算法实验报告

计算机1604 王殊 1611640413

二分查找

* 题目要求

设数组a，长度N。要求在有序且非降序的一维数组中找到指定的值，如果失败需要返回-1否则返回指定值的位置，返回值介于[0,N)。

* 设计思路

假设查找值b。由于是非降序的顺序，所以可以取得数组的中部mid的值a[mid]，如果b>a[mid]则表明a[0]~a[mid-1]不存在等于b的值。同理，若b<a[mid],a[mid+1]~[N-1]不存在等于b的值。

* 算法描述

function binary\_search(A, n, T):

L := 0

R := n − 1

while L <= R:

m := floor((L + R) / 2)

if A[m] < T:

L := m + 1

else if A[m] > T:

R := m - 1

else:

return m

return unsuccessful

* 效率分析

二分搜索中，总是把原问题的规模降到，所以直到结束，问题的规模为1,2,……，。从递归的角度看，递归的过程可以看作二叉排序树的向下搜索的过程。已知N个二叉搜索树的高度为所以二分搜索的最多的比较次数不会超过二叉搜索树的高度。

* 代码

Main:

int arrays[10] = {1,2,4,5,9,10,11,12,20,30};

int tofind;

std::cout << "请输入查询的数字:";

std::cin >> tofind;

tofind = binary\_sort(arrays, tofind, 0, 9);

std::cout << "位置:" << tofind << (tofind == -1 ? ". 没有找到" : "") << std::endl;

binary\_sort:

int binary\_sort(int a[], int fnd, int l, int h)

{

int mid;

while(l <= h)

{

mid = (l + h) / 2;

if(a[mid] == fnd)

return mid;

else if(a[mid] > fnd)

h = mid -1;

else

l = mid + 1;

}

return -1;

}

* 运行结果



快速排序

* 题目要求

使用快速排序对随机生成的数据进行排序。

* 设计思路

首先要选择一个轴pivot，然后将所有小于pivot的对应的位置i，使得a[0]~a[i-1]≤pivot > a[i+1]~a[N-1]。然后对于子数组a[0]~a[i-1]和a[i+1]~a[N-1]进行相同的操作。

* 算法描述

function partition(a, left, right, pivotIndex)

pivotValue := a[pivotIndex]

swap(a[pivotIndex], a[right]) // 把pivot移到結尾

storeIndex := left

for i from left to right-1

if a[i] < pivotValue

swap(a[storeIndex], a[i])

storeIndex := storeIndex + 1

swap(a[right], a[storeIndex]) // 把pivot移到它最後的地方

return storeIndex

procedure quicksort(a, left, right)

if right > left

select a pivot value a[pivotIndex]

pivotNewIndex := partition(a, left, right, pivotIndex)

quicksort(a, left, pivotNewIndex-1)

quicksort(a, pivotNewIndex+1, right)

* 效率分析

快速排序的子问题的最好的情况是对数组二分，使得子问题的规模为N/2。所以快速排序的划分的计算时间：。如果不能pivot的移动的位置很少，那么

* 代码

template<typename t> void QuickSort(t \*arr, int left, int right){

int i = left;int j = right;t pivot = arr[left];//基准元素

while (i < j){

while (i < j && arr[j] >= pivot) --j;//将j递减到小于基准数值的位置 为什么j先走呢？因为要确定i所能走的上界 防止出现j < i的情况

if (i < j) arr[i] = arr[j];//swap1(arr, i, j);//如果是因为小于基准数值而退出的while循环 就交换i与j所对应的值

while (i < j && arr[i] <= pivot) ++i;//将i递增到第一个大于基准数值的位置

if (i < j) arr[j] = arr[i];//交换位置

//交换之后 pivot对应的位置左边的数字 < pivot 而右边的>pivot

//上两次的交换就是 大于的值 与小于的值得交换

}//优化了多次比较 因为第一次交换pivot交换为j的位置 然后从i选择最大的那个交换需要吧pivot往i的位置交换 所以只需要把arr[j]的位置改为arr[i]就不需要多次赋值pivot

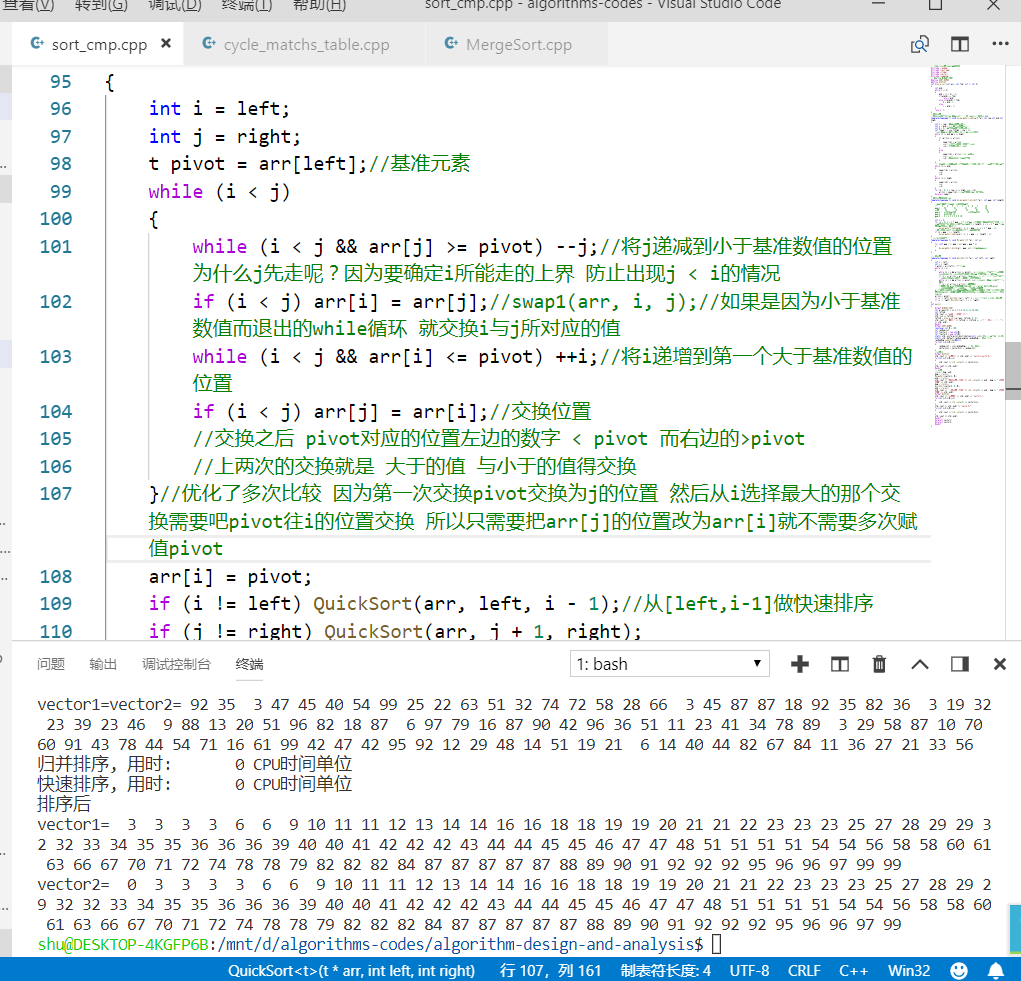
arr[i] = pivot;

if (i != left) QuickSort(arr, left, i - 1);//从[left,i-1]做快速排序

if (j != right) QuickSort(arr, j + 1, right);}

* 运行结果

快速排序使用的原数组vector2。问题规模N=100，随机数产生的范围[1,99]。



归并排序

* 题目要求

归并排序可以将问题的规模减小，然后再对两个小数组进行合并，保持有序。

* 设计思路

首先需要将数组a分开到只剩下至少两个元素（1个元素的情况保留不动）的情况下，然后对于这个两个子数组a[i]和a[j]合并到缓冲区的某个部分（取决于递归的顺序）开始的2个位置。按照从小到大的顺序保存到该部分。然后递归函数返回到主调函数，对更大的部分进行合并。

* 算法描述

这里使用了循环的模式，类似于希尔排序分组的思想（通过控制gap分组）。每次只对相邻gap个元素进行归并。

例：

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 元素 | 5 | 6 | 9 | 1 | 3 | 2 | 0 | 4 | 10 |
| gap=1 | 5 6 | | 9 1 | | 3 2 | | 0 4 | | 10 |
| 合并 | 5 6 | | 1 9 | | 2 3 | | 0 4 | | 10 |
| gap=2 | 1 5 6 9 | | | | 0 2 3 4 | | | | 10 |
| gap=4 | 0 1 2 3 4 5 6 9 | | | | | | | | 10 |
| gap=8 | 0 1 2 3 4 5 6 9 10 | | | | | | | | |

* 效率分析

归并排序的最好需要进行的合并操作，在合并中需要进行不超过N次的比较。所以极端情况下的复杂度为O(Nlog(N))。空间复杂度为O(N)。

* 代码
* 接口函数：

//这个hi接口的函数

template<typename t> void MergeSort(t \*arr, int sz)

{

for (int gap = 1; gap < sz; gap = gap \* 2)

{

do\_mergeSort\_divid(arr, gap, sz);//每一次gap递增

}

}

* 分解函数：

//归并排序的分解算法

template<typename t> void do\_mergeSort\_divid(t \*arr, int gap, int length)

{

//gap表示的是每一次同级比较的间隔

int i = 0;

for (i = 0; i + 2 \* gap - 1/\* i+2\*gap -1表示的是第一组需要合并的 如果只有一对需要合并 那么这个循环执行一次\*/ < length; i = i + 2 \* gap/\*下一个替换的一对的起始位置\*/)

do\_mergeSort\_hebing(arr, i, i + gap - 1, i + 2 \* gap - 1);

//余下了两个子表 因为长度的大小是不可能满足的

if (i + gap - 1 < length)

do\_mergeSort\_hebing(arr, i, i + gap - 1, length - 1);

}

* 合并函数：

//归并排序

//归并算法的合并算法 第一个参数原数组 low为开始的地址 mid

template<typename t> void do\_mergeSort\_hebing(t \*arr,int low,int mid,int high)

{

int i = low;//第一段的序列的下标

int j = mid + 1;//第二段的序列下标

int k = 0;//可临时存放合并序列的下标

int \*newarr = new t[high - low + 1];

//扫描第一段个第二段序列 直到有一个扫描结束

while (i <= mid && j <= high)

{

if (arr[i] <= arr[j])

{

newarr[k] = arr[i];

++i;//有最小值的的数组的小标递增

++k;//临时序列的下标递增

}

else

{

newarr[k] = arr[j];//填入新的值

++j;//同理

++k;//反正k无论如何也要变化

}

}

//注意可能还有没完成的 这里的步骤就是把所有没有写入到新的表中的元素写入

while (i <= mid)

{

newarr[k] = arr[i];

++i;

++k;

}

while (j <= high)

{

newarr[k] = arr[j];

++j;

++k;

}

for (k = 0, i = low; i <= high; ++i, ++k)

arr[i] = newarr[k];//更新对应的元素数组里的值

delete[] newarr;

}

* 运行结果

调整随机数生成的范围[1,99]，问题的规模N=999，得到的归并排序的运行时间。归并排序使用的是数组vector1。



循环日程表

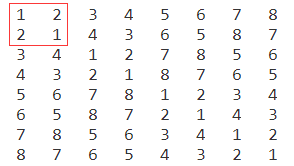
* 题目要求

生成n个选手的比赛日程表。要求：

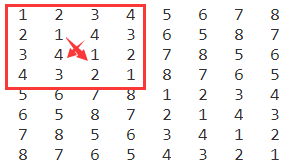
1. 每个选手与其他n-1个选手各赛一次
2. 每个选手一天赛一次
3. 循环赛进行n-1天

* 设计思路

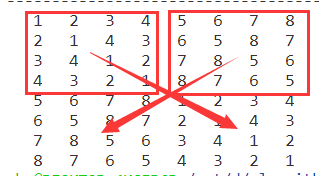
将循环比赛的日程表的问题分解，直到只有上下左右四个元素为止：



对于对角线的元素进行复制。然后考虑将这四个元素视为一个整体，再找到另外上下左右四个“元素”：



进行之前的合并过程。最终可以进行的复制过程。



* 算法描述

1. 计算k=log(n),令s在[1,k]的范围内变化，s控制了执行复制的次数。设m=1，m用来控制每一次复制过程的起始位置。A表示目标的日程表。
2. n的规模减少到n/2，令t在[1,n]的范围内变化，这一步的目的是控制列号的移动的幅度。
3. 令i在[m+1,2m]的位置变化，i的作用是控制表行号，
4. 令j在[m+1,2m]的位置变化，j的作用是控制表列号。
5. A[i][j + (t - 1)\* m \* 2] = A[i - m][j + (t - 1)\*m \* 2 - m]。
6. A[i][j + (t - 1) \* m \* 2 - m] = A[i - m][j + (t - 1)\*m \* 2]。

* 效率分析

最终执行的语句是四层循环中最里层的5、6语句。在对于包含的选手的日程表中，有三层循环，执行的次数是n \* \* 次，也就是。

* 代码

#include <iostream>

#include <iomanip>

using namespace std;

int WIDTH = 4;

int T;

void Table(int n,int k, int \*\*a)

{

// a 是二维数组, k是选手的数量

// 对于n个选手, 总共交换 k=log2(n) 次

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

a[1][i] = i; // 填充第一行

int m = 1; // m 是外循环用来控制

for (int s = 1; s <= k; ++s)

{

n /= 2; // 用于控制交换次数, 代替了取对数的方式.

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

{

for (int i = m + 1; i <= 2 \* m; ++i)// 控制行。\*2是因为每一个兑换过程中，被交换的起始位置是交换的2倍的增长

{

for (int j = m + 1; j <= 2 \* m; ++j) // 控制列

{

a[i][j + (t - 1)\* m \* 2] = a[i - m][j + (t - 1)\*m \* 2 - m];

a[i][j + (t - 1) \* m \* 2 - m] = a[i - m][j + (t - 1)\*m \* 2];

}

}

}

m \*= 2;

}

}

int main()

{

int num\_comper;

int k=0, n=0;

cout << "请输入选手的数量:";

cin >> num\_comper;

n = T = num\_comper;

// 计算k

while (n > 1)

{

++k;

n /= 2;

}

int \*\*table = new int\*[num\_comper + 1];

for (int i = 0; i <= num\_comper; ++i)

{

table[i] = new int[num\_comper + 1];

for(int j=0;j<= num\_comper;++j)

{

table[i][j] = -1;

}

}

Table(num\_comper, k,table);

for (int i = 1; i <= num\_comper; ++i)

{

for (int j = 1; j <= num\_comper; ++j)

{

cout << setw(WIDTH) << table[i][j];

}

cout << endl;

}

for (int i = 0; i <= num\_comper; ++i)

{

delete[] table[i];

}

delete[] table;

}

* 运行结果

