

**实 验 报 告**

（ 2021/2022 学年 第 2 学期）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 课程名称 | 算法分析与设计 | | | | | |
| 实验名称 | 分治策略 | | | | | |
| 实验时间 | 2022 | 年 | 3 | 月 | 28 | 日 |
| 指导单位 | 南京邮电大学计算机学院、软件学院 | | | | | |
| 指导教师 | 陈春玲 | | | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| 学生姓名 | 张宸冉 | 学 号 | B20060527 |
| 学院(系) | 计算机学院 | 专 业 | 计算机科学与技术 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实验名称 | | 分治策略 | | | | 指导教师 | | | 陈春玲 |
| 实验类型 | | 验证 | 实验学时 | | 2 | 实验时间 | | | 2022.3.28 |
| 1. **实验任务和目的**   1、任务：用分治法实现一组无序序列的两路合并排序和快速排序。能将输入的一组无序序列排列为有序序列后输出。  2、目的：理解分治法的算法思想，用分治法解决应用问题的方法和策略，清楚两路合并排序和快速排序算法的基本原理和实施过程。 | | | | | | | | | |
| 1. **实验环境**   硬件：AMD Ryzen 7 4800H with Radeon Graphics  软件：Windows操作系统，Visual C++ 6.0。 | | | | | | | | | |
| **三、核心算法设计及分析**  1、核心算法设计  说明用分治策略（描述分治策略、要素）设计核心算法（包含快速排序和两路合并排序），并说明为什么核心算法可以用分治策略求解；给出核心算法的程序流程图。   1. 合并排序   **分治法理由：**可以把排序问题分解为若干个规模较小的子序列进行排序，并且可以使用相同的方法进行求解，而且子序列之间相互独立。  **当问题足够小时：**当子序列只有1个元素，直接有序，有2个元素，直接进行大小比较就可以得到有序序列。  **分解：**求解出中间元素位置之后，递归调用函数，但是边界需要变化  **求解：**分解到最小规模就只有一个元素，一个元素直接有序，子问题的解就是原问题的解  **合并：**创建一个空数组，大小是需要合并的两个子序列的长度和，比较两个子序列元素的大小，按照从小到大的顺序向数组中添加元素。     1. 快速排序   **分治法理由：**可以把排序问题分解为若干个规模较小的子序列进行排序，并且可以使用相同的方法进行求解，而且子序列之间相互独立  **当问题足够小时：**当分划元素的一边的子序列只有不大于1个的元素，子序列已经有序  **分解：**选择分划元素，分划元素左边都是不大于分划元素的元素，右边都是不小于分划元素的元素。  **求解：**分划元素两边的左右子序列如果规模足够小，就完成了求解，如果不是足够小，就对子序列继续进行分划  **合并：**当左右两个子序列都是已经有序的情况，无需进行额外操作，整个序列已经有序     1. 选择第k小元素   **分治法理由：**二次取中法寻找分划元素符合分治法要求，寻找第k小元素也符合分治法。  **二次取中法**首先把所有序列分成大小一样的若干子序列，把所有子序列的中值合并成一个序列，然后对这个序列再次进行递归调用，寻找中位数。  **寻找第k小元素**：通过二次取中法确定了分划元素之后，对原序列进行分划操作，分划结束之后，如果需要寻找的元素就是分划元素，直接返回索引，如果小于分划元素，那就对分划元素之前的序列搜索第k小元素，如果大于分划元素，那就对分划元素之后的序列搜索第k-(j-left+1)小元素。  **当问题足够小时：**第k小元素直接用插入排序排序完成之后，输出索引号为k-1的元素  **分解：**通过分划元素把序列分解成两个小份，然后在小份里面寻找第k小，或者k-(j-left+1)小，或者就是分划元素本身  **求解：**当搜索的元素就是分划元素本身直接返回索引，不是本身递归调用左右子序列。如果子序列的长度小于每组的元素个数，就直接插入排序，排序后输出索引  **合并：**此问题不存在合并。    2、核心算法分析  快速排序：  时间复杂度： 最好 平均 最坏  空间复杂度： 一般O( 最坏  合并排序：  时间复杂度： 最好 平均 最坏  空间复杂度：  选择第k小元素：  时间复杂度：  空间复杂度： | | | | | | | | | |
| **四、实验及结果分析**  1、实验  测试用例：1,20,15,18,25,63,456,451,22,23  实验结果：  合并排序的结果：1, 15, 18, 20, 22, 23, 25, 63, 451, 456,  快速排序的结果：1, 15, 18, 20, 22, 23, 25, 63, 451, 456,  请输入需要查找的元素序号：3  选择第3小元素的索引是：2  目前的序列是：1, 15, 18, 20, 25, 22, 23, 63, 451, 456,  第3小元素是：18  2、实验结果分析  1）快速排序和合并排序的结果和手工排序一致，都是从小到大非递减顺序。说明在目前的测试用例中，程序运行正确。  单步调试发现，程序运行和算法设计思路一致。  2）选择第k小元素运行结果和手工排序实现效果一致。程序首先对前5个元素按照插入排序进行排序，并且把5个元素的中值和第一个元素进行了交换。之后对后面5个元素进行了同样的操作，并且和整个序列的第二个元素进行了交换。算法的第一个步骤完成。之后对前两个元素进行了对自身的递归调用，发现序列足够小直接进行了插入排序，将第1小元素做完二重取中的中间值。成为整个序列的分划元素，进行分划操作，发现该元素正好是需要寻找的元素，返回索引。如果需要寻找的元素索引小于目前的分划元素，那就对分划元素前的所有元素进行递归调用。符合分治法的思路，依旧是寻找第k小元素。如果需要寻找的元素索引大于目前的分划元素，那就对分划元素后面的元素进行递归调用。但是这个时候就是寻找第k-(j-left+1)小元素，符合分治法思路。 | | | | | | | | | |
| **五、实验小结（包括问题和解决办法、心得体会、建议和意见等）**  分治法求解问题的前提是，问题可以分解为规模较小，相互独立，类型相同的子问题，当问题规模足够小的时候可以直接求解，并且可以把对于子问题的求解合并成对于原问题的求解。  分治法因为会处理大量类型相同的子问题，会大量调用递归算法，会造成一定程度上的时间和空间的损失，但是因为调用了递归，在算法设计上较为简单，程序比较优雅  思考题（3）：  对于分划元素，如果选择随机分划元素，有可能会导致出现最坏情况。 | | | | | | | | | | |
| **六、指导教师评语** | | | | | | | | | | |
| 成 绩 |  | | 批阅人 |  | | | 日 期 |  | | |

**附件：**

#include <iostream>

using namespace std;

template<class K, class D>

struct E

{

//定义可排序表类每一个元素的结构

//重载类型转换运算符

operator K()const {

return key;

}

K key;

D data;

};

template<class T>

class SortableList

{

//定义可排序表类

private:

T\* l; //一个指针 用于动态生成一维数组

int maxSize;

const int INF = 2147483647; //哨兵元素

int n;

/\*合并排序\*/

void Merge(int left, int mid, int right) {

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

int i = left, j = mid + 1, k = 0;

while ((i <= mid) && (j <= right)) {

if (l[i] <= l[j]) {

temp[k++] = l[i++];

}

else {

temp[k++] = l[j++];

}

}

while (i <= mid) {

temp[k++] = l[i++];

}

while (j <= right) {

temp[k++] = l[j++];

}

for (i = 0, k = left; k <= right;) {

l[k++] = temp[i++]; //输出

}

}

void MergeSort(int left, int right) {

if (left < right) {

int mid = (left + right) / 2;

MergeSort(left, mid);

MergeSort(mid + 1, right);

Merge(left, mid, right);

}

}

/\*快速排序\*/

void Swap(int i, int j) {

int c = l[i];

l[i] = l[j];

l[j] = c;

}

int Partition(int left, int right) {

int i = left, j = right + 1;

do {

do{

i++;

}while (l[i] < l[left]);

do {

j--;

}while (l[j] > l[left]);

if (i < j){

Swap(i, j);

}

}while (i < j);

Swap(left, j);

return j;

}

void QuickSort(int left, int right) {

if (left < right){

int j = Partition(left, right);

QuickSort(left, j - 1);

QuickSort(j + 1, right);

}

}

int RandomizedPartition(int left, int right)

{

srand((unsigned)time(NULL)); //用当前时间作为种子

int i = rand() % (right - left) + left; //产生一个left~right范围内的随机数

Swap(i, left);

return Partition(left, right); //调用Partition函数

}

void QuickSortRandom(int left, int right) {

if (left < right) {

int j = RandomizedPartition(left, right); //调用RandomizedPartition函数

QuickSort(left, j - 1);

QuickSort(j + 1, right);

}

}

/\*选择第k小元素\*/

void InsertSort(int left ,int right) {

int insertItem; //每一趟待插入的元素

int j;

for (int i = 1; i <right-left+1; i++) {

insertItem = l[left+i]; //从待排序序列的第二个元素开始，作为待插入元素 因为第一个元素自身一个的话就是有序的

for (j = left+i - 1; j >= left; j--) { //在有序序列中 从后往前，找到需要插入的位置，移动数组 让待插入元素插入

if (insertItem < l[j]) {

l[j + 1] = l[j];

}

else

{

break; //不出现逆序说明已经找到位置 退出移动数组这个模块

}

}

l[j + 1] = insertItem; //在正确位置插入待插入元素

}

}

int Select(int k, int left, int right, int r) {

//每个分组r个元素，寻找第k小元素

int n = right - left + 1;

if (n <= r) {

//若问题足够小，使用直接插入排序 特殊情况直接处理

InsertSort(left, right);

return left + k - 1; //取其中的第k小元素，其下标为left+k-1

}

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

//二次取中规则求每组的中间值

InsertSort(left + (i - 1) \* r, left + i \* r - 1); //每r个元素一组，对每组的元素进行排序

Swap(left + i - 1, left + (i - 1) \* r + (int)ceil((double)r / 2) - 1); //将每组的中间值交换到子表前部集中存放

}

int j = Select( (int)ceil((double)n / r / 2) , left , left + (n / r) - 1 , r); //求二次中间值，(int)ceil((double)n / r / 2)得到的就是中间的值 返回下标为j

Swap(left, j); //二次中间值为枢纽元，并换至left处

j = Partition(left, right); //对表（子表）进行分划操作

if (k == j - left + 1) {

return j; //返回第k小元素下标

}

else if (k < j - left + 1) {

return Select(k, left, j - 1, r);//在左子表求第k小元素

}

else {

return Select(k - (j - left + 1), j + 1, right, r);

//在右子表求第k-(j-left+1)小元素

}

}

public:

SortableList(int mSize) {

maxSize = mSize;

l = new T[maxSize+1];

n = 0; //当前元素个数

}

~SortableList() {

delete[] l;

}

void Input(int \* a, int num) {

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

l[i] = a[i];

n++;

}

l[n] = INF; //设置哨兵元素

}

void Output() {

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

cout << l[i] << ", ";

}

cout << endl;

}

void OutPutByIndex(int index) {

cout << l[index];

}

void MergeSort() {

MergeSort(0, n - 1);

}

void QuickSort() {

QuickSort(0,n-1);

}

void QuickSortRandom() {

QuickSortRandom(0, n - 1);

}

int Select(int k) {

return Select(k, 0, n - 1, 5);

}

};

int main() {

int N = 10;

int a[10] = {1,20,15,18,25,63,456,451,22,23};

SortableList<int> listForMergeSort(20);

listForMergeSort.Input(a, N);

listForMergeSort.MergeSort();

cout << "合并排序的结果：";

listForMergeSort.Output();

SortableList<int> listForQuickSort(20);

listForQuickSort.Input(a, N);

listForQuickSort.QuickSort();

cout << "快速排序的结果：";

listForQuickSort.Output();

SortableList<int> listForQuickSortRandom(20);

listForQuickSortRandom.Input(a, N);

listForQuickSortRandom.QuickSortRandom();

cout << "快速排序（随机分划）的结果：";

listForQuickSortRandom.Output();

SortableList<int> listForSelect(20);

listForSelect.Input(a, N);

int k;

cout << "请输入需要查找的元素序号：" ;

cin >> k;

cout << "选择第"<<k<<"小元素的索引是：";

cout << listForSelect.Select(k) << endl;

cout << "目前的序列是：";

listForSelect.Output();

cout << "第" << k << "小元素是：";

listForSelect.OutPutByIndex(k-1);

return 0;

}