:"<<dif<<" len:"<<len<<"\*\*\*\*\*\*\*\*\*\*"<<endl;  
 for(int i = 0; i < temp.size();i++)  
 cout<<temp[i]<<endl;  
}  
int main(){  
 vector<double> rst[n];  
 vector<double> dp[n+1];  
 init(dp[0]);  
 for(int i = 1; i<n+1; i++)  
 setdp(dp[i],dp[i-1], dp[0], rst);  
 cout<<"ANSWER:"<<endl;  
 for(int i = 0;i<rst[0].size();i++)  
 cout<<rst[0][i]<<" ";  
 return 0;  
}

1. 运行结果

输入：



答案输出：



**3>贪心问题**

有n个作业和m台相同设备，每个作业i可选择一台设备进行加工，加工时间为ti。每台机器一次只能加工一个作业。编写算法，实现对作业的调度，使得n个作业的等待时间和最小。

1. 问题分析

先解决最小的作业，再解决最大的作业。

贪心选择性质证明：

每一次选择都是贪心的，选的是耗费最小的task。如果不选耗费最小的task也能得到最优解，那么选择耗时最小的task费时更小，因为小task可以先做完，大task等待了小task的耗时这是比小task等待大task耗时小的。故不选耗费最小的task得不到最优解。

最优子结构证明：

没被放进machine的task的等待时间总是从完成的task的耗时上累加。故最短的总耗时由最短分耗时累加而成。满足最优子结构。

1. 算法设计

随机创造一个task[n]序列和machine[m]序列，然后对task排序使之变成一个栈顶小于栈底的有序栈。

开始-->当一个machine为空接收task，直到machine都装了task，则开始计时。

计时按微妙为单位，每一次计时，machine中的task的耗时都减一。每一次计时之后都要判断machine序列中是否有空machine即该machine的task减至0，如果有，则该machine从task栈中取新task继续工作。

循环进行“开始-->”直到task为空。

输出计时时间time

（3）算法实现

#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <ctime>  
#include <unistd.h>  
using namespace std;  
int n = 10;  
int m = 5;  
vector<int> task(n);

//task列表  
vector<int> machine(m);

//machine列表  
int main(){  
 srand((unsigned)time(NULL));

//生成task参数  
 for(int i = 0; i<n; i++)  
 task.push\_back(rand()%100);  
 sort(task.begin(),task.end());

//task时间从小到大排序  
 reverse(task.begin(),task.end());

//task翻转成从大到小排序  
 int t = 0;  
 while(true){

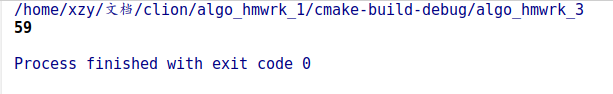
//将task从小到大放入machine，直到task为空  
 for(int i = 0;i<m;i++){  
 if(machine[i]>0){

//机器在运行的话，就减一秒  
 machine[i]-=1;  
 }  
 else{  
 if(task.size()>0){

//机器不在运行的话，就往机器放task  
 machine[i] = task.back();  
 task.pop\_back();  
 }  
 }  
 }  
 usleep(1);  
 t+=1;  
 if(task.size()<=0)break;  
 }  
 cout<<t<<endl;

//运行时间  
 return 0;  
}

（4）运行结果如下：



**4>必做题**

给定1个1000行×20列的0-1矩阵，对于该矩阵的任意1列，其中值为1的元素的数量不超过10%。设有两个非空集合A和B，每个集合由矩阵的若干列组成。集合A和B互斥是指对于矩阵的任意一行，同时满足下列2个条件：1）若A中有一个或多个元素在这一行上的值是1，则B中的元素在这一行全部是0；2）若B中有一个或多个元素在这一行上的值是1，则A中的元素在这一行全部是0。请你设计一个算法，找出一对互斥集合A和B，使得A和B包含的列的总数最大。

1. 问题分析

这是一个回溯问题，在这道题中，我们假设没有一列全是0，否则答案就是20。因为没有全是0的一列，故A、B集合不相等。那么只要对A集合使用子集树进行回溯地选择应该添加那一列进A集合。当A集合添加完毕后，再对剩下的列进行判断，判断出那些能进入B即可。最后输出回溯后的|A|+|B|的最大值。

1. 算法设计

设计一个完全二叉树，一共有21层即-1~19。从根节点开始，每个节点有成员变量level，判断是否添加进A的add变量，以及左子树和右子树。其中左子树的add为1，右子树add为0，分别对应添加和不添加进A集合。

然后对创建的二叉树进行先序遍历。当遍历到第19层则运行当前节点的check（）用于得出B集合。

当递归遍历完成后即可得到最大|A|+|B|

（3）算法实现

#include <iostream>  
#include <vector>  
#include <cstdlib>  
using namespace std;  
vector<int> row; //储存A集合选择的列  
vector<vector<int>> row\_temp; //递归的时候储存row  
vector<int> rst(1000,0); //A集合或与浓缩成一个int[1000]数组，用于约束B集合  
vector<vector<int>> rst\_temp; //递归的时候储存rst  
int matrix[1000][20] = {0}; //题目给出的[1000][20]矩阵  
int Max = 0; //答案  
  
class Node{ //子集树的节点  
public:  
 int add = 1; //判断当前列是否加入row，左子树为１添加，右子树为０不添加  
 int level; //当前列  
 Node\* left; //左子树  
 Node\* right; //右子树  
public:  
 void ini(){  
 if(this->level<=19) {  
 this->left = new Node(this->level, 1);  
 this->right = new Node(this->level, 0);  
 }  
 else{  
 this->left = NULL;  
 this->right = NULL;  
 }  
 }  
 Node(int a, int r){ //节点初始化，如果是第１９层则不创建子树  
 this->add = r;  
 this->level = a+1;  
 this->ini();  
 }  
 ~Node(){  
 }  
public:  
 void check(){ //在深度优先搜索到１９层时意味着A集合完毕．check()判断B集合  
 if(row.size()>=20||row.size()<=0) //简单的剪枝函数  
 return;  
 vector<int> temp0;  
 temp0.assign(row.begin(),row.end());  
 int b = 0;  
 int temp = temp0.size();  
 for(int i=19;i>=0;i--){ //判断A集合以外的列，如果符合约束则加入B集合，大小为ｂ  
 if(i!=temp0.back()){  
 int j=0;  
 for(j = 0;j<1000;j++){  
 if((matrix[j][i]&&rst[j])==1){  
 break;  
 }  
 if(j>=999&&temp<=19){  
 temp++;  
 b++;  
 }  
 }  
 }  
 else{  
 temp0.pop\_back();  
 }  
 }  
 if(b>0){ //如果B集合非空,且b+row.size()大于Max,则更新Max  
 if(temp>Max){  
 Max = temp;  
 cout<<"bigger answer:"<<Max<<endl;  
 }  
 }  
 return;  
 }  
};  
void setnew(){ //在A集合加入一列之后更新rst  
 for(int i = 0;i<1000;i++)  
 rst[i]=rst[i]||matrix[i][row.back()];  
}  
void depth\_first\_tree\_search(Node \*v) { //深度优先搜索  
 if (v->level >= 19) {  
 v->check();  
 return;  
 }  
 if (v->add == 1) {  
 if(row.size()<=0||v->level>row.back())  
 row.push\_back(v->level);  
 setnew();  
 }  
 row\_temp.push\_back(row); //row进栈  
 rst\_temp.push\_back(rst); //rst出栈  
 depth\_first\_tree\_search((Node \*) v->left);//搜索左子树  
 row = row\_temp.back(); //row出栈  
 row\_temp.pop\_back();  
 rst = rst\_temp.back(); //rst出栈  
 rst\_temp.pop\_back();  
 depth\_first\_tree\_search((Node \*) v->right);//搜索右子树  
 return;  
 }  
void init(){ //初始化题目矩阵  
 cout << "init" << endl;  
 srand((unsigned)time(NULL));  
 int t = 0;  
 for(int i = 0; i<20; i++){  
 t = (rand()%349+1);  
 for(int j = 0; j<t; j++){  
 matrix[rand()%1000][i] = 1;  
 }  
 }  
 for(int i = 0;i<1000;i++) //cout矩阵  
 {  
 cout<<endl;  
 for(int j = 0;j<20;j++)  
 cout<<matrix[i][j];  
 }  
 cout<<endl;  
}  
int main(){  
 init();  
 Node \*root = new Node(-1, 0);  
 depth\_first\_tree\_search((Node \*) root);//从根节点开始搜索  
 cout<<Max;  
 return 0;  
}

（4）运行结果

