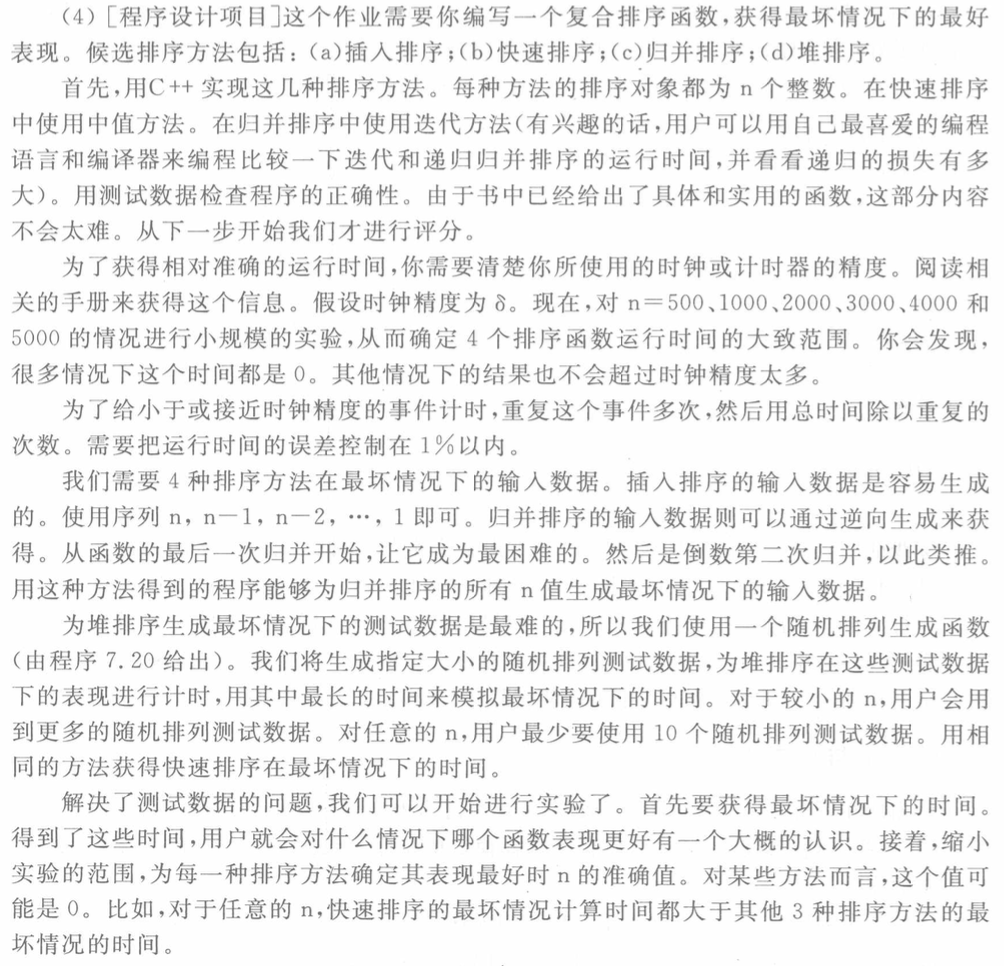
**《数据结构》实验报告（第七章）**

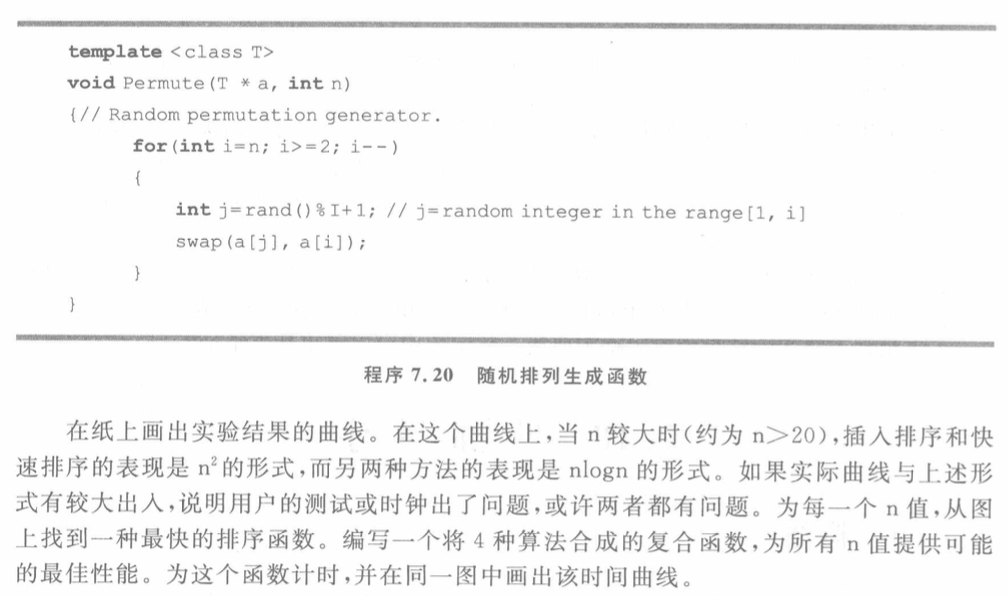
**学号：\_\_\_\_58122231\_\_\_\_\_\_ 姓名 ：\_\_\_陆文韬\_\_\_\_\_**

**实验题号： P435 4\_ 实验日期：\_2023.12.26\_\_\_\_**

**实验一**

**1．问题描述：**描述实验内容和要求以及需要解决的问题。



****

**2．算法思想：**详细描述解决相应问题所需要的算法设计思想。

插入排序，快速排序，归并排序和堆排序根据书上的代码书写完成，算法思想在此不多赘述。这里主要阐述一下四个排序算法对应的最坏情况的数组生成函数的算法思想。插入排序的最坏情况是生成一个逆序的数组，快速排序因为是以最后一个元素为基准，所以最坏情况是生成一个已排序的数组，归并排序将数组进行递归划分，使得两个子数组两两都需要交换，堆排序没有明显的最坏情况，因此采用生成随机数组的方式。

**3．功能函数：**描述所设计的功能函数。如果有多个函数，需要描述它们之间的关系。

generateWorstCaseForInsertionSort：生成插入排序的最坏情况

generateWorstCaseForQuickSort: 生成快速排序的最坏情况

generateWorstCaseForMergeSort: 生成归并排序的最坏情况

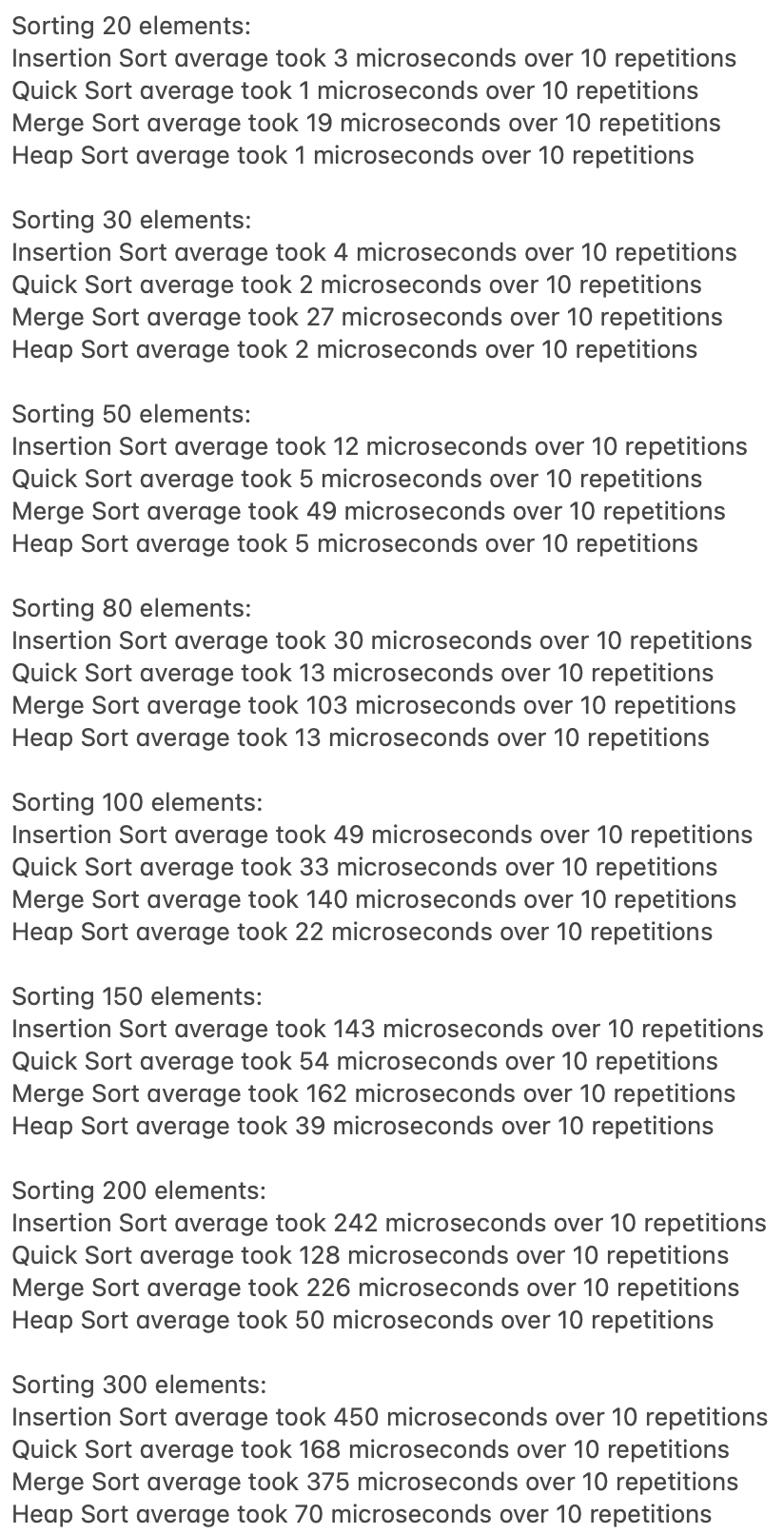
generateWorstCaseForHeapSort: 生成堆排序的最坏情况

**4．测试数据：**设计测试数据，或具体给出测试数据。

提示：要求测试数据能全面地测试所设计程序的功能。

vector<size\_t> sizes = { 20, 30, 50, 80, 100, 150, 200, 300, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7000 };

**5．测试情况：**给出程序的测试情况，分析运行结果，显示实验结果截图。

****

**6．实验总结：**写出实验过程中遇到的问题，以及问题的解决过程。分析算法的时间复杂度和空间复杂度，总结实验心得体会。

最坏情况下，插入排序和快速排序的时间复杂度为O(n\*n)，归并排序和堆排序的时间复杂度为O(nlogn)，在排序的元素数量增加的时候，所需要的时间会快速增加，但是归并排序和堆排序的速度相对来说比较稳定。

在数量相对较少时，插入排序、快速排序和堆排序速度接近，归并排序所需要的时间明显大于其他几个方法。

而当数据量增大的时候，插入排序的速度开始慢慢低于快速排序和堆排序，堆排序的效率明显高于其他三种排序方法。

堆排序综合来说是四种排序中稳定性最好的排序方法，在数据量小和大的时候都有不俗的表现。

**7. 源代码：**给出项目所有源程序清单。

#include<iostream>

#include<vector>

#include<algorithm>

#include<chrono>

#include<random>

#include<functional>

#include<numeric>

**using** **namespace** std;

**using** **namespace** std::chrono;

**void** insertionSort(vector<**int**>& vec) {

**for** (size\_t i = 1; i < vec.size(); i++) {

**int** key = vec[i];

size\_t j = i;

**while** (j > 0 && vec[j-1] > key) {

vec[j] = vec[j-1];

j = j - 1;

}

vec[j] = key;

}

}

**int** partition(std::vector<**int**>& vec, **int** low, **int** high) {

**int** pivot = vec[high]; // Taking the last element as pivot

**int** i = (low - 1); // Index of smaller element

**for** (**int** j = low; j <= high - 1; j++) {

// If current element is smaller than or equal to pivot

**if** (vec[j] <= pivot) {

i++; // increment index of smaller element

std::swap(vec[i], vec[j]);

}

}

std::swap(vec[i + 1], vec[high]);

**return** (i + 1);

}

**void** quickSort(vector<**int**>& arr, **int** left, **int** right)

{

**if** (left < right)

{

**int** i = left, j = right + 1, pivot = arr[left];

**do**

{

**do** i++; **while** (arr[i] < pivot);

**do** j--; **while** (arr[j] > pivot);

**if** (i < j) swap(arr[i], arr[j]);

} **while** (i < j);

swap(arr[left], arr[j]);

quickSort(arr, left, j - 1);

quickSort(arr, j + 1, right);

}

}

**void** merge(std::vector<**int**>& vec, **int** l, **int** m, **int** r) {

**int** n1 = m - l + 1; // Size of the first half

**int** n2 = r - m; // Size of the second half

// Create temporary vectors to hold the halves

std::vector<**int**> L(n1), R(n2);

// Copy data to temp vectors L[] and R[]

**for** (**int** i = 0; i < n1; i++)

L[i] = vec[l + i];

**for** (**int** j = 0; j < n2; j++)

R[j] = vec[m + 1 + j];

// Merge the temp vectors back into vec[l..r]

**int** i = 0; // Initial index of the first subvector

**int** j = 0; // Initial index of the second subvector

**int** k = l; // Initial index of the merged subvector

**while** (i < n1 && j < n2) {

**if** (L[i] <= R[j]) {

vec[k] = L[i];

i++;

} **else** {

vec[k] = R[j];

j++;

}

k++;

}

// Copy the remaining elements of L[], if there are any

**while** (i < n1) {

vec[k] = L[i];

i++;

k++;

}

// Copy the remaining elements of R[], if there are any

**while** (j < n2) {

vec[k] = R[j];

j++;

k++;

}

}

**void** mergeSort(vector<**int**>& vec)

{

**int** n = **int**(vec.size());

**for** (**int** curr\_size = 1; curr\_size <= n - 1; curr\_size \*= 2)

{

**for** (**int** left\_start = 0; left\_start < n - 1; left\_start += 2 \* curr\_size)

{

**int** mid = min(left\_start + curr\_size - 1, n - 1);

**int** right\_end = min(left\_start + 2 \* curr\_size - 1, n - 1);

merge(vec, left\_start, mid, right\_end);

}

}

}

**void** heapify(std::vector<**int**>& vec, **int** n, **int** i) {

**int** largest = i; // Initialize largest as root

**int** left = 2 \* i + 1; // left child

**int** right = 2 \* i + 2; // right child

// If left child is larger than root

**if** (left < n && vec[left] > vec[largest])

largest = left;

// If right child is larger than largest so far

**if** (right < n && vec[right] > vec[largest])

largest = right;

// If largest is not root

**if** (largest != i) {

std::swap(vec[i], vec[largest]);

// Recursively heapify the affected sub-tree

heapify(vec, n, largest);

}

}

**void** heapSort(vector<**int**>& vec)

{

**int** n = **int**(vec.size());

**for** (**int** i = n / 2 - 1; i >= 0; i--)

{

heapify(vec, n, i);

}

**for** (**int** i = n - 1; i >= 0; i--)

{

swap(vec[0], vec[i]);

heapify(vec, i, 0);

}

}

**void** printArray(**const** vector<**int**>&arr)

{

**for** (**int** num : arr)

{

cout << num << " ";

}

cout << endl;

}

// 生成具有n个随机整数的函数

vector<**int**> generateRandomArray(size\_t n)

{

vector<**int**> arr(n);

random\_device random;

mt19937 generate(random());

uniform\_int\_distribution<> distribute(1, 10000);

**for** (size\_t i = 0; i < n; ++i)

{

arr[i] = distribute(generate);

}

**return** arr;

}

// 测量并输出排序函数的执行时间

**void** measureSortingTime(**const** function<**void**(vector<**int**>&)>& sortFunction, vector<**int**>& arr, **const** string& sortName, **int** repetitions)

{

**long** **long** totalDuration = 0;

**for** (**int** i = 0; i < repetitions; ++i) {

vector<**int**> arrCopy = arr; // 使用最坏情况的数组副本进行排序

**auto** start = high\_resolution\_clock::now();

sortFunction(arrCopy);

**auto** stop = high\_resolution\_clock::now();

totalDuration += duration\_cast<microseconds>(stop - start).count();

}

**long** **long** averageDuration = totalDuration / repetitions;

cout << sortName << " average took " << averageDuration << " microseconds over " << repetitions << " repetitions" << endl;

}

// 最坏情况生成函数

vector<**int**> generateWorstCaseForInsertionSort(**int** n)

{

vector<**int**> arr(n);

iota(arr.begin(), arr.end(), 1);

reverse(arr.begin(), arr.end()); //对于插入排序，逆序数组是最坏情况

**return** arr;

}

vector<**int**> generateWorstCaseForQuickSort(**int** n)

{

vector<**int**> arr(n);

iota(arr.begin(), arr.end(), 1); //对于快速排序，已排序的数组是最坏情况

**return** arr;

}

//将一个向量平均分成两个子向量，一个包含奇数项，一个包含偶数项

**void** split(vector<**int**>& v, vector<**int**>& odd, vector<**int**>& even)

{

**for** (**int** i = 0; i < v.size(); i++)

{

**if** (i % 2 == 0)

{ // 如果是偶数项

even.push\_back(v[i]); // 放入even向量

}

**else**

{ // 如果是奇数项

odd.push\_back(v[i]); // 放入odd向量

}

}

}

//将一个向量里的所有数据按照归并排序的要求处理并合并到另一个向量中

**void** process(vector<**int**>& v, vector<**int**>& result)

{

**if** (v.size() == 1)

{

result.insert(result.begin(), v[0]);

}

**else** **if** (v.size() == 2)

{ // 如果向量里只有两个数据

swap(v[0], v[1]); // 交换这两个数据

result.insert(result.begin(), v.begin(), v.end()); // 将交换后的向量合并到结果向量中

}

**else**

{ // 如果向量里有多于两个数据

vector<**int**> odd, even; // 定义两个子向量，分别存放奇数项和偶数项

split(v, odd, even); // 将原向量平均分成两个子向量

process(odd, result); // 对奇数项子向量递归处理并合并到结果向量中

process(even, result); // 对偶数项子向量递归处理并合并到结果向量中

}

}

vector<**int**> generateWorstCaseForMergeSort(**int** n)

{

vector<**int**> arr(n);

iota(arr.begin(), arr.end(), 1);

vector<**int**> result;

process(arr, result);

**return** result;

}

// 生成具有n个随机整数的函数

vector<**int**> generateRandomArray(**int** n)

{

vector<**int**> arr(n);

random\_device random;

mt19937 generate(random());

uniform\_int\_distribution<> distribute(1, 10000);

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

{

arr[i] = distribute(generate);

}

**return** arr;

}

vector<**int**> generateWorstCaseForHeapSort(**int** n)

{

// 堆排序的最坏情况不容易造成，通常认为它的性能较为稳定，这里创建一个随机数组

vector<**int**> arr(n);

arr = generateRandomArray(n);

**return** arr;

}

**int** main()

{

vector<size\_t> sizes = { 20, 30, 50, 80, 100, 150, 200, 300, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7000 };

**int** repetitions = 10; // 排序重复的次数

**for** (size\_t k : sizes)

{

cout << "Sorting " << k << " elements:" << endl;

**int** n = **int**(k);

vector<**int**> arr = generateWorstCaseForInsertionSort(n);

measureSortingTime(insertionSort, arr, "Insertion Sort", repetitions);

arr = generateWorstCaseForQuickSort(n);

measureSortingTime([&](vector<**int**>& vec) { quickSort(vec, 0, **int**(vec.size() - 1)); }, arr, "Quick Sort", repetitions);

arr = generateWorstCaseForMergeSort(n);

measureSortingTime(mergeSort, arr, "Merge Sort", repetitions);

arr = generateWorstCaseForHeapSort(n);

measureSortingTime(heapSort, arr, "Heap Sort", repetitions);

cout<<endl;

} **int** a;

cin>>a;

**return** 0;

}