**实验6 分支限界法实现**

姓名：叶倩琳

班级：软工1706

完成时间：2019/06/06

**一、实验目标：**

1. 熟悉分支限界法应用场景及实现的基本方法步骤；

2. 学会分支限界法的实现方法和分析方法：

**二、实验内容**

1. n后问题：

编程计算当n=1到8时解的个数，分别利用回溯法和分支限界法实现，比较并分析你的算法效率。

2. 单源最短路径问题：



如图求出从源顶点0到其它顶点的最短路径长度，比较贪心算法和分支限界法。

3. 二分搜索问题：

对于给定的有序整数集a={6，9，13，15，25，33，45}，编写程序查找18与33，记录每次比较基准数。

1. 对快速排序进行随机化改造，比较不同输入下改造前后的性能。

**三、实验内容**

**问题一**

**1、问题描述**

n后问题：

编程计算当n=1到8时解的个数，分别利用回溯法和分支限界法实现，比较并分析你的算法效率。

**2、代码实现**

（1）回溯法

#include <iostream>

#include<cmath>

using namespace std;

/\* 回溯法求解 n后问题 \*/

#define MAXN 1000

int n; //皇后个数

int sum=0; //可行解个数

int x[MAXN]; //当前解

bool Place(int k){

//与前面放好的皇后进行比较（不在同一斜线、同一列上）

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

if((abs(k-j)==abs(x[j]-x[k])) || (x[j]==x[k]))

return false;

}

return true;

}

void Backtrack(int t){

if(t>n)

sum++; //如果到达叶子节点，解的个数增加

else{

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

x[t]=i; //表示 t 行皇后的位置放在 i 处

//如果这个位置能放，则进行下一层的遍历

if(Place(t))

Backtrack(t+1);

}

}

}

int main() {

sum=0;

while(1){

cout<<"n后问题\n请输入皇后个数："<<endl;

cin>>n;

//初始化

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

x[i]=0;

Backtrack(1);

cout<<"回溯法求得"<<n<<"后问题的解的个数为："<<sum<<endl;

sum=0;

cout<<endl;

}

return 0;

}

（2）分支限界法

#include<iostream>

#include<cmath>

#include<queue>

using namespace std;

/\*分支限界法（队列）求解 n后问题\*/

class Node{

public:

//初始化

Node(int n) : t(0), n(n){

x = new int[n + 1];

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

x[i] = 0;

}

Node(const Node& node){

t = node.t;

n = node.n;

x = new int[node.n + 1];

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

x[i] = node.x[i];

}

~Node(){

if (x != NULL){

delete[] x;

x = NULL;

}

}

bool check(int k);

int t; //当前已经放置的皇后个数

int n; //皇后数量

int \*x; //存放皇后位置，表示第i行的皇后放在 x[i] 处

};

//检查放在第k列的皇后是否与前面已放的皇后有冲突

bool Node::check(int k){

//与已经放置皇后的行进行比较

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

//不在同一列或同一对角线

if(x[i]==k || abs(k-x[i])==abs(t-i+1))

return false;

}

return true;

}

int QueenArrange(int n){

int sum=0;

queue<Node> q; //建立皇后节点的队列 q

Node firstnode(n);

q.push(firstnode);

while (!q.empty()){ //若队列非空

Node node = q.front();

q.pop();

if (node.t == n)

sum++; //解的总数加一

//BFS，将符合条件的插入队列

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

//剪枝，只有符合check()函数的皇后节点才考虑，加入队列 q

if (node.check(i)){

Node newnode(node); //拷贝node节点给newnode

newnode.t++;

newnode.x[newnode.t] = i; //记录位置

q.push(newnode);

}

}

}

return sum;

}

int main(){

int n;

while(1){

cout<<"n后问题\n请输入皇后个数："<<endl;

cin>>n;

cout<<"分支限界法求得"<<n<<"后问题的解的个数为："<<QueenArrange(n)<<endl;

cout<<endl;

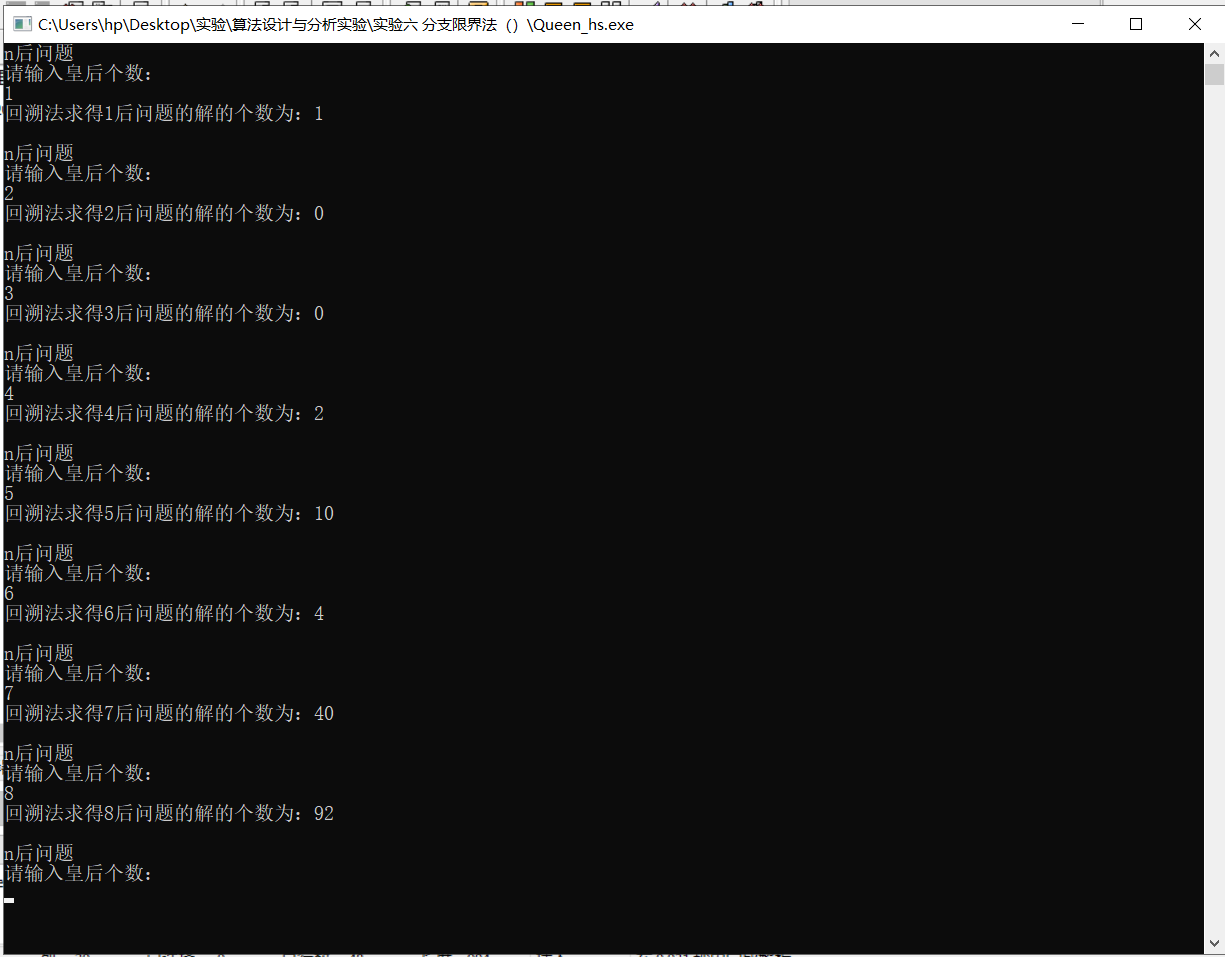
}

return 0;

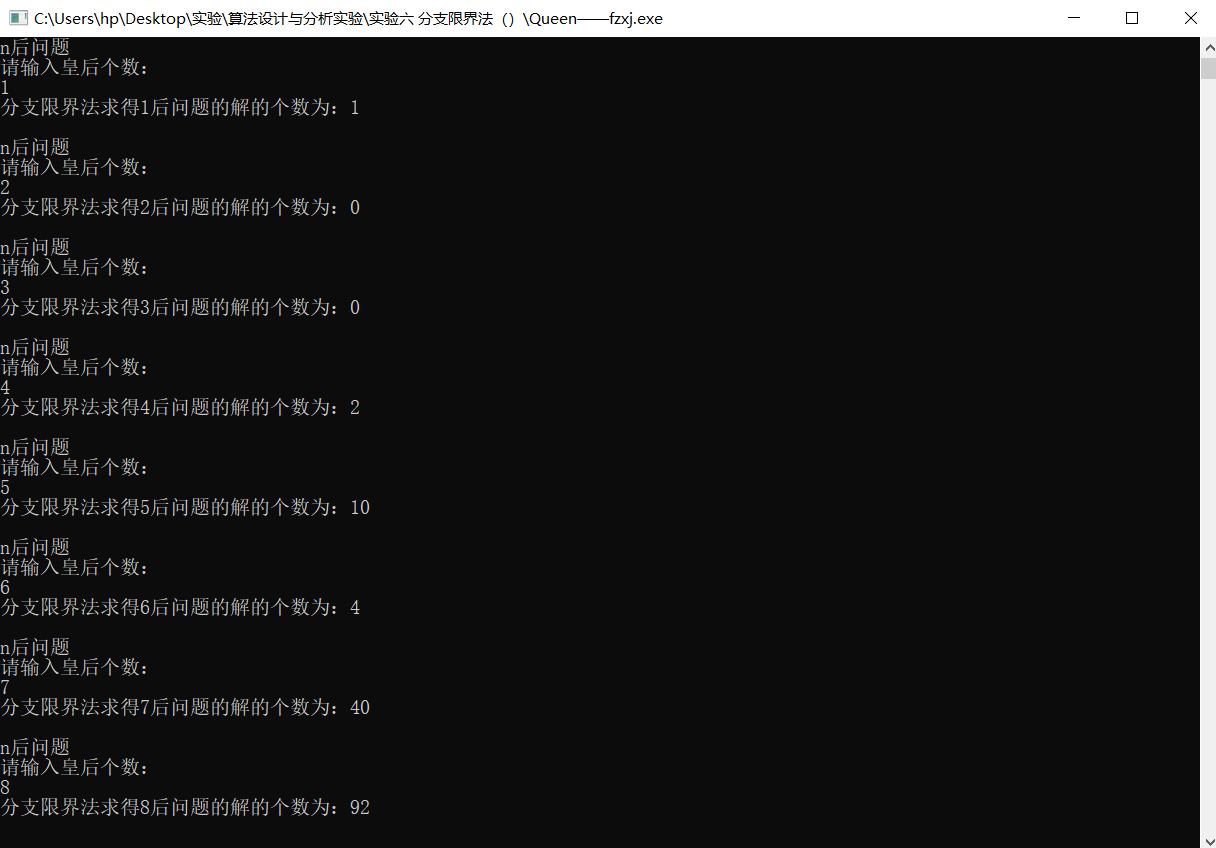
}

**3、输出结果**

（1）回溯法



（2）分支限界法



**4、性能分析**

（1）回溯法

时间复杂度：递归n行， 每次循环 n 列， 比较 0～n-1 次。 n \* n \* (n - 1) / 2 = O（n^3）

空间复杂度：只用了数组 x[MAXN] ，因此空间复杂度为 O(n)

（2）分支限界法

空间复杂度：其解空间是一棵完全二叉树 ，因此空间复杂度为 O(n\*n)

**5、实验体会**

（1）回溯法

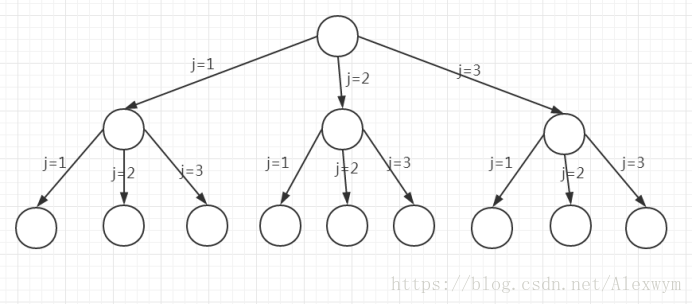
回溯法就是按选优条件向前搜索，以达到目标，但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择。

本题中回溯法通过引入数组x[MAXN]来存放当前解，x[t]=i 表示第 t 行的皇后放在第 i 列上，并通过 if((abs(k-j)==abs(x[j]-x[k])) || (x[j]==x[k])) 把隐约束（各皇后不在同一斜线、同一列上）变成了显约束。用回溯法解n后问题时，用完全二叉树表示解空间，并用可行约束Place以深度优先的方式递归搜索，剪去不满足行、列和斜线约束的子树。

（2）分支限界法

分支限界法中，每一个活结点只有一次机会成为扩展节点，且活结点一旦成为扩展节点，就一次性产出所有儿子节点，对所有的儿子节点进行剪枝，满足条件的儿子结点被加入到活结点表（队列）中。

用分支限界来解决n皇后问题，本问题它满足两种树结构。由于每一个合适的放置点出现最优解的概率是相等的，因此不需要使用优先队列，直接使用队列queue。并设定行、列、对角线约束条件，于是它的子集树空间结构如下：



**问题二**

**1、问题描述**

单源最短路径问题：



如图求出从源顶点0到其它顶点的最短路径长度，比较贪心算法和分支限界法。

**2、代码实现**

（1）贪心算法

#include<iostream>

#include<cstring>

#define MAXN 1000

using namespace std;

/\*单源最短路径问题 贪心求解（Dijstra）\*/

/\*n是顶点数，v是源，

dist[i]表示当前从源到顶点i的最短特殊路径长度,

prev[i]=j：最短路径中顶点i的前一个顶点是j

\*/

void dijkstra(int n,int v, int a[][MAXN], int dist[], int prev[]) {

if(v<0 || v>n)

return;

bool s[n+1]; //判断是否已存入该点到集合中

// 初始化

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

dist[i]=a[v][i]; //从顶点0到顶点i的路径长度

s[i]=false;

if(dist[i]==MAXN){

prev[i]=-1;

}

else

prev[i]=v;

}

dist[v]=0;

s[v]=true; //把顶点0加入集合中

// 循环n-1次，找出当前未加入集合的顶点j的dist[j]最小

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

int temp=MAXN;

int u=v;

for(int j=1;j<n;j++) { // 寻找不在集合内且距离集合最近的节点 j

if((!s[j]) && (dist[j]<temp)){

u=j;

temp=dist[j];// 记录最短特殊路径长度

}

}

s[u] = true;// 将节点u放入集合

//更新dist[]和prev[]的值(只要更新和新加入的顶点u相邻的顶点即可)

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

if ((!s[j]) && (a[u][j]<MAXN)){ //寻找不在集合内，且可达的节点

int newdist=dist[u]+a[u][j];

if(newdist<dist[j]) { // 与旧值进行比较，保留小的值

dist[j]=newdist;

prev[j]=u;

}

}

}

}

}

int main() {

//边权

int c[][MAXN]={{MAXN,MAXN,20,5,MAXN},

{MAXN,MAXN,30,20,MAXN},{MAXN,MAXN,MAXN,30,MAXN},

{MAXN,MAXN,MAXN,MAXN,10},{MAXN,MAXN,MAXN,MAXN,MAXN}};

int n=5; //顶点数

int dist[n+1]={0}; //dist[i]表示当前从源到顶点i的最短特殊路径长度

int prev[n+1]={0}; //前一个节点

dijkstra(n,0, c, dist, prev);

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

if(dist[i]==0 || dist[i]==MAXN){

cout<<"顶点0无法到达顶点"<<i<<endl;

}

else

cout<<"顶点0到顶点"<<i<<"的最短路径长度为"<<dist[i]<<endl;

}

}

（2）分支限界法

#include<iostream>

#include"MinHeap.h"

#define MAXN 1000

using namespace std;

//单源最短路径问题 分支限界法求解

template<class Type>

class Graph{

friend int main();

public:

void ShortesPaths(int);

private:

int n, //图G的顶点数

\*prev; //前驱顶点数组

Type \*\*c, //图G的领接矩阵

\*dist; //最短距离数组

};

//节点属性

template<class Type>

class MinHeapNode{

friend Graph<Type>;

public:

operator int ()const{return length;}

private:

int i; //顶点编号

Type length; //当前路长（优先级）

};

template<class Type>

void Graph<Type>::ShortesPaths(int v){ //单源最短路径问题的优先队列式分支限界法

//定义最小堆容量为1000

MinHeap<MinHeapNode<Type> > H(1000);

//定义源为初始扩展节点

MinHeapNode<Type> E;

E.i=v;

E.length=0;

dist[v]=0;

while (true){//搜索问题的解空间

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

//若顶点i到顶点j可达，且满足控制约束

if((c[E.i][j]!=0)&&(E.length+c[E.i][j]<dist[j])) {

dist[j]=E.length+c[E.i][j];

prev[j]=E.i;

// 加入活结点优先队列

MinHeapNode<Type> N;

N.i=j;

N.length=dist[j];

H.Insert(N);

}

try {

H.DeleteMin(E); // 取下一扩展结点

}

catch (int) {

break;

}

if (H.currentSize==0){// 优先队列空

break;

}

}

}

int main(){

int n=5;

int prev[n+1] = {0};

int dist[n+1]={MAXN,MAXN,MAXN,MAXN,MAXN};

//边权

int a[][MAXN]={{MAXN,MAXN,20,5,MAXN},

{MAXN,MAXN,30,20,MAXN},{MAXN,MAXN,MAXN,30,MAXN},

{MAXN,MAXN,MAXN,MAXN,10},{MAXN,MAXN,MAXN,MAXN,MAXN}};

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

cout<<"单源图的邻接矩阵如下："<<endl;

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

c[i]=new int[n+1];

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

{

c[i][j]=a[i][j];

cout<<c[i][j]<<" ";

}

cout<<endl;

}

//从顶点0开始

int v=0;

Graph<int> G;

G.n=n;

G.c=c;

G.dist=dist;

G.prev=prev;

G.ShortesPaths(v);

cout<<"从顶点0到顶点4的最短路长是："<<dist[4]<<endl;

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

cout<<"prev("<<i<<")="<<prev[i]<<" "<<endl;

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

if(dist[i]==0 || dist[i]==MAXN){

cout<<"顶点0无法到达顶点"<<i<<endl;

}

else

cout<<"顶点0到顶点"<<i<<"的最短路径长度为"<<dist[i]<<endl;

}

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

delete []c[i];

delete []c;

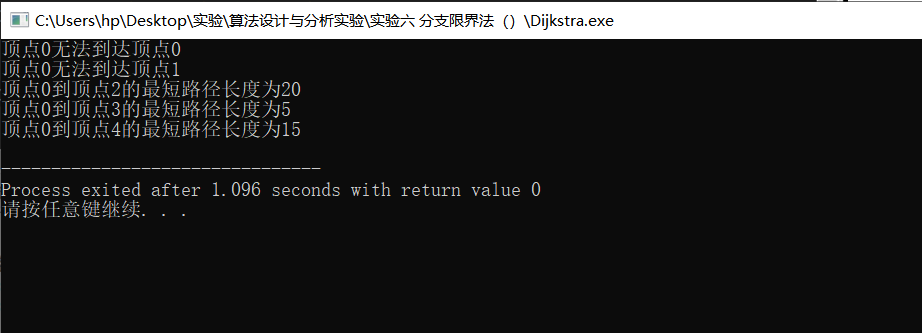
c=0;

return 0;

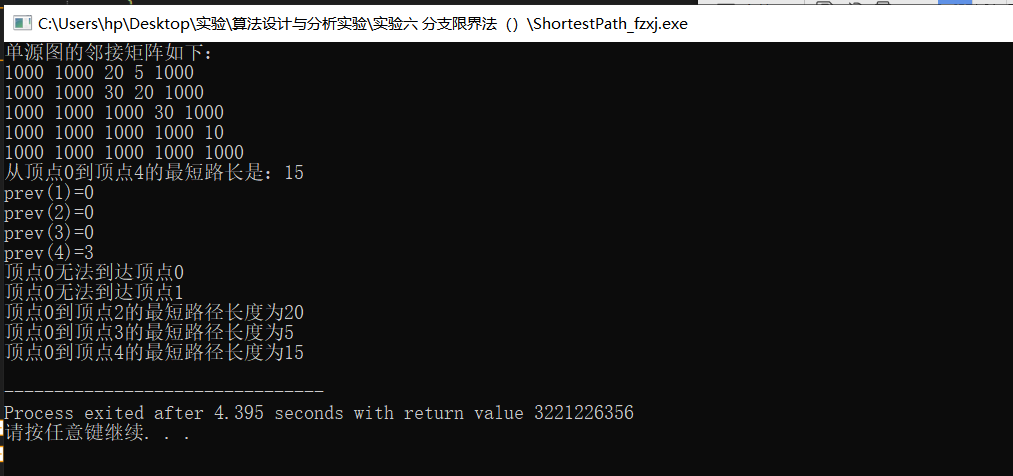
}

**3、输出结果**

（1）贪心算法



（2）分支限界法



**4、性能分析**

（1）贪心算法

**Dijkstra算法**是解单源最短路径问题的贪心算法，主循环需要选取最小的dist[i]和根据找到的结点更新dist[i]，这个循环共要执行n-1次，所以完成循环需要O(n)的时间，该算法的时间复杂度是 O（n^2）。

（2）分支限界法

分支限界法解单元路径问题采用优先队列，用最小堆来存储活结点表，优先级是节点对应的当前路长。在算法扩展结点的过程中，一旦发现一个结点的下界不小于当前找到的最短路长，则算法剪去以该结点为根的子树。

其解空间为子集树，时间复杂度为 O(n!)，空间复杂度为 O(n^2)。

**5、实验体会**

（1）贪心算法

本题中的贪心采用的是Dijkstra算法，其基本思想是：设置顶点集合S并不断地做贪心选择来扩充集合。

初始时，集合S中仅含有源一个元素。设curr是G的某个顶点，把从源到curr且中间只经过集合S中顶点的路称之为从源到顶点curr的特殊路径，并且使用数组 dist 记录当前每个顶点所对应的最短路径的长度。Dijkstra算法每次从图G中的(V-S)的集合中选取具有最短路径的顶点加入到集合S中，同时对数组 dis t进行更新，直到S包含了所有的V中元素。

（2）分支限界法

分支限界法是采用广度优先 BFS 的策略优先产生状态空间树的结点，并使用剪枝函数的方法称为分枝限界法。从而提高搜索效率。按照广度优先的原则，一个活结点一旦成为扩展结点后，将依次生成它的全部孩子结点，将那些导致不可行解或导致非最优解的儿子舍弃，其余儿子加入活结点表中。然后，从活结点表中取出一个结点作为当前扩展结点。重复上述结点扩展过程，直至找到问题的解或判定无解为止。

在本题中，从源顶点s出发到达图G的各个顶点，可以在搜索二叉树时将路长长于当前路长的路径所对应的树中的结点为根的子树剪去，实现剪枝。

**问题三**

**1、问题描述**

二分搜索问题：

对于给定的有序整数集a={6，9，13，15，25，33，45}，编写程序查找18与33，记录每次比较基准数。

**2、代码实现**

#include<iostream>

#include<string>

using namespace std;

/\*二分搜索 (要先排好序)\*/

int BinarySearch(int array[],int x,int n){

int left,right,mid,count=1;

left=0,right=n-1;

while(left<=right){

mid=(left+right)/2;

cout<<" 第"<<count++<<"次比较的基准数为："<<array[mid]<<endl;

if (array[mid]==x)

return mid;

else if(array[mid]>x)

right=mid-1;

else

left=mid+1;

}

return -1; //未找到

}

int main(){

int array[]={6,9,13,15,25,33,45};

int size=sizeof(array)/sizeof(array[0]);

int x; //要查找的数

cout<<"原数组为：";

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

cout<<array[i]<<" ";

cout<<endl;

while(true){

cout<<"请输入要查找的数 x = ";

cin>>x;

if(BinarySearch(array,x,size)==-1){

cout<<"未找到数"<<x<<endl;

}

else

cout<<"找到数"<<x<<"的下标为："<<BinarySearch(array,x,size)<<endl;

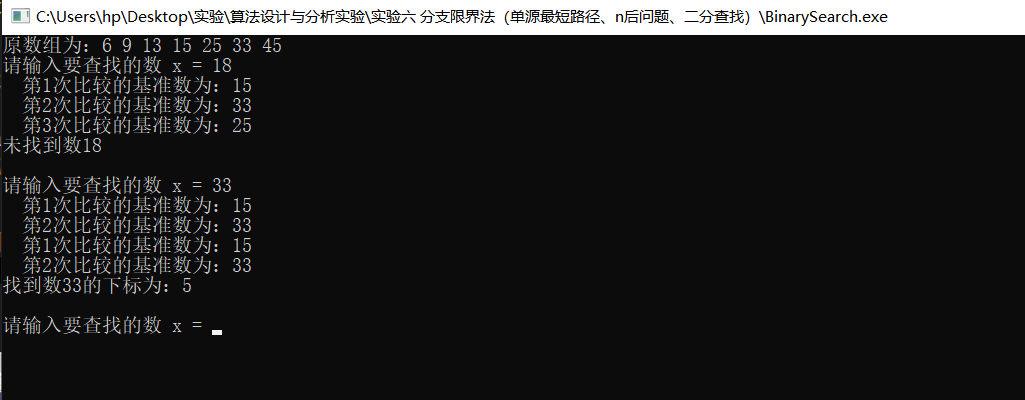
cout<<endl;

}

return 0;

}

**3、输出结果**



**4、性能分析**

二分查找即折半查找，时间复杂度为 O(logn)。

**5、实验体会**

1.首先查找数组必须是有序的。

2.取查找数组中间的数作为基准,如果需要查找的数据大于基准说明该数存在于 数组的左边。反之存在于基准右边。

3 假设待查找的数小于基准,那么将基准换成左子数组的中间的数，重复步骤2，直到找到该数。

**问题四**

**1、问题描述**

对快速排序进行随机化改造，比较不同输入下改造前后的性能。

**2、代码实现**

（1）快速排序

#include<iostream>

#include<algorithm>

using namespace std;

/\*快速排序\*/

//打印数组

void print(int \*a,int left,int right){

int i;

for(i=left;i<=right;i++)

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

cout<<endl<<endl;

}

//取首位元素作为划分基准

int Partition(int \*a,int left,int right){

int i=left,j=right+1;

int x=a[left]; //基准

while(true){

while(a[++i]<x && i<right);

while(a[--j]>x); //两个都为false时跳出

if(i>=j) break;

swap(a[i],a[j]);

}

a[left]=a[j];

a[j]=x;

return j; //j左侧的数都是比a[j]小的数，右侧都是比a[j]大的数

}

int num=1;

//快速排序

void QuickSort(int \*a,int left,int right){

if(left<right){

int q=Partition(a,left,right);

QuickSort(a,left,q-1); //对基准左侧排序

QuickSort(a,q+1,right); //对基准右侧排序

cout<<" 第"<<num++<<"次快速排序的基准为："<<a[q]<<endl;

}

}

int main(){

int a[]={3,2,5,7,8,9,5,0,1,3,4};

int n=sizeof(a)/sizeof(a[0])-1; //n=10

cout<<" 原数组为：";

print(a,0,n);

QuickSort(a,0,n);

cout<<endl;

cout<<" 快速排序后的数组为：";

print(a,0,n);

return 0;

}

（2）随机化快速排序

#include <iostream>

#include<algorithm>

using namespace std;

/\*随机化快速排序\*/

//打印数组

void print(int \*a,int left,int right){

int i;

for(i=left;i<=right;i++)

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

cout<<endl<<endl;

}

//快速排序

int Partition(int \*a,int left,int right);

int RandomPartition(int \*a,int left,int right);

int num=1;

void RandomQuickSort(int \*a,int left,int right){

if(left<right){

int q=RandomPartition(a,left,right);

RandomQuickSort(a,left,q-1); //对左半段排序

RandomQuickSort(a,q+1,right); //对右半段排序

cout<<" 第"<<num++<<"次快速排序的基准为："<<a[q]<<endl;

}

}

int RandomPartition(int \*a,int left,int right){ //产生随机划分

int q=left+rand()%(right-left);

swap(a[q],a[left]);

return Partition(a,left,right);

}

int Partition(int \*a,int left,int right){

int i=left,j=right+1;

int x=a[left]; //基准

while(true){

while(a[++i]<x && i<right);

while(a[--j]>x); //两个都为false跳出循环

if(i>=j) break;

swap(a[i],a[j]);

}

a[left]=a[j];

a[j]=x;

return j;

}

int main(int argc, char\*\* argv) {

int a[]={3,2,5,7,8,9,5,0,1,3,4};

int n=sizeof(a)/sizeof(a[0])-1; //n=10

cout<<" 原数组为：";

print(a,0,n);

RandomQuickSort(a,0,n);

cout<<endl;

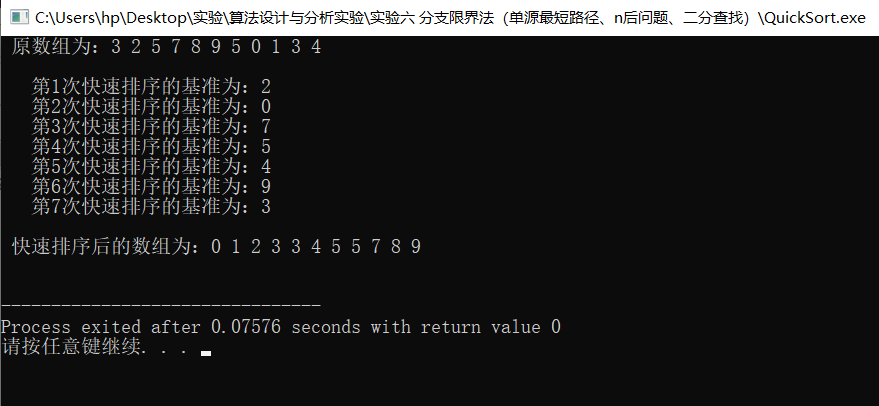
cout<<" 随机化快速排序后的数组为：";

print(a,0,n);

return 0;

}

**3、输出结果**





**4、性能分析**

Partition函数用来确定基准元素以对子数组进行划分，该函数要遍历一遍数组，因此Partition函数的计算时间为O(n)。

综上，快速排序算法的时间复杂度：

最坏情况：函数Partition的每一步都出现不对称划分，T(n)=T(n-1)+O(n)，复杂度为 O(n^2) 。

最好情况：每次选取的基准平分整个序列，T(n)=2T(n/2)+O(n),时间复杂度为O(nlog2n)。

平均情况：时间复杂度为O(nlog2n)。

**5、实验体会**

快速排序也是基于分治策略的一种排序算法。它的原理是选择一个基准元素，通过一趟扫描，将待排序列分成两部分，比基准元素小和大于等于基准元素，再用递归地排序划分的两部分。快速排序每次排序都可能会打乱之前排好序的元素，所以是不稳定的算法。其中，随机化快速排序是以随机一个数为基准（避免了取首位元素直接作为基准的缺点）。