

八种排序算法的比较

八种排序算法的比较

1651573 刘客

编译说明

- 一. 项目介绍
- 二. 设计
- 三. 八种排序算法实现
- 四. 正确性检验
- 五. 算法比较
- 六. Linux下运行效果

1651573 刘客

编译说明

- 在windows平台下的.exe文件
 - 在Linux平台下的.out文件
-

一. 项目介绍

- 背景

排序算法在编写程序时会经常遇到,面对不同的应用情景,高效又稳定的排序算法是必须考虑的。

本题实现了冒泡排序、选择排序、直接插入排序、希尔排序、快速排序、堆排序、归并排序、基数排序八种排序算法,并对其区别进行了分析。

- 功能

使用随机函数产生n个随机数,并选定排序算法进行排序。统计排序时间和交换次数。

二. 设计

- 公用函数设计
 - 函数

函数名称	返回值类型	描述
drawScreen	void	绘制操作台画面函数
main	int	主函数
coutResult	void	输出结果函数
judgeSort	bool	测试函数,判断排序后的结果是否有序

- coutResult

param sortName 排序名称
param sortTimes 排序次数
param sortCost 排序时间开销

```
void coutResult(string sortName, int sortTimes, double sortCost) {  
    cout << sortName << "所用时间:\t" << sortCost << "ms\n";  
    cout << sortName << "交换次数:\t" << sortTimes << "次\n\n";  
}
```

- judgeSort

param numArray 排序后的数组
param size 数组大小

算法
从前向后逐一比较,若出现后一位比前一位小的情况,则说明排序失败

```
...  
bool judgeSort(int *numArray, int size) {  
    for(int i = 0; i < size-1; i++) {  
        if (numArray[i] > numArray[i + 1]) {  
            return false;  
        }  
    }  
    return true;  
}  
...
```

三. 八种排序算法实现

均以从小到大(升序)为例

1. 冒泡排序

算法思想

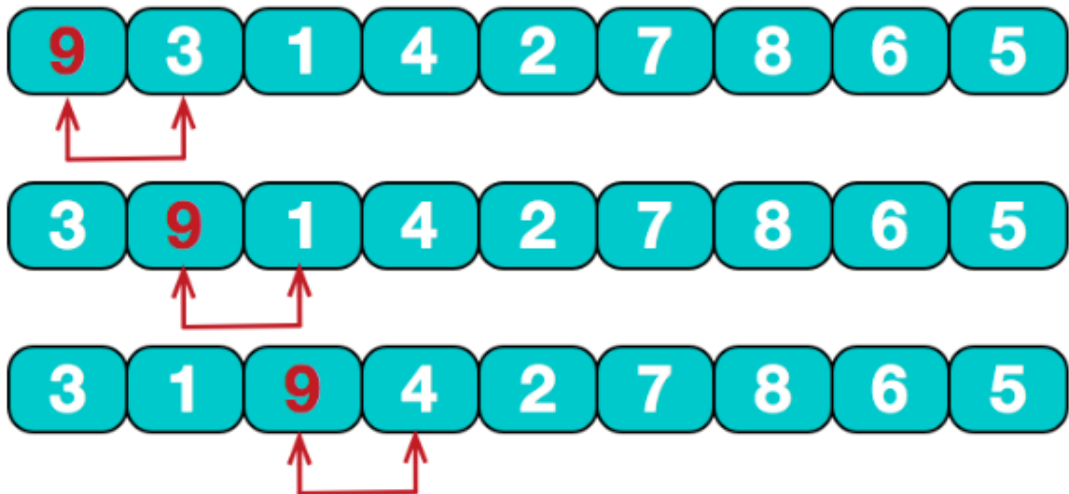
对相邻元素进行两两比较,如果顺序相反则进行交换,这样每一次遍历都会有最小的元素浮到顶端(假设按升序排列),经过n-1次后达到有序
是稳定的排序

优化

当在一次遍历过程中,不存在交换行为,则说明排列已经有序,不必再做后面的遍历了。

图解

相邻元素两两比较，反序则交换



第一轮完毕，将最大元素9浮到数组顶端



同理,第二轮将第二大元素8浮到数组顶端



排序完成



时间控件复杂度分析

冒泡排序的时间复杂度极为不理想。在最坏情况下，即要排序的数字完全是倒序排列，这样我们每执行一次循环，只能将一个数字放在正确的位置上，并且每次比较都需要交换两个数字，这是一笔非常大的开销，它的时间复杂度为 $O(n^2)$ 。即使对于平均情况而言，它的时间复杂度也为 $O(n^2)$ 。

冒泡排序对空间的要求不高，既不需要辅助数组，也不执行递归调用，所以它的空间复杂度为 $O(1)$ 。

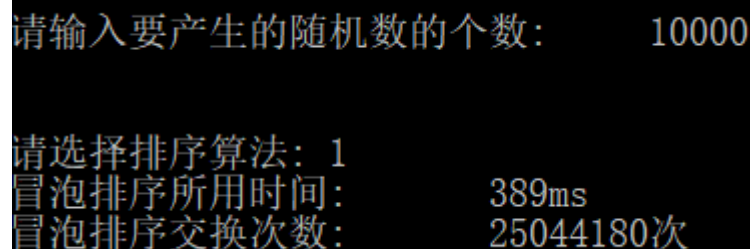
```
//冒泡排序
void bubbleSort(int num) {
    int* numArray = new int[num + 1];
    srand((unsigned)time(NULL));
    for (int i = 0; i < num; i++) {
        numArray[i] = rand();
    }
}
```

```

    }
    clock_t start = clock();
    int swapTime = 0;
    for (int i = 0; i < num; i++) {
        //用来判断是否在本次冒泡过程中存在无交换的情况(即后续未进行冒泡的序列恰好满足排序规律,则
        无需再排)
        int swapPerTime = 0;
        for (int j = num - 1; j > i; j--) {
            if (numArray[j - 1] > numArray[j]) {
                int temp = numArray[j - 1];
                numArray[j - 1] = numArray[j];
                numArray[j] = temp;
                swapTime++;
                swapPerTime++;
            }
        }
        if (swapPerTime == 0) {
            break;
        }
    }
    clock_t end = clock();
    double costTime = (double)(end - start) / CLOCKS_PER_SEC * 1000;
    coutResult("冒泡排序", swapTime, costTime);
    cout << "\n" << judgeSort(numArray, num)<<"\n\n";
    delete[] numArray;
}

```

运行示例



```

请输入要产生的随机数的个数:      10000

请选择排序算法: 1
冒泡排序所用时间:      389ms
冒泡排序交换次数:      25044180次

```

2. 选择排序

算法思想

每一次遍历,从待排序的元素中,选出最小的元素与本次循环的首元素进行交换,进行n-1次,完成排序。
存在远距离交换 是**不稳定**的排序

时间控件复杂度分析

在最坏情况下,时间复杂度为 $O(n^2)$,平均情况下时间复杂度也为 $O(n^2)$ 。
选择排序的空间复杂度为 $O(1)$,不需要辅助数组和递归调用

```

//选择排序
void chooseSort(int num) {
    int * numArray = new int[num + 1];
    srand((int)time(NULL));
    for (int i = 0; i < num; i++) {
        numArray[i] = rand();
    }
}

```

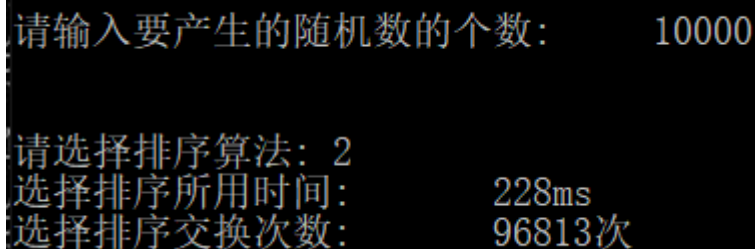
```

    }

    clock_t start = clock();
    int swapTime = 0;
    for (int i = 0; i < num - 1; i++) {
        //将最小值初始化为INT的类型的最大值
        int min = INT_MAX;
        int minIndex = -1;
        for (int j = i; j < num; j++) {
            if (numArray[j] < min) {
                min = numArray[j];
                minIndex = j;
                swapTime++;
            }
        }
        int temp = numArray[i];
        numArray[i] = min;
        numArray[minIndex] = temp;
        swapTime++;
    }
    clock_t end = clock();
    double costTime = (double)(end - start) / CLOCKS_PER_SEC * 1000;
    coutResult("选择排序", swapTime, costTime);
    cout << "\n" << judgeSort(numArray, num) << "\n\n";
    delete numArray;
}

```

运行示例



```

请输入要产生的随机数的个数:      10000

请选择排序算法: 2
选择排序所用时间:      228ms
选择排序交换次数:      96813次

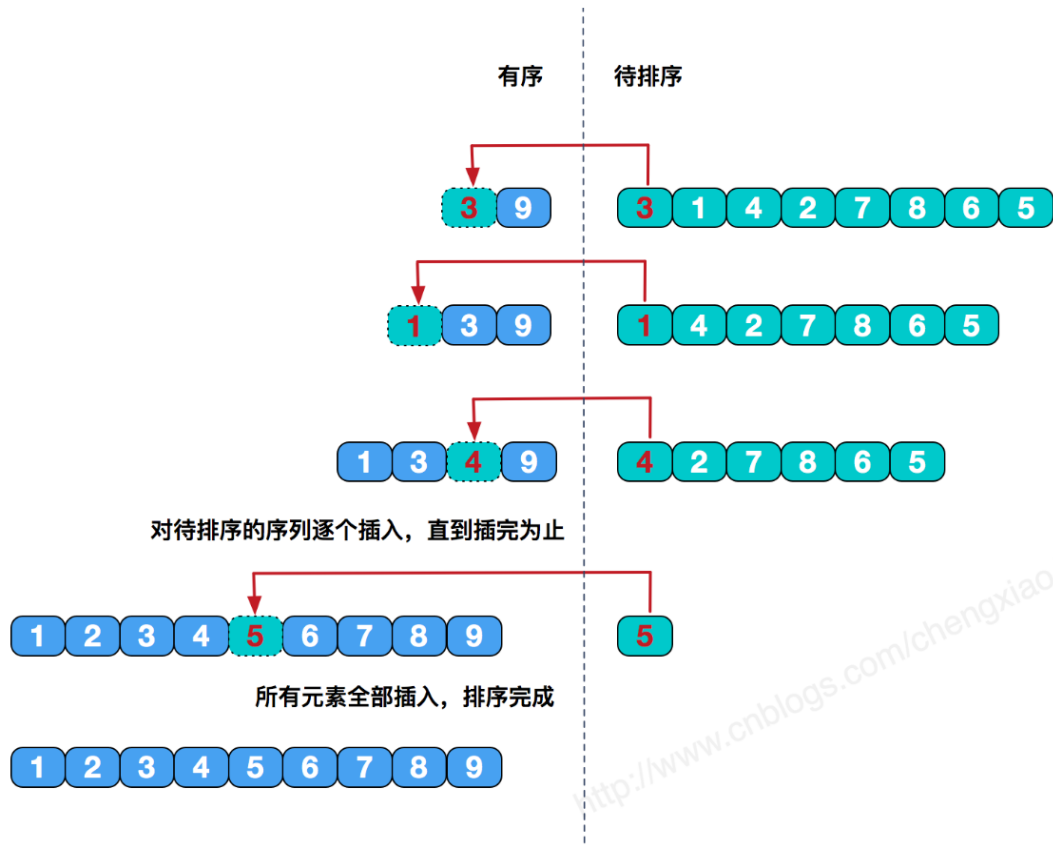
```

3. 直接插入排序

算法思想

每一次遍历,将一个待排序的记录,插入到前面已经排好序的有序序列中,直到插满所有元素为止是**稳定**的排序

图解



时间空间复杂度分析

最坏情况下为 $O(n^2)$ ，平均情况下为 $O(n^2)$ 。

空间复杂度为 $O(1)$

//直接插入排序

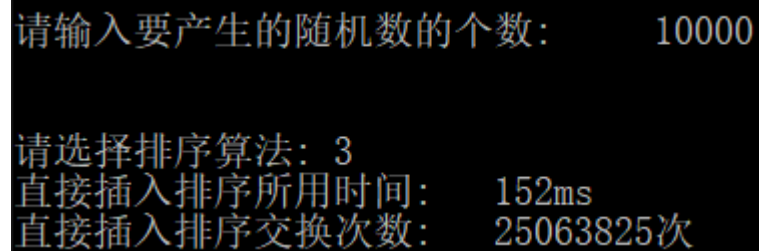
```
void insertSort(int num) {
    int * numArray = new int[num + 1];
    srand((int)time(NULL));
    for (int i = 0; i < num; i++) {
        numArray[i] = rand();
    }

    clock_t start = clock();
    int swapTime = 0;
    for (int i = 1; i < num; i++) {
        int j = i - 1;
        int temp = numArray[i];
        for (; j >= 0; j -= 1) {
            if (numArray[j] > temp) {
                numArray[j + 1] = numArray[j];
                swapTime++;
            }
            else {
                break;
            }
        }
        numArray[j + 1] = temp;
        swapTime++;
    }
}
```

```
}
clock_t end = clock();
double costTime = (double)(end - start) / CLOCKS_PER_SEC * 1000;

coutResult("直接插入排序", swapTime, costTime);
cout << "\n" << judgeSort(numArray, num) << "\n\n";
}
```

运行示例



```
请输入要产生的随机数的个数:    10000

请选择排序算法: 3
直接插入排序所用时间:    152ms
直接插入排序交换次数:    25063825次
```

4. 希尔排序

算法思想

希尔排序是将数组按一定间隔进行分离,将一个待排序序列,分成若干个小的待排序序列,然后分别对每个序列执行插入排序,执行完毕后,缩小间隔,继续进行这一过程,直到间隔为1,此时再进行一次直接插入排序,即可得到排序序列。

显然,对一个较小数据量的序列使用直接插入排序是迅速,而希尔排序通过对序列划分,使得当间隔接近1的时候,序列近乎排序成功,从而有效地降低了时间开销

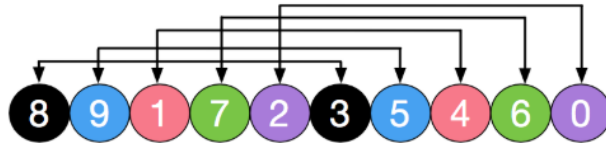
存在远距离交换,是**不稳定**的排序

图解

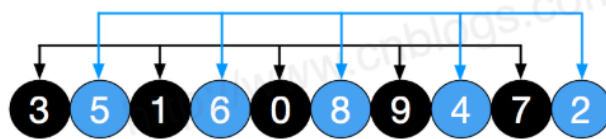
原始数组 以下数据元素颜色相同为一组



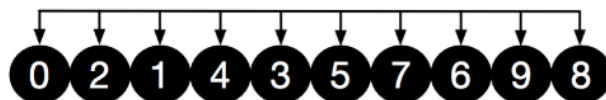
初始增量 $gap=length/2=5$ ，意味着整个数组被分为5组，[8,3] [9,5] [1,4] [7,6] [2,0]



对这5组分别进行直接插入排序，结果如下，可以看到，像3，5，6这些小元素都被调到前面了，然后缩小增量 $gap=5/2=2$ ，数组被分为2组 [3,1,0,9,7] [5,6,8,4,2]



对以上2组再分别进行直接插入排序，结果如下，可以看到，此时整个数组的有序程度更进一步啦。再缩小增量 $gap=2/2=1$ ，此时，整个数组为1组[0,2,1,4,3,5,7,6,9,8]，如下



经过上面的“宏观调控”，整个数组的有序化程度成果喜人。

此时，仅仅需要对以上数列简单微调，无需大量移动操作即可完成整个数组的排序。



时间空间复杂度分析

时间复杂度难以估算。

空间复杂度为 $O(1)$

//希尔排序

```
void shellSort(int num) {  
    int * numArray = new int[num + 1];  
    srand((int)time(NULL));  
    for (int i = 0; i < num; i++) {  
        numArray[i] = rand();  
    }  
}
```

```
clock_t start = clock();  
int swapTime = 0;  
int gap = num / 3 + 1;  
int count = 0;  
while (count == 0) {  
    if (gap == 1) {
```



```

        if (pivotpos != i) {
            int temp = numArray[i];
            numArray[i] = numArray[pivotpos];
            numArray[pivotpos] = temp;
            swapTime++;
        }
    }
    numArray[low] = numArray[pivotpos];
    numArray[pivotpos] = pivot;
    swapTime++;
    return pivotpos;
}
//递归,使用swapTime的引用进行交换次数计数
void quickSort(int *numArray, int low, int high,int& swapTime) {
    if (low < high) {
        int pivotpos = partition(low, high, numArray, swapTime);
        quickSort(numArray, low, pivotpos - 1, swapTime);
        quickSort(numArray, pivotpos + 1, high, swapTime);
    }
}

void useQuickSort(int num) {
    int * numArray = new int[num + 1];
    srand((int)time(NULL));
    for (int i = 0; i < num; i++) {
        numArray[i] = rand();
    }
    int swapTime = 0;
    clock_t start = clock();
    quickSort(numArray, 0, num - 1,swapTime);
    clock_t end = clock();

    double costTime = (double)(end - start) / CLOCKS_PER_SEC * 1000;
    coutResult("快速排序", swapTime, costTime);
    cout << "\n" << judgeSort(numArray, num) << "\n\n";
    delete[] numArray;
}

```

运行示例

请输入要产生的随机数的个数: 10000

请选择排序算法: 5

快速排序所用时间: 2ms

快速排序交换次数: 81133次

6. 堆排序

算法思想

采用最大堆来进行排序,首先通过最大堆来将数据处理,此时堆顶的元素即为最大的值,这时,将堆顶元素与堆的最后一个元素交换,并将堆的size减一,然后通过向下转移函数重新排序,重复 $n-1$ 次则可得到排好序的数组

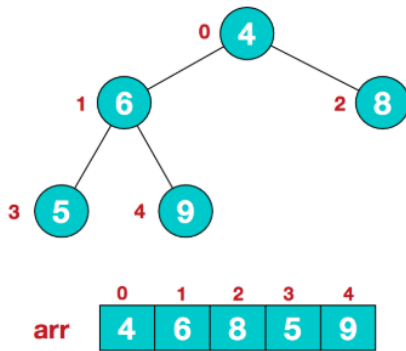
使用最大堆的原因则是可以重复利用空间结构进行存储,减少了空间开销

存在远距离交换,是**不稳定**的排序

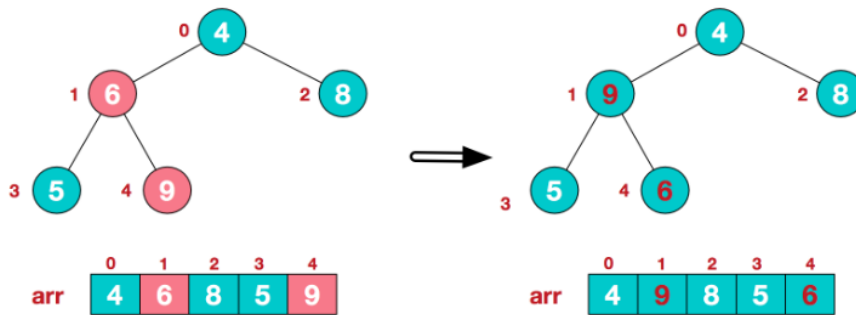
图解

步骤一 构造初始堆。将给定无序序列构成一个大顶堆（一般升序采用大顶堆，降序采用小顶堆）。

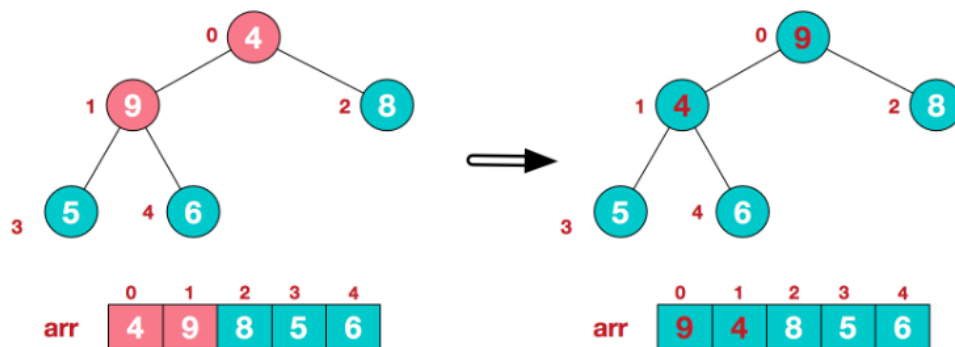
a. 假设给定无序序列结构如下



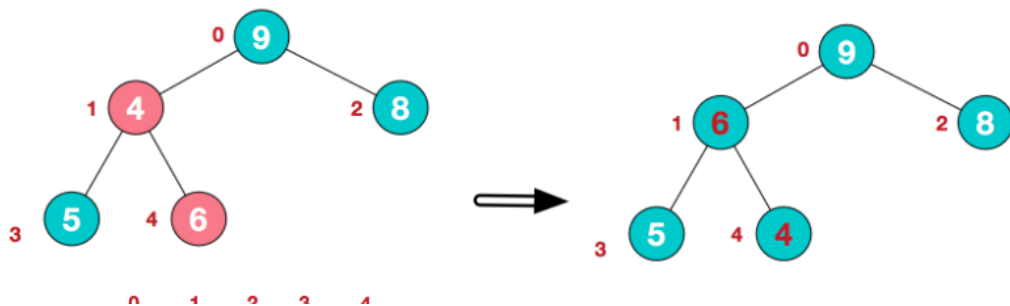
2. 此时我们从最后一个非叶子结点开始（叶结点自然不用调整，第一个非叶子结点 $\text{arr.length}/2-1=5/2-1=1$ ，也就是下面的6结点），从左至右，从下至上进行调整。

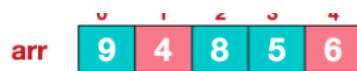


4. 找到第二个非叶节点4，由于[4,9,8]中9元素最大，4和9交换。



这时，交换导致了子根[4,5,6]结构混乱，继续调整，[4,5,6]中6最大，交换4和6。

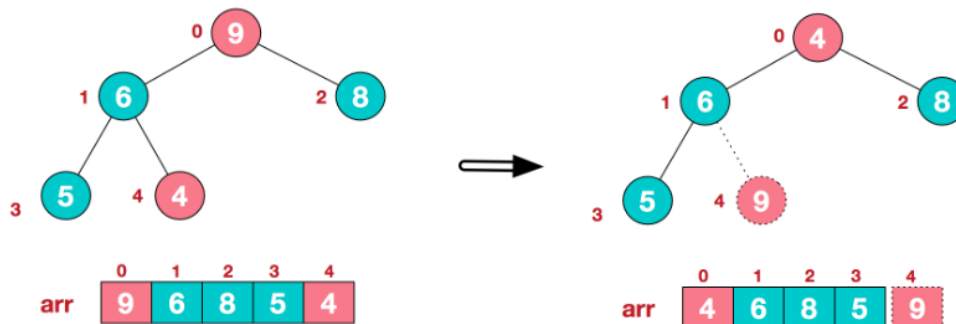




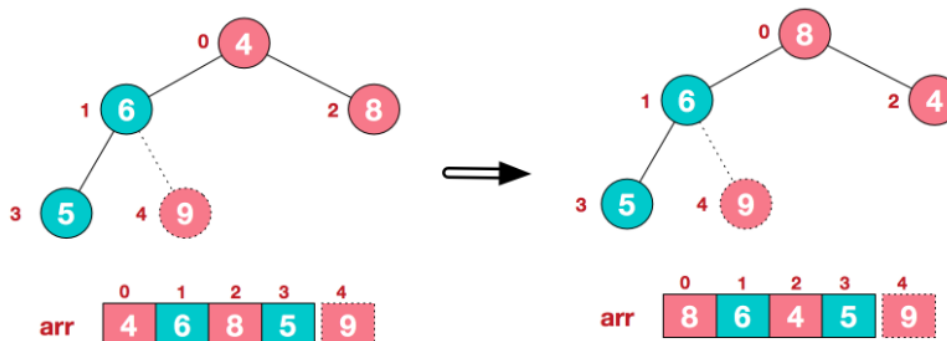
此时，我们就将一个无序序列构造成了一个大顶堆。

步骤二 将堆顶元素与末尾元素进行交换，使末尾元素最大。然后继续调整堆，再将堆顶元素与末尾元素交换，得到第二大元素。如此反复进行交换、重建、交换。

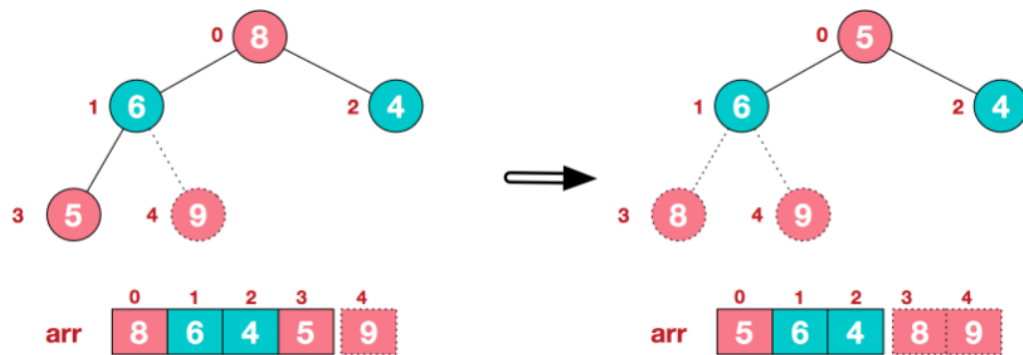
a. 将堆顶元素9和末尾元素4进行交换



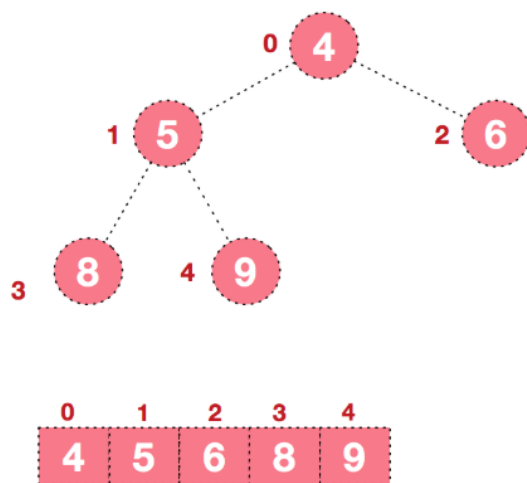
b. 重新调整结构，使其继续满足堆定义



c.再将堆顶元素8与末尾元素5进行交换，得到第二大元素8。



后续过程，继续进行调整，交换，如此反复进行，最终使得整个序列有序



时间空间复杂度分析 堆排序是一种选择排序，它的最坏，最好，平均时间复杂度均为 $O(n\log n)$ 。堆排序不是递归算法，也不需要额外的存储空间（辅助数组），所以其空间复杂度为 $O(1)$ 。

```
void heapSort(int num) {
    MaxHeap myHeap(num);
    clock_t start = clock();
    srand((int)time(NULL));
    for (int i = 0; i < num; i++) {
        myHeap.insert(rand());
    }
    for (int i = num - 1; i >= 1; i--) {
        myHeap.swap(0, i);
        myHeap.shiftDown(i - 1);
    }
    clock_t end = clock();
    double costTime = (double)(end - start) / CLOCKS_PER_SEC * 1000;
    coutResult("堆排序", myHeap.swapTime, costTime);
    cout << "\n" << judgeSort(myHeap.numArray, num) << "\n\n";
}
```

请输入要产生的随机数的个数: 10000

请选择排序算法: 6

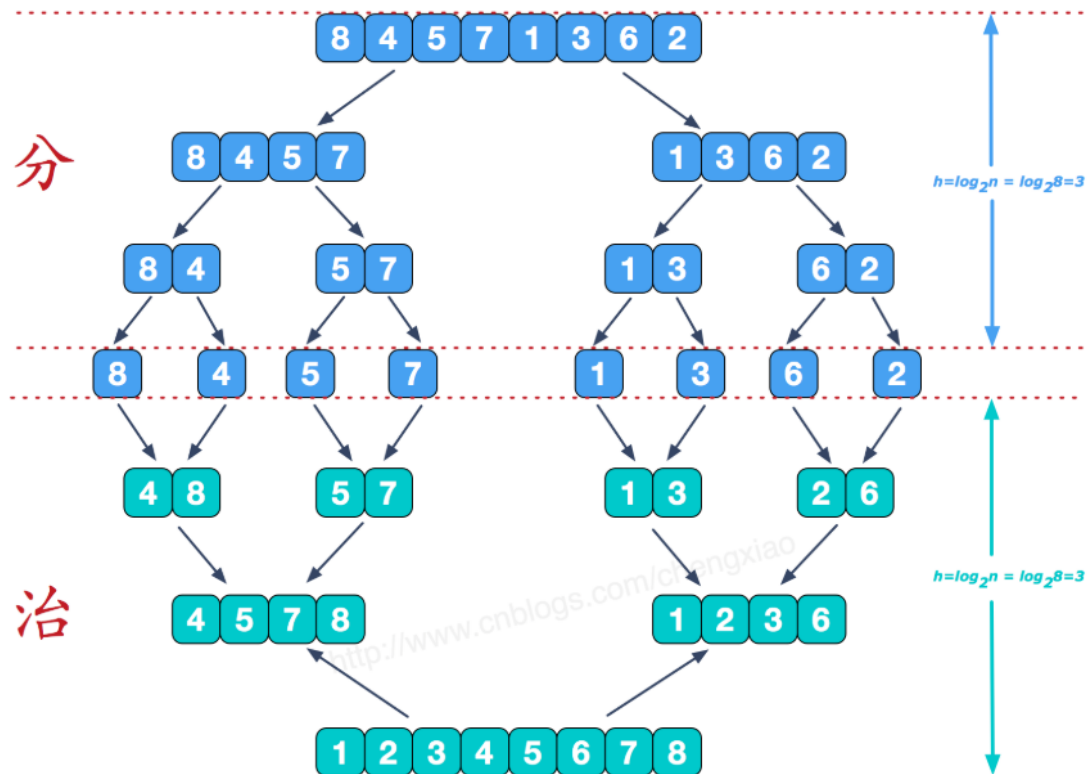
堆排序所用时间: 13ms

堆排序交换次数: 149644次

7. 归并排序

算法思想 归并排序是利用归并的思想实现的排序方法。采用分治策略,将一个序列分成两个子序列,重复上述过程不断递归,直至子序列的长度全部唯一,然后再两两进行合并,最终形成排好序的序列
存在远距离交换,是**不稳定**的排序

图解



时间空间复杂度分析

归并排序将序列进行划分的时间复杂度为 $O(\log n)$,而每次进行合并的时间复杂度为 $O(n)$ 因此时间复杂度为 $O(n \log n)$

辅助数组占据的空间为 $O(n)$,消耗的栈空间为 $O(\log n)$,因此时间复杂度为 $O(n)$

```
void merge(int *arrayOne, int first, int mid, int last, int * temp,int& swapTime)
{
    int left = first;
    int right = mid + 1;
    for (int i = first; i < last; i++) {
        temp[i] = arrayOne[i];
    }
    int count = first;
```

```

//如果左右两个序列都未执行完毕
while (left <= mid && right <= last) {
    if (temp[left] < temp[right]) {
        arrayOne[count++] = temp[left++];
        swapTime++;
    }
    else if (temp[right] < temp[left]) {
        arrayOne[count++] = temp[right++];
        swapTime++;
    }
    else
    {
        //两个数相同情况
        arrayOne[count++] = temp[left++];
        swapTime++;
        arrayOne[count++] = temp[right++];
        swapTime++;
    }
}
//如果左序列未到头, 而右序列已经结束
while (left <= mid) {
    arrayOne[count++] = temp[left++];
    swapTime++;
}
//如果右序列未到头,而左序列已经结束
while (right <= last) {
    arrayOne[count++] = temp[right++];
    swapTime++;
}
}

void mergeSort(int* numArray, int first, int last, int* temp,int& swapTime) {
    if (first < last) {
        int mid = (first + last) / 2;
        mergeSort(numArray, first, mid, temp,swapTime);
        mergeSort(numArray, mid + 1, last, temp,swapTime);
        merge(numArray, first, mid, last, temp,swapTime);
    }
}
}

```

```

void useMergeSort(int num) {
    int * numArray = new int[num + 1];
    int *temp = new int[num];
    srand((int)time(NULL));
    int swapTime = 0;
    for (int i = 0; i < num; i++) {
        numArray[i] = rand();
    }
    clock_t start = clock();
    mergeSort(numArray, 0, num - 1, temp,swapTime);
    clock_t end = clock();
    double costTime = (double)(end - start) / CLOCKS_PER_SEC * 1000;
    coutResult("归并排序", swapTime, costTime);
}

```



```
cout << "\n" << judgeSort(numArray, num) << "\n\n";
}
...
>运行示例

```

8. 基数排序

算法思想

本处实现采用MSD最高位优先的方法,因此需要额外的空间进行存储。

首先确定基数radix,因为是对数进行排序,故本算法基数选定为10

然后通过遍历,从各个数字的最高位开始分析,将最高位相同的数扔到同一块连续的存储区域,记录存储区域的结束地址,和元素的个数(此处使用数组,可以通过 `array[address--]` 将在某位上相同的元素取出)。

使用循环,对被扔到同一块区域的数进行递归,对次一位进行排序。当区域中的数少于一定数量时,可以使用直接插入排序,避免因为稀疏数据而进行遍历所造成的浪费。

本次算法当区域中数小于200时,会选择进行直接插入排序

是稳定的排序

时间空间复杂度分析

基数排序的时间复杂度为 $O(d(n+r))$ 其中 d 为最大元素的位数, r 为基数的个数

基数排序的空间需求在于两个辅助数组, 分别用于储存中间过程的结果和每个基数对应的开始位置, 故空间复杂度为 $O(n+r)$

```
//获得当前数字的最大位数
int getMaxDigit(int num) {
    int n = 1;
    while (static_cast<int> (num / pow(10,n))) {
        n++;
    }
    return n;
}

//获得当前数字的第n位
int getDigit(int num,int n) {
    //先整除 10^n 得到余数,再将余数除以 10^(n-1) 得到首位的数字
    return static_cast<int>((num % static_cast<int>(pow(10, n))) / pow(10, n - 1));
}

void insertSortForMSD(int *numArray, int left, int right, int &swapTime) {
    for (int i = left + 1; i <= right; i++) {
        int j = i - 1;
        int temp = numArray[i];
        for (; j >= 0; j -= 1) {
            if (numArray[j] > temp) {
                numArray[j + 1] = numArray[j];
                swapTime++;
            }
            else {
                break;
            }
        }
        numArray[j + 1] = temp;
    }
}
```

```

        swapTime++;
    }
}

//数的区间为[left,right],右边为闭区间
void MSDsort(int *numArray, int left, int right, int index,int &swapTime) {
    if (index <= 0) {
        //递归终止条件,处理完毕,返回
        return;
    }
    if (right - left < 200) {
        insertSortForMSD(numArray, left, right, swapTime);
        return;
    }
    int i, j,tempLeft,tempRight;
    int count[radix];
    int* tempArray = new int[right - left + 1];

    //初始化记录数位的数组
    for (i = 0; i < radix; i++) {
        count[i] = 0;
    }
    //写入count数组关于在当前位数下,每个数所有的元素的个数
    for (i = left; i <= right; i++) {
        count[getDigit(numArray[i], index)]++;
    }
    //重定向各个区间的位置
    for (j = 1; j < radix; j++) {
        count[j] += count[j - 1];
    }
    //将每个区域的数写入临时数组
    for (i = left; i <= right; i++) {
        j = getDigit(numArray[i], index);
        tempArray[count[j] - 1] = numArray[i];
        swapTime++;
        count[j]--;
    }
    //将临时数组中的数保存到相对应的真正存储的数组区间中
    for (i = left, j = 0; i <= right; i++, j++) {
        numArray[i] = tempArray[j];
        swapTime++;
    }
    for (j = 0; j < radix; j++) {
        tempLeft = left + count[j];
        if (j != radix - 1) {
            tempRight = left + count[j + 1] - 1;
        }
        else {
            tempRight = right;
        }

        MSDsort(numArray, tempLeft, tempRight, index - 1,swapTime);
    }
}

```

```

/* for (int i = tempLeft; i < tempRight; i++) {
    cout << numArray[i] << "\t";
    if ((i + 1) % 5 == 0) {
        cout << endl << endl;
    }
}*/
delete[] tempArray;
}

void useMSDsort(int num) {
    int * numArray = new int[num + 1];
    srand((int)time(NULL));
    for (int i = 0; i < num; i++) {
        numArray[i] = rand();
    }
    int swapTime = 0;
    clock_t start = clock();
    int maxDigit = 0;
    //获取随机生成的数中的最高位数
    for (int i = 0; i < num; i++) {
        int index = getMaxDigit(numArray[i]);
        if (index > maxDigit) {
            maxDigit = index;
        }
    }
    MSDsort(numArray, 0, num - 1, maxDigit, swapTime);
    clock_t end = clock();
    double costTime = (double)(end - start) / CLOCKS_PER_SEC * 1000;
    coutResultt("基数排序", swapTime, costTime);
    cout << "\n" << judgeSort(numArray, num) << "\n\n";
}

```

运行示例

```

请输入要产生的随机数的个数:      10000

请选择排序算法: 8
基数排序所用时间:      19ms
基数排序交换次数:      144060次

```

四. 正确性检验

输出为1时为排序正确

输出为0时为排序错误

输出

请输入要产生的随机数的个数: 10000

请选择排序算法: 1

冒泡排序所用时间: 443ms

冒泡排序交换次数: 24882225次

判断情况: 1

请选择排序算法: 2

选择排序所用时间: 166ms

选择排序交换次数: 97888次

判断情况: 1

请选择排序算法: 3

直接插入排序所用时间: 89ms

直接插入排序交换次数: 25166911次

判断情况: 1

请选择排序算法: 4

希尔排序所用时间: 2ms

希尔排序交换次数: 233156次

判断情况: 1

请选择排序算法: 5

快速排序所用时间: 1ms

快速排序交换次数: 74737次

判断情况: 1

请选择排序算法: 6

堆排序所用时间: 7ms

堆排序交换次数: 149782次

判断情况: 1

请选择排序算法: 7

归并排序所用时间: 1ms

```
冒泡排序所用时间: 1ms
归并排序交换次数: 133616次

判断情况: 1

请选择排序算法: 8
基数排序所用时间: 62ms
基数排序交换次数: 145648次

判断情况: 1
```

五. 算法比较

- 时间空间复杂度

排序方式	平均时间复杂度	最坏时间复杂度	空间复杂度	稳定性
冒泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
直接插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
希尔排序	$O(n^{7/6})$	$O(n^{4/3})$	$O(1)$	不稳定
快速排序	$O(n \log n)$	$O(n^2)$	$O(\log n)$	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(1)$	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n)$	不稳定
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(n+r)$	稳定

- 不同数据量下比较
 - 数据量为1000

请输入要产生的随机数的个数： 1000

请选择排序算法： 1

冒泡排序所用时间： 10ms

冒泡排序交换次数： 253250次

请选择排序算法： 2

选择排序所用时间： 2ms

选择排序交换次数： 7423次

请选择排序算法： 3

直接插入排序所用时间： 2ms

直接插入排序交换次数： 240553次

请选择排序算法： 4

希尔排序所用时间： 0ms

希尔排序交换次数： 14315次

请选择排序算法： 5

快速排序所用时间： 1ms

快速排序交换次数： 5739次

请选择排序算法： 6

堆排序所用时间： 0ms

堆排序交换次数： 11546次

请选择排序算法： 7

归并排序所用时间： 0ms

归并排序交换次数： 9976次

请选择排序算法： 8

基数排序所用时间： 2ms

基数排序交换次数： 13242次

- 数据量为10000

请输入要产生的随机数的个数: 10000

请选择排序算法: 1

冒泡排序所用时间: 339ms

冒泡排序交换次数: 25086124次

请选择排序算法: 2

选择排序所用时间: 166ms

选择排序交换次数: 95318次

请选择排序算法: 3

直接插入排序所用时间: 216ms

直接插入排序交换次数: 25191709次

请选择排序算法: 4

希尔排序所用时间: 3ms

希尔排序交换次数: 234643次

请选择排序算法: 5

快速排序所用时间: 2ms

快速排序交换次数: 79258次

请选择排序算法: 6

堆排序所用时间: 16ms

堆排序交换次数: 149528次

请选择排序算法: 7

归并排序所用时间: 2ms

归并排序交换次数: 133616次

请选择排序算法: 8

基数排序所用时间: 61ms

基数排序交换次数: 145230次

- 数据量为100000

请输入要产生的随机数的个数: 100000

请选择排序算法: 1
冒泡排序所用时间: 37786ms
冒泡排序交换次数: 2500986894次

请选择排序算法: 2
选择排序所用时间: 15595ms
选择排序交换次数: 1149619次

请选择排序算法: 3
直接插入排序所用时间: 8492ms
直接插入排序交换次数: 2510478294次

请选择排序算法: 4
希尔排序所用时间: 28ms
希尔排序交换次数: 3175054次

请选择排序算法: 5
快速排序所用时间: 26ms
快速排序交换次数: 1027154次

请选择排序算法: 6
堆排序所用时间: 43ms
堆排序交换次数: 1827633次

请选择排序算法: 7
归并排序所用时间: 55ms
归并排序交换次数: 1668928次

请选择排序算法: 8
基数排序所用时间: 267ms
基数排序交换次数: 1585037次

六. Linux下运行效果

```

**                               **
排序算法比较
=====
**      1 --- 冒泡排序      **
**      2 --- 选择排序      **
**      3 --- 直接插入排序  **
**      4 --- 希尔排序      **
**      5 --- 快速排序      **
**      6 --- 堆排序        **
**      7 --- 归并排序      **
**      8 --- 基数排序      **
**      9 --- 退出程序      **
=====

请输入要产生的随机数的个数： 10000

请选择排序算法： 2
选择排序所用时间：      220ms
选择排序交换次数：      97106次

请选择排序算法： 1
冒泡排序所用时间：      610ms
冒泡排序交换次数：      24817921次

请选择排序算法： 3
直接插入排序所用时间：  160ms
直接插入排序交换次数：  24709915次
```

```

请选择排序算法： 4
希尔排序所用时间：      0ms
希尔排序交换次数：      239456次

请选择排序算法： 5
快速排序所用时间：      10ms
快速排序交换次数：      76721次

请选择排序算法： 6
堆排序所用时间： 0ms
堆排序交换次数： 149764次

请选择排序算法： 7
归并排序所用时间：      0ms
归并排序交换次数：      133616次

请选择排序算法： 8
基数排序所用时间：      30ms
基数排序交换次数：      185280次

请选择排序算法： 9
```