**实 验 报 告**

|  |  |
| --- | --- |
| **课程名称：** | **数据结构与算法** |
| **学生姓名：** | **林墨馨** |
| **学生学号：** | **201730682485** |
| **学生专业：** | **软件工程** |
| **开课学期：** | **2018-2019学年第1学期** |

**软件学院**

**2018年9月**

**目 录**

[实验一 线性表的基本操作实现和应用 1](#_Toc471221116)

[实验二 二叉树的建立及相关算法的实现 3](#_Toc471221117)

[实验三 基本排序算法的实现和应用 5](#_Toc471221118)

[实验四 自组织线性表以及哈希表的实现和应用 7](#_Toc471221119)

# 基本排序算法的实现和应用

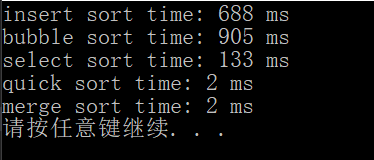
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **地 点：** | | B7 楼B座 | | 房 |  |
| **实验日期与时间：** | | 2018年10月27日 上午第1-4节 | | |
| **实验教师：** | 黄敏、齐海涛 | |

**实验内容：**

a.基本要求

请上机分别实现插入排序、起泡排序、选择排序、快速排序和合并排序算法，并比较这五种算法的时间和空间复杂度，说明相关算法的优势和劣势。

运行结果截图:用随机数生成数据，数据量大小 10000 ，数据范围0-10000



比较:



代码部分

//冒泡排序

template<typename T>

void swap(T & a1, T & a2)

{

T temp = a1;

a1 = a2;

a2 = temp;

}

template<typename T>

void BubbleSort(T \* a, int amount)

{

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

{

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

{

if (a[j] < a[j + 1])

{

swap(a[j], a[j + 1]);

}

}

}

}

//归并排序

template<typename T>

void MergeSort\_r(T \* a, T \* temp, int left, int right) //merge sort using recursion

{

if (right - left == 1)return;

int mid = (right + left) / 2;

MergeSort\_r(a, temp, left, mid);

MergeSort\_r(a, temp, mid, right);

int ii = left, i1 = left, i2 = mid;

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

temp[i] = a[i];

while (i1 < mid && i2 < right)// if want to change the rule of sort ,attention here

{

if (temp[i1] > temp[i2])

{

a[ii++] = temp[i1++];

}

else a[ii++] = temp[i2++];

}

while (i1 < mid)

{

a[ii++] = temp[i1++];

}

while (i2 < right)

{

a[ii++] = temp[i2++];

}

return;

}

#include<stack>

using std::stack;

struct subarray

{

subarray(int l, int r)

{

right = r;

left = l;

ifSort = false;

};

bool ifSort;

int right, left;

};

template <typename T>

void MergeSort\_nr(T \* a, int amount)//use stack to implement merge sort

{

T \* temp = new T[amount];

stack<subarray>s;

subarray current(0, amount);

s.push(current);

while (!s.empty())

{

current = s.top();

if (current.right - current.left == 1)

{

s.pop();

continue;

}

int mid = (current.left + current.right) / 2;

if (current.ifSort == false)//make sure we solve it when meet it twice

{

s.pop();

current.ifSort = true;

s.push(current);

s.push(subarray(current.left, mid));

s.push(subarray(mid, current.right));

continue;

}

//merge sort part

int ii = current.left, i1 = current.left, i2 = mid;

for (int i = current.left; i < current.right; i++)

temp[i] = a[i];

while (i1 < mid && i2 < current.right)

{

if (temp[i1] > temp[i2])// if want to change the rule of sort ,attention here

{

a[ii++] = temp[i1++];

}

else a[ii++] = temp[i2++];

}

while (i1 < mid)

{

a[ii++] = temp[i1++];

}

while (i2 < current.right)

{

a[ii++] = temp[i2++];

}

s.pop();

}

}

template<typename T>

void MergeSort(T \* a, int amount)

{

int \* temp = new T[amount];

MergeSort\_r(a, temp, 0, amount);

delete[] temp;

}

//插入排序

template<typename T>

void InsertSort(T \* a, int amount)

{

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

{

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

{

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

}

}

}

//选择排序

template<typename T>

void SelectSort(T a[], int n) //选择排序

{

T mix, temp;

for (int i = 0; i<n - 1; i++) //每次循环数组，找出最小的元素，放在前面，前面的即为排序好的

{

mix = i; //假设最小元素的下标

for (int j = i + 1; j<n; j++) //将上面假设的最小元素与数组比较，交换出最小的元素的下标

if (a[j]<a[mix])

mix = j;

//若数组中真的有比假设的元素还小，就交换

if (i != mix)

{

temp = a[i];

a[i] = a[mix];

a[mix] = temp;

}

}

return;

}

//快速排序

template<typename T>

class QS//make quicksort into a class,the only surface is QuickSort function

{

public:

static void QuickSort(T \* a, int amount)

{

qsort(a, 0, amount - 1);

}

private:

static int findpivot(T \* a, int left, int right)

{

return (left + right) / 2;

}

static void swap\_(T \* a, int i, int j)

{

T tmp = a[i];

a[i] = a[j];

a[j] = tmp;

}

static int partition(T \* a, int left, int right, const T & pivot)

{

do

{

while (a[++left] < pivot);

while ((left < right) && pivot < a[--right]);

swap\_(a, left, right);

} while (left < right);

return left;

}

static void qsort(T \* a, int left, int right)

{

if (right <= left)return;

int pivot = findpivot(a, left, right);

swap\_(a, pivot, right);//swap pivot element and the last element

int k = partition(a, left - 1, right, a[right]);// becasue use ++l so give i - 1 instead of i

swap\_(a, k, right);//put the pivot in place!

qsort(a, left, k - 1);//now pivot index is k!!

qsort(a, k + 1, right);

}

};

#include<iostream>

#include<stdlib.h>

#include <time.h>

const int rand\_max = 10000;//数据范围 0-10000

const int maxn = 10000;//10000个数据

using namespace std;

int main()

{

int a[5][maxn];

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

{

a[0][i] = rand() % rand\_max;

a[1][i] = a[0][i];

a[2][i] = a[0][i];

a[3][i] = a[0][i];

a[4][i] = a[0][i];

}

clock\_t start, finish;

double totaltime;

start = clock();

InsertSort(a[0], maxn);

finish = clock();

totaltime = finish - start;

cout << "insert sort time: " << totaltime << " ms\n";

start = clock();

BubbleSort(a[1], maxn);

finish = clock();

totaltime = finish - start;

cout << "bubble sort time: " << totaltime << " ms\n";

start = clock();

SelectSort(a[2], maxn);

finish = clock();

totaltime = finish - start;

cout << "select sort time: " << totaltime << " ms\n";

start = clock();

QS<int>::QuickSort(a[3], maxn);

finish = clock();

totaltime = finish - start;

cout << "quick sort time: " << totaltime << " ms\n";

start = clock();

MergeSort(a[4], maxn);

finish = clock();

totaltime = finish - start;

cout << "merge sort time: " << totaltime << " ms\n";

system("pause");

return 0;

}

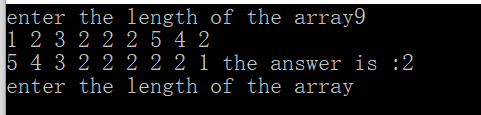
b.具体应用

本部分需要完成两题，第一题和第二题均是任选一题实现。

第一题，从下面两题中，任选一题实现：

1. 数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。例如，输入一个长度为9的数组{1，2，3，2，2，2，5，4，2}。由于数字2在数组中出现了5次，超过数组长度的一半，因此输出2。

实验结果截图



思路:首先使用归并排序给数组排序，然后再创建一个数组记录元素出现的次数，如果出现的次数超过数组长度的一半，就结束循环立即输出（注意 这里的前提是输入的数据要是有结果的 ，也就是有一个数字出现的次数超过数组长度的一半）

代码:

template<typename T>

void MergeSort\_r(T \* a, T \* temp, int left, int right) //merge sort using recursion

{

if (right - left == 1)return;

int mid = (right + left) / 2;

MergeSort\_r(a, temp, left, mid);

MergeSort\_r(a, temp, mid, right);

int ii = left, i1 = left, i2 = mid;

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

temp[i] = a[i];

while (i1 < mid && i2 < right)// if want to change the rule of sort ,attention here

{

if (temp[i1] > temp[i2])

{

a[ii++] = temp[i1++];

}

else a[ii++] = temp[i2++];

}

while (i1 < mid)

{

a[ii++] = temp[i1++];

}

while (i2 < right)

{

a[ii++] = temp[i2++];

}

return;

}

template<typename T>

void MergeSort(T \* a, int amount)

{

int \* temp = new T[amount];

MergeSort\_r(a, temp, 0, amount);

delete[] temp;

}

template<typename T>

T solve(T \* a,int amount)

{

MergeSort(a, amount);

int time = 1;

for (int i = 0; i < amount; i++)cout << a[i] << " ";

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

{

if (a[i] == a[i + 1])

{

time++;

if (time > (amount / 2))//如果找到了 就直接返回 直接跳出循环了

{

return a[i];

}

}

else time = 1;

}

}

//attention: must make sure that there are at least one answer!!!

#include<iostream>

using namespace std;

int main()

{

int \*a;

int length;

while (true)

{

cout << "enter the length of the array";

cin >> length;

a = new int[length];

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

cin >> a[i];

cout << "the answer is :"<<solve<int>(a, length) << endl;

delete[] a;

}

}

②　输入n个整数，找出其中最小的k个数。例如输入4、5、1、6、2、7、3、8这8个数字，则最小的4个数字是1、2、3、4。

第二题，从下面两题中，任选一题实现：

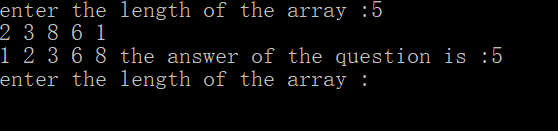
①　我们把只包含因子2、3和5的数称作丑数（Ugly Number）。求按从小到大的顺序的第1500个丑数。例如，6、8都是丑数，但14不是，因为它包含因子7。习惯上我们把1当做第一个丑数。

②　在数组中的两个数字如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组，求出这个数组中的逆序对的总数。

思路：利用归并排序的合并过程来计算逆序对：在合并的过程中，如果左边的值大于右边的值，就产生了逆序对。因为左边数组是有序的，所以，这个时候产生的逆序对的个数是剩余未处理的部分的长度，把这个数据统计起来，就是我们要的逆序对的个数

Tips:还有树状数组的版本

程序运行截图:



代码部分：

template<typename T>

void MergeSort\_r(T \* a, T \* temp, int left, int right, int & ans) //merge sort using recursion

{

if (right - left == 1)return;//如果只有一个元素 则直接返回

int mid = (right + left) / 2;

MergeSort\_r(a, temp, left, mid, ans);//递归的方式 处理左边

MergeSort\_r(a, temp, mid, right, ans);//处理右边

int ii = left, i1 = left, i2 = mid;

for (int i = left; i < right; i++)//copy

temp[i] = a[i];

while (i1 < mid && i2 < right)

{

//比较两边子数组的

if (temp[i1] <= temp[i2])// if want to change the rule of sort ,attention here

{

a[ii++] = temp[i1++];

//ans += mid - i1 + 1;

}

else

{

a[ii++] = temp[i2++];

ans += mid - i1;//利用线性代数学到的

//time++;

}

}

//下面的两个循环只会进入一个

while (i1 < mid)

{

a[ii++] = temp[i1++];

}

while (i2 < right)

{

a[ii++] = temp[i2++];

}

return;

}

template<typename T>

void MergeSort(T \* a, int amount, int & ans)//使用归并排序 只需要调用这个函数

{

ans = 0;

int \* temp = new T[amount];

MergeSort\_r(a, temp, 0, amount, ans);

delete[] temp;

}

#include<iostream>

using namespace std;

int main()

{

int len;

int \* a = nullptr;

while (true)

{

cout << "enter the length of the array :";

cin >> len;

a = new int[len];

int ans;

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

cin >> a[i];

MergeSort<int>(a, len, ans);

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

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

cout << "the answer of the question is :" << ans;

cout << endl;

//cout << ans;

delete[] a;

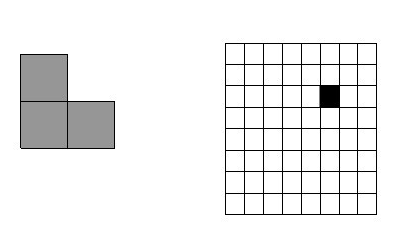
}

}

c.附加题

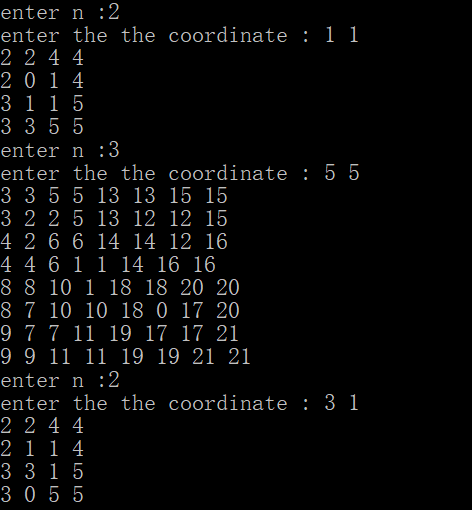
本部分需要完成以下两题：

①Tromino（更准确地说是“右Trominio”）是一个由棋盘上的三个1x1方块组成的L型骨牌。我们的问题是，如何用Tromino覆盖一个缺少了一个方块（可以在棋盘上的任何位置）的2^n\*2^n棋盘。除了这个缺失的方块，Tromino应该覆盖棋盘上的所有方块，Tromino可以任意转向但不能有重叠。为此问题设计一个分治算法。（PS：具体可以参考下图）



程序运行截图：首先输入棋盘的大小（输入题目中的n）,再输入缺块的坐标位置

输出，得到骨牌的摆放方式(用不同的数字表示不同的骨牌)



(三个例子)

思路:对于一个2^n\*2^n 的tromino，我们采用分治的方法，也就是要考虑怎么将大的问题转化成一个个小的子问题，那么很容易就能想到划分的方法:首先把它划分成四个象限，这样就会得到三个完整的子方块和一个缺了一块的子方块，这个时候放一个骨牌（可以在纸上画一下，是一定可以将三个完整的方块转换成tromino的），将三个完整的子方块转换为缺了一块的子方块，也就是我们现在得到了四个缺块的子方块,这个时候我们就将一个大的tromino转换成了四个较小的tromino，这也就是一开始提到的分治的方法。

上面提到的思路是对tromino的一次划分，当然一次划分不一定能解决问题，那么这个时候我们就需要对子tromino继续执行上述的步骤

直到什么时候停止划分呢：当这个tromino无法再划分了，也就是这是1\*1的小方块的时候停止划分

代码:

#include<iostream>

#include<cmath>

using std::endl;

using std::cout;

using std::cin;

const int maxn = 1000;

int ans[maxn][maxn];//棋盘数组

int number = 0;//牌的号码

//sx(start x) sy(start y)代表子棋盘的开始坐标(也就是最左上角的坐标)

//posx,posy 代表缺块的坐标

//size 棋盘坐标

void tromino(int sx,int sy,int posx,int posy,int size)

{

if (size == 1)return;//1\*1的子棋盘 且在之前已经被覆盖了

size /= 2;

int c = number++;

//划分为四个等大的子棋盘

//可以看作 1 2 3 4象限

//象限1

if (posx <= sx + size - 1 && posy <= sy + size - 1)//缺块在第一象限

{

tromino(sx, sy, posx, posy, size);

}

else//不在第一象限 则添加骨牌 构造成缺块的子棋盘

{

ans[sx + size - 1][sy + size - 1] = c;

tromino(sx, sy, sx + size - 1, sy + size - 1, size);

}

//象限2

if (posx > sx + size - 1 && posy <= sy + size - 1)

{

tromino(sx + size, sy,posx, posy,size);

}

else

{

ans[sx + size][sy + size - 1] = c;//左下角被牌覆盖

tromino(sx + size, sy, sx + size, sy + size - 1, size);

}

if(posx <= sx + size - 1 && posy > sy + size - 1)//象限3

{

tromino(sx, sy + size, posx, posy, size);

}

else

{

ans[sx + size - 1][sy + size] = c;

tromino(sx,sy + size , sx + size - 1, sy + size, size);

}

if(posx > sx + size - 1 && posy > sy + size - 1)

{

tromino(sx + size, sy + size, posx, posy, size);

}

else

{

ans[sx + size][sy + size] = c;

tromino(sx + size, sy + size, sx + size, sy + size, size);

}

}

void solve(int size, int x, int y)

{

ans[x][y] = 0;

number = 1;//initialize

tromino(0, 0, x, y, size);

}

int main()

{

int a;

int x, y;

while (true)

{

cout << "enter n :";

cin >> a;

int size = pow(2, a);

cout << "enter the the coordinate : ";

cin >> x >> y;

if (x >= size || y >= size)

{

cout << "wrong input,try again\n";

continue;

}

if (x < 0 || y < 0)

{

cout << "wrong input,try again\n";

continue;

}

solve(size, x, y);

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

{

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

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

cout << endl;

}

}

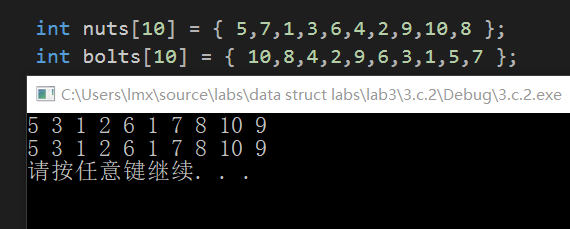
}

1. 假设我们有n个直径各不相同的螺钉以及n个相应的螺母。我们一次只能比较一对螺钉和螺母，来判断螺母是大于螺钉、小于螺钉还是正好适合螺钉。然而，我们不能拿两个螺母做比较，也不能拿两个螺钉做比较。我们的问题是要找到每一对匹配的螺钉和螺母。为该问题设计一个算法，它的平均效率必须属于集合O(nlogn)。

螺母nuts 螺钉bolts

思路：不能拿两个螺母做比较，也不能拿两个螺钉做比较，那么可以用类似快速排序的方法，快速排序的pivot选的是待排序数组中的值。在这里，对于nuts数组，我们选择一个bolts数组中的元素作为pivots将nuts数组分为两个子数组。对于bolts数组，我们选择一个nuts数组钟的元素作为pivots将bolts数组分为两个子数组，然后再递归的重复这个过程，最终就会哭得到排序好的两个数组了

运行截图:上面两个是测试用例，下面是排序得到的结果



代码部分:

#include<iostream>

using std::endl;

using std::cout;

using std::cin;

void swap(int & a, int & b)

{

int tmp = a;

a = b;

b = tmp;

}

void Fix(int \*n, int \*b, int left, int right)

{

if (left >= right)return;

int temp = n[left];//取一个nuts 将bolts划分成两个子数组

int i = left, j = right;

while (i < j)//类似快速排序的过程，不同的是pivot不是bolts数组中的 而是nuts中的一个元素

{

while (i < j && b[i] < temp)

{

i++;

}

while (i < j && b[j] > temp)

{

j--;

}

if (i < j)

{

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

}

}

b[i] = temp;

swap(b[left], b[i]);

//一趟下来，i=j的tmp的位置。以tmp为界限，左右分别是小于和大于它的元素

//再用bolts中的一个元素将nuts数组划分为两个子数组

temp = b[left + 1];

i = left + 1, j = right;

while (i < j)

{

while (i < j && n[i] < temp)

{

i++;

}

while (i < j && n[j] > temp)

{

j--;

}

if (i < j)

{

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

}

}

n[i] = temp;

swap(n[left + 1], n[i]);

//分为两个数组之后，递归的调用函数

Fix(n, b, left + 2, i);

Fix(n, b, i + 1, right);

}

void print(int \* a, int size)

{

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

{

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

}

cout << endl;

}

int main()

{

int nuts[10] = { 5,7,1,3,6,4,2,9,10,8 };

int bolts[10] = { 10,8,4,2,9,6,3,1,5,7 };

int size = 8;

Fix(nuts, bolts, 0, 9);

print(nuts, 10);

print(bolts, 10);

system("pause");

return 0;

//sort(nuts, bolts, 0,size - 1);

}