W13笔记

Bubble sort

- **工作原理**

1. 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
2. 对每一对相邻元素做同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。
3. 针对所有的元素重复以上的步骤，除了最后一个。
4. 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

- **代码实现**

```
void sort(int a[], int begin, int end){
    if(end>begin){
        // bubble
        for(int i=begin; i<end; i++){
            if(a[i]>a[i+1]){
                // swap
                swap(a[i], a[i+1]);
            }
        }
        sort(a, begin, end-1);
    }
}
```

若文件的初始状态是正序的，一趟扫描即可完成排序。所需的关键字比较次数C和记录移动次数M均达到最小值：$C_{\min}=n-1$，$M_{\min}=0$。[1]

所以，冒泡排序最好的时间复杂度为$O(n)$。

若初始文件是反序的，需要进行$n-1$趟排序。每趟排序要进行$n-i$次关键字的比较(1≤i≤n-1)，且每次比较都必须移动记录三次来达到交换记录位置。在这种情况下，比较和移动次数均达到最大值：[1]

$$C_{\max}=\frac{n(n-1)}{2}=O(n^2)$$

$$M_{\max}=\frac{3n(n-1)}{2}=O(n^2)$$

冒泡排序的最坏时间复杂度为$O(n^2)$。[1]

综上，因此冒泡排序总的平均时间复杂度为$O(n^2)$。[1]

Insert sort

```
void insertSort(int* a,int T){
    int tmp,p;
    for(int i=1;i<T;i++){
        tmp=a[i];
        p=i-1;
        while(p>=0&&tmp<a[p]){
            a[p+1]=a[p];
            p--;
        }
        a[p+1]=tmp;
    }
}
```

```
void sort(int a[], int begin, int end){
    if(end>begin){
        sort(a, begin, end-1);
        for(int i=end; i>0; i--){
            if(a[i]<a[i-1]){
                swap(a[i], a[i-1]);
            }else{
                break; //不加break的话性能就和冒泡一样了
            }
        }
    }
}
```

1、当初始序列为正序时，只需要外循环n-1次，每次进行一次比较，无需移动元素。此时比较次数（C_{min}）和移动次数（M_{min}）达到最小值。

C_{min}=n-1

M_{min}=0

此时时间复杂度为$O(n)$。

2、当初始序列为反序时，需要外循环n-1次，每次排序中待插入的元素都要和[0,i-1]中的i个元素进行比较且要将这i个元素后移i次，加上`tmp=arr[i]`与`arr[j]=temp`的两次移动，每趟移动次数为i+2,此时比较次数和移动次数达到最大值。

C_{max} = 1+2+...+(n-1) = n*(n-1)/2*=$O(n^2)$

M_{max} = *(1+2)* + *(2+2)* +.....+ *(n-1+2)* = *(n-1)* (n+4)/2=$O(n^2)$

此时时间复杂度$O(n)$

Merge sort

- **算法描述**

第一步：申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列

第二步：设定两个指针，最初位置分别为两个已经排序序列的起始位置

第三步：比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置

重复步骤3直到某一指针超出序列尾，将另一序列剩下的所有元素直接复制到合并序列尾

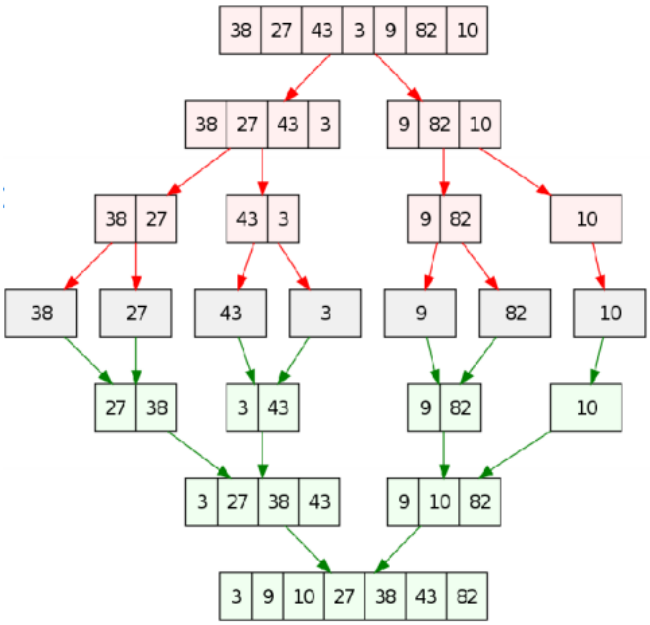
（1）“分解”——将序列每次折半划分（递归实现）

（2）“合并”——将划分后的序列段两两合并后排序

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
归并排序	O(n log n)	O(n log n)	O(n log n)	O(n)	Out-place	稳定

- **代码实现**

```
/*课堂代码*/
void merge(int a1[], int len1, int a2[], int len2, int t[], int lent){
    int p1=0, p2=0, p=0;
    while(1){
        if(a1[p1]<a2[p2]/*省略后面两个循环加一个||p2>=len2来表示p2已经凉了*/){
            t[p]=a1[p1];
            p1++;
        }else if(a1[p1]>=a2[p2]/*||p1>=len1*/){
            t[p]=a2[p2];
            p2++;
        }
        p++;
        if(p1>=len1||p2>=len2){
            break;
        }
    }
    for(; p1<len1; p1++){
        t[p++]=a1[p1];
    }
    for(; p2<len2; p2++){
        t[p++]=a2[p2];
    } //这两个循环其实可以省略掉
}
```




```
/* 将序列对半拆分直到序列长度为1*/
void MergeSort_UptoDown(int *num, int start, int end){
    int mid = start + (end - start) / 2;

    if (start >= end){
        return;
    }
```

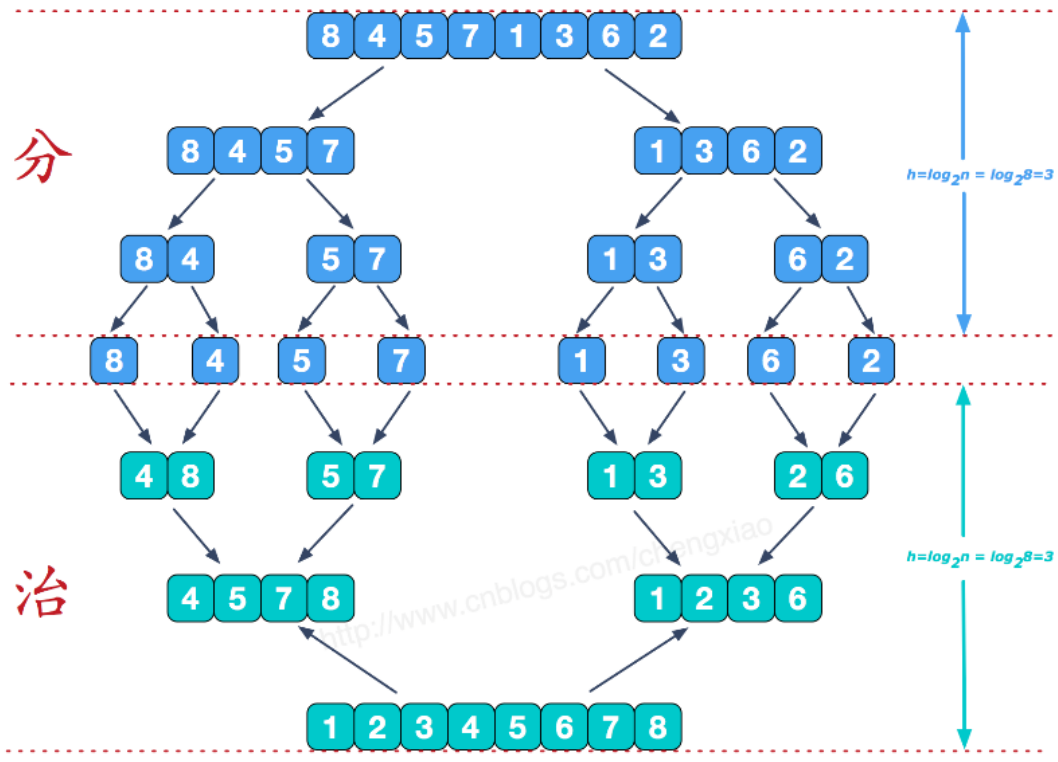
```
    MergeSort_UptoDown(num, start, mid);
    MergeSort_UptoDown(num, mid + 1, end);

    Merge(num, start, mid, end);
}

void Merge(int *num, int start, int mid, int end){
    int *temp = (int *)malloc((end-start+1) * sizeof(int));
    //申请空间来存放两个有序区归并后的临时区域
    int i = start;
    int j = mid + 1;
    int k = 0;

    while (i <= mid && j <= end){
        if (num[i] <= num[j]){
            temp[k++] = num[i++];
        }
        else{
            temp[k++] = num[j++];
        }
    }
    while (i <= mid){
        temp[k++] = num[i++];
    }
    while (j <= end){
        temp[k++] = num[j++];
    }
    //将临时区域中排序后的元素，整合到原数组中
    for (i = 0; i < k; i++){
        num[start + i] = temp[i];
    }

    free(temp);
}
```



Quick sort

- **基本思想**

通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

- **代码实现**

```
void quick_sort(int *arr,int left,int right){
    if(left>right) return;
    int pivot=getPivot();
    quick_sort(arr,left,pivot-1);
    quick_sort(arr,pivot+1,right);
}
```

- **划分方法**

(一) Lomuto Partition

This scheme chooses a pivot that is typically the last element in the array. The algorithm maintains index i as it scans the array using another index j such that the elements lo through i-1 (inclusive) are less than the pivot, and the elements i through j (inclusive) are equal to or greater than the pivot. This scheme degrades to $O(n^2)$ when the array is already in order

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi do
        if A[j] < pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

(二) Hoare partition

uses two indices that start at the ends of the array being partitioned, then move toward each other, until they detect an inversion: a pair of elements, one greater than or equal to the pivot, one lesser or equal, that are in the wrong order relative to each other. The inverted elements are then swapped. When the indices meet, the algorithm stops and returns the final index. Hoare's scheme is more efficient than Lomuto's partition scheme because it does three times fewer swaps on average, and it creates efficient partitions even when all values are equal

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
     pivot := A[lo + (hi - lo) / 2]
     i := lo - 1
     j := hi + 1
     loop forever
        do
            i := i + 1
         while A[i] < pivot
         do
            j := j - 1
         while A[j] > pivot
         if i >= j then
            return j
         swap A[i] with A[j]
```