排序算法复习

1. 总结

1.1 算法思想分析

O(n^2)型排序：冒泡、选择、插入

- 冒泡排序: 逐渐交换冒泡，将较大大的数放到后边，O(n2)
- 选择排序: 逐渐从整个数组上选择最小的元素放到最开始的位置，O(n2)
- 插入排序: 逐渐，将元素插入到前边可放下的最大位置处，O(n2)

O(n*logN)型排序：归并、快速、堆、希尔

- 归并排序:

开始，每个元素为长度 1 的有序区间，相邻区间进行合并，得到长度 2 的有序区间,接下来翻倍增长，直到合并为一个有序区间，O(n*logN)

- 快速排序:

在数组中，随机选择一个数，将小于他的数，放在其左边，大于它的数放在其右边，并以此数为划分，递归进行类似操作，O(n*logN)

划分过程：随意选择一个数，与数组最后一个数交换，设置一个小于等于区间，开始为空，放在整个数组左边；接下来从左到右遍历所有元素，如果当前元素大于划分数，则和划分区间下一个数交换，同时划分区间扩展一位，到最后，将划分数和小于等于区间下一个数交换位置，划分的时间复杂度 O(N)

- 堆排序

首先将整个元素构建成大根队，然后将根脱离处堆，重新调整堆，最后会得到一个有序数组

- 希尔排序：插入排序的改良

希尔的步长是从逐渐减少到 1 的，希尔排序的关键在于步长的选择,在步长为 1 的时候，即成为一个插入排序。

时间复杂读为 O(N)的排序算法:计数排序/基数排序。

这一类算法不是基于比较的排序算法，其思想来自于桶排序。

- 计数排序:

对身高进行排序，在身高范围上划分 160-180 之间 20 个桶，将员工放在对应身高的桶里，最后按桶的大小，依次倒出桶中的数据即可得到排序。

- 基数排序:

一组数字，若是十进制，则划分 0-9 十个桶，依次把所有数字放在其个位对应的桶上，得到初步序列；接着对十位进行桶排序，得到进一步的序列；直到最高位结束，得到有序序列。

1.2 排序算法表格统计

时间复杂度	算法
O(n2)	冒泡、选择、插入
O(n*logN)	归并、快速、堆、希尔
O(N)	计数、基数
空间复杂度	算法
O(1)	插入、选择、冒泡、堆、希尔
O(logN)~O(N)	快排
O(N)	归并排序
O(M)	计数排序、基数怕排序 (M 由桶的数量决定)
稳定性	算法
稳定	冒泡、插入、归并、计数、基数、桶
不稳定	选择、快速、希尔、堆

---

2. 代码部分

插入排序

排序思想

将每个元素逐个插入到前面已经排好序的序列中 引入二分查找位置，从有序序列中查找插入位置

```
//插入排序，关键是找插入位置
void insertionSort(int A[], int n)
{
    for (int i = 1; i < n; i++){        //从第二个元素开始遍历
        int get = A[i];                 //获取第 i 个要排序的元素
        int j = i - 1;                  //前 j 个元素已经排好序

        //将当前的元素插入到有序的前 j 个元素数组中
        while (j >= 0 && A[j] > get)    //从大到小遍历，寻找第一个小于待插元素的位置，边找边移动位置
        {//找到了当前元素应该所在的位置，A[j] > get，表示当前序列是递增排序的，因为要找到第一个小于
        //排序元素的位置就停止循环，该循环是将所有大于 get 的元素后移一位
```

```
            A[j + 1] = A[j];
            j--;            //比较元素位置前移一位，继续查找
        }
        A[j + 1] = get;                 //此时 A[j] <= get ，将get 插入到
 j+1 个位置因为j-- 了，所以用j + 1
    }
}

//二分插入排序
void insertionSortDichotomy(int A[], int n)
{
    for (int i = 1; i < n; i++){
        int get = A[i];                         // 右手抓到一张扑克牌
        int left = 0;                           // 拿在左手上的牌总是排序好的，
所以可以用二分法
        int right = i - 1;                      // 手牌左右边界进行初始化
        while (left <= right){                  // 采用二分法定位新牌的位置
            int mid = (left + right) / 2;       //获得有序序列的中间值索引
            if (A[mid] > get)                   //中间值较大
                right = mid - 1;                //舍去上半序列
            else                                //中间值较小
                left = mid + 1;                 //舍去下半序列
        }
        for (int j = i - 1; j >= left; j--) // 将欲插入新牌位置右边的牌整
体向右移动一个单位
        {
            A[j + 1] = A[j];
        }
        A[left] = get;                          // 将抓到的牌插入手牌
    }
}

void swap(int A[], int i, int j)
{
    int temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}
```

选择排序

排序思想

每次预留一个位置给最值，然后遍历求最值，将最值放到目的位置

```
//选择排序也是一个渐进排序过程，逐个调出最小值（最大值），直至没有未完成 序列
为空
void selectionSort(int A[], int n)
```

```
{
    for (int i = 0; i < n - 1; i++)          //最后一个值肯定是最大值，不
需要再选择
    {
        int min = i;                         //记录当前待排序的元素
        for (int j = i + 1; j < n; j++)      // 子循环，完成找到当前未排序
序列的最小值
        {
            if (A[j] < A[min]){
                min = j;
            }
        }
        if (min != i)                        // 如果当前元素不是最小值，将
当前最小值与待排序位置值互换
        {
            swap(A, min, i);
        }
    }
}
```

冒泡排序

排序思想

冒泡，查找最大值或者最小值，将最值放到最后。逐渐有序。

变体：鸡尾酒排序的排序思想

每次循环找两个最值，最大和最小，放置于两端，直至完成排序。

```
//冒泡排序
void bubbleSort(int A[], int n)
{
    for (int j = 0; j < n - 1; j++)          // 每次最大元素就像气泡一样"
浮"到数组的最后，使序列变得有序
    {
        for (int i = 0; i < n - 1 - j; i++) // 依次比较相邻的两个元素,冒
出最大的元素,
        {
            if (A[i] > A[i + 1])             // 如果条件改成A[i] >= A[i +
 1],则变为不稳定的排序算法
            {
                swap(A, i, i + 1);
            }
        }
    }
}
```

```
//鸡尾酒排序
void cocktailSort(int A[], int n)
{
    int left = 0;                                   // 初始化边界
    int right = n - 1;
    while (left < right)
    {
        for (int i = left; i < right; i++)   // 前半轮,将最大元素放到后面
        {
            if (A[i] > A[i + 1])
            {
                swap(A, i, i + 1);
            }
        }
        right--;
        for (int i = right; i > left; i--)   // 后半轮,将最小元素放到前面
        {
            if (A[i - 1] > A[i])
            {
                swap(A, i - 1, i);
            }
        }
        left++;
    }
}
```

快速排序

排序思想

二分，随机或者按照规则选取一个值，按照值进行二分，将原数组分成一堆大数和一堆小数。递归执行二分。直至完成排序。

```
int partition(int A[], int left, int right)  // 划分函数
{
    int pivot = A[right];                // 这里每次都选择最后一个元素作为基准
    int tail = left - 1;                 // tail 为小于基准的子数组最后一个元素的索引
    for (int i = left; i < right; i++)  // 遍历基准以外的其他元素
    {
        if (A[i] <= pivot)               // 该元素小于等于基准
        {
            swap(A, ++tail, i);          // 该元素放到前一个子数组末尾
        }
    }
    swap(A, tail + 1, right);            // 最后把基准放到前一个子数组的后边，剩下的子数组就是大于基准的子数组
```

```
                                        // 该操作很有可能把后面元素的稳定性打乱，所以快速排序是不稳定的排序算法
    return tail + 1;                    // 返回基准的索引
}

//快速排序
void quickSort(int A[], int left, int right)
{
    if (left >= right)//递归遍历条件
        return;
    int pivot_index = partition(A, left, right); // 获得基准的索引
    quickSort(A, left, pivot_index - 1);//递归左半部分
    quickSort(A, pivot_index + 1, right);//递归右边部分
}
```

堆排序

算法思想

基于顺序表的完全二叉树，规则：根节点要么最大，要没最小。d 递归堆调整方法。

```
//堆排序
void heapify(int A[], int i, int size)  // 从A[i]向下进行堆调整，size 是待排序的元素个数
{
    int left_child = 2 * i + 1;         // 左孩子索引
    int right_child = 2 * i + 2;        // 右孩子索引
    int max = i;                        // 选出当前结点与其左右孩子三者之中的最大值
    if (left_child < size && A[left_child] > A[max])//当前结点与左孩子比较
        max = left_child;
    if (right_child < size && A[right_child] > A[max])//当前结点与右孩子比较
        max = right_child;
    if (max != i)//最大值不是当前结点
    {
        swap(A, i, max);                // 把当前结点和它的最大(直接)子节点进行交换
        heapify(A, max, size);          // 递归调用，继续从当前结点向下进行堆调整，求子树的最大值
    }
}

int buildHeap(int A[], int n)           // 建堆，时间复杂度O(n)
{
    int heap_size = n;
```

```
    for (int i = heap_size / 2 - 1; i >= 0; i--) // 从每一个非叶结点开始向下进行堆调整
    // heap_size / 2 - 1 的意思是  这个数之后就没有左右孩子了，所以从这个结点开始
        heapify(A, i, heap_size);
    return heap_size;
}

void heapSort(int A[], int n)
{
    int heap_size = buildHeap(A, n);    // 建立一个最大堆
    int k = 0;
    while (heap_size > 1)               // 堆（无序区）元素个数大于1，未完成排序
    {
        // 将堆顶元素与堆的最后一个元素互换，并从堆中去掉最后一个元素
        // 此处交换操作很有可能把后面元素的稳定性打乱，所以堆排序是不稳定的排序算法
        swap(A, 0, --heap_size);
        heapify(A, 0, heap_size);     // 从新的堆顶元素开始向下进行堆调整，时间复杂度O(logn)
    }
}
```

归并排序

排序思想

排序小组节点集合，两两归并相邻集合、完成排序。递归思想+归并思想。

```
//归并排序
void merge(int A[], int left, int mid, int right)// 合并两个已排好序的数组A[left...mid]和A[mid+1...right]
{
    int len = right - left + 1;
    int *temp = new int[len];       // 辅助空间O(n)
    int index = 0;
    int i = left;                   // 前一数组的起始元素
    int j = mid + 1;                // 后一数组的起始元素
    while (i <= mid && j <= right)
    {
        temp[index++] = A[i] <= A[j] ? A[i++] : A[j++];  // 带等号保证归并排序的稳定性
    }
    while (i <= mid)
    {
        temp[index++] = A[i++];
    }
```

```
    while (j <= right)
    {
        temp[index++] = A[j++];
    }
    for (int k = 0; k < len; k++)
    {
        A[left++] = temp[k];
    }
}

void mergeSortRecursion(int A[], int left, int right)    // 递归实现的归并排序(自顶向下)
{
    if (left == right)    // 当待排序的序列长度为1时，递归开始回溯，进行merge操作
        return;
    int mid = (left + right) / 2;
    mergeSortRecursion(A, left, mid);
    mergeSortRecursion(A, mid + 1, right);
    merge(A, left, mid, right);
}

void mergeSortIteration(int A[], int len)    // 非递归(迭代)实现的归并排序(自底向上)
{
    int left, mid, right;                    // 子数组索引,前一个为A[left...mid]，后一个子数组为A[mid+1...right]
    for (int i = 1; i < len; i *= 2)         // 子数组的大小i初始为1，每轮翻倍
    {
        left = 0;
        while (left + i < len)               // 后一个子数组存在(需要归并)
        {
            mid = left + i - 1;
            right = mid + i < len ? mid + i : len - 1;// 后一个子数组大小可能不够
            merge(A, left, mid, right);
            left = right + 1;                // 前一个子数组索引向后移动
        }

    }
}
```

希尔排序

排序思想

利用插入排序来排序间隔点，缩小间隔，完成排序！ 数量大时，可以用二分插入排序

```
//希尔排序
void mhellSort(int A[], int n)
{
    int h = 0;
    while (h <= n) {                    // 生成初始增量
        h = 3 * h + 1;
    }
    int k = 0;
    while (h >= 1){
        for (int i = h; i < n; i++){    //插入排序
            int j = i - h;
            int get = A[i];
            while (j >= 0 && A[j] > get)
            {
                A[j + h] = A[j];
                j = j - h;
            }
            A[j + h] = get;
        }
        h = (h - 1) / 3;                // 递减增量
        k++;
    }
}
```