基础-第一章

时间复杂度：一个算法流程中，常数操作的数量指标，这个指标叫O，也叫bigO。

空间复杂度：一个算法流程中，为了支持流程使用的辅助空间的指标，不包含输入和输出。

对数器：为了测试一个流程，写一个简答的绝对正确的方法，随机生成测试数据，对比二者的结果，打印出错的数据。

递归：程序调用自身的过程，借助栈来实现。定义一个过程，其会调用子过程，子过程和整体的处理流程是一样的，只是样本的范围发生了变化。

在写递归时，首先需要写baseCase，也就是递归什么时候结束。

稳定排序：相同的值在排序后其相对位置保持不变

冒泡排序

时间复杂度为O(\n^2)，空间复杂度为O(1)，稳定的排序算法

```
public static void BubbleSort(int[] arr) {
  if(arr == null || arr.length < 2) {
    return;
  }
  for(int e = arr.length - 1; e > 0; e--) {
    for(int i = 0; i < e; i++) {
      if(arr[i] > arr[i+1]) {
        swap(arr, i, i+1);
      }
    }
  }
}
```

插入排序

时间复杂度为O(n^2)，空间复杂度为O(1)，

```
public static void InsertSrot(int[] arr) {
  if(arr == null || arr.lenght < 2) {
    return;
  }
  for(int i = 1; i < arr.length; i++) {
    for(int j = i - 1; j >=0 && arr[j] > arr[j+1]; j--) {
      swap(arr, j, j+1);
    }
  }
}
```

归并排序

时间复杂度，空间复杂度

```
public static void MergeSrot(int[] arr) {
  if(arr == null || arr.lenght < 2) {
    return ;
  }
  mergeSort(arr, 0, arr.lenght-1);
}

public static void mergeSrot(int[] arr, int l, int r) {
  if(l == r) {
    return;
  }
  int mid = l + ((r - l) / 2); //保证不会溢出
  mergeSort(arr, l, mid);
  mergeSort(arr, mid + 1, r);
  merge(arr, l, mid, r);
}

public static void merge(int[] arr, int l, int mid, int r) {
  int[] help = new int[r - l + 1];
  int i = 0;
  int p1 = l;
  int p2 = m + 1;
  while(p1 <= m && p2 <= r) {
    help[i++] = arr[p1] < arr[p2] ? arr[p1++] : arr[p2++];
  }
  while(p1 <= m) {
    help[i++] = arr[p1++];
  }
  while(p2 <= r) {
    help[i++] = arr[p2++];
  }
```

```
  for(i = 0; i < help.length; i ++) {  //拷贝回原数组
    arr[l + i] = help[i];
  }
}
```

Pattition过程

时间复杂度，空间复杂度

> 以数组最后一个数为参考，将小于的放左边，等于的放中间，大于的放右边（可以无序）。

```
public static int[] partition(int[] arr, int L, int R) {
  int less = L - 1;
  int more = R;
  whilt(L < more) {
    if(arr[L] < arr[R]) {
      swap(arr, ++less, L++);
    } else if(arr[L] > arr[R]) {
      swap(arr, --more, L);
    } else {
      L++;
    }
  }
  swap(arr, more, R);
  return new int[] {less+1, more};
}
```

快速排序

时间复杂度，空间复杂度

快速排序是不稳定的，举一个例子：3,6,...0...,0,3；可以实现稳定排序，有一个论文叫《0 1 stable sort》。

```
public static void QuickSort(int[] arr) {
  if(arr == null || arr.length < 2) {
    return;
  }
  quickSosrt(arr, 0, arr.length - 1);
}

public static void quickSort(int[] arr, int l, int R) {
  if(L < R) {
    swap(arr, L + (int)(Math.random() * (R - L +1)), R);  //随机快速排序
```

```
    int[] p = partition(arr, L, r);
    quickSort(arr, L, p[0] - 1);
    quickSort(arr, p[l] + 1, r);
  }
}

public static int[] partition(int[] arr, int L, int R){
  ///实现方法同上
}
```

堆排序

时间复杂度，空间复杂度

堆结构：一颗完全二叉树结构，这棵树是一个满二叉树或者在通往满二叉树的路上，以及每一层节点都是依次从左往右填好。

将一个数组通过下标变换转化为一个树结构，针对一个下标i，其左子节点下标为2*i+1，其右子节点小标为2*i+2；同时其父节点下标可表示为i-1/2。

大根堆：针对一颗完全二叉树，整颗树的最大值就是头节点，每棵子树的最大值是子树的头节点。左右孩子没有关系。

例子：5 7 0 6 8

关键步骤：heapInsert（堆向上调整）；heapfiy（向下调整）

建立大根堆时间复杂度为O(N)，堆调整的时间复杂度O(N{logN})

```
// 非递归过程
public static void HeapSort(int[] arr) {
  if(arr == null || arr.lenght < 2) {
    return;
  }
  for(int i = 0; i < arr.length; i++) {  //建立大根堆heapInsert
    heapInsert(arr, i);
  }
  int size = arr.length;
  swap(arr, 0, size);
  while(size > 0) {   //堆向下调整heapify
    heapify(arr, 0, size);
    swap(arr, 0, --size);
  }
}

public static void heapInsert(int[] arr, int index) {
  while(arr[index] > arr[(index - 1) / 2]) {
    swap(arr, index, (index - 1) / 2);
```

```
    index = (index - 1) / 2;
  }
}

public static void heapify(int[] arr, int index, int size) {
  int left = index * 2 + 1;
  while(left < size) {
    int largest = left + 1 < size && arr[left + 1] > arr[left] ? left + 1 :
left;
    largest = arr[largest] > arr[index] ? largest : index;
    if (largest == index) {
      break;
    }
    swap(arr, largest, index);
    index = largest;
    left = index * 2 + 1;
  }
}
```

Java中的array.sort()

java中的排序方法采用**综合排序**，根据数据的规模进行划分。

- 当数组的size小于60时，采用InsertSrot()。
- 当数组的size大于60时，采用MergeSort()或者QuickSort()。
  - 针对基础数据类型：int. char, double等采用QuickSort()进行排序。因为在基础排序时，不要求数据的稳定性，针对一堆无差别的3，不需要排序稳定。
  - 针对自定义的class，后台默认采用MergeSrot()进行排序，需要自己实现比较器（继承后，计算需要排序的字段的差值大小返回即可）。在现实世界中，需要排序具有把原始相对次序向下传的特性，比如对用户国籍信息和年龄，当先按照年龄排序，再按照国籍排序时，可以在同国籍中保证年龄有序。

> 当数组的大小为300，先采用快排或者归并排序将数据划分为子块，当子快的size小于60时使用插入排序，虽然插入排序的时间复杂度为O(n^2)，但是其常数很小。

桶排序

非基于比较的排序方法，是基于数据状况的排序。可以实现为稳定的排序，将队列作为容器时是稳定的。

一个数据有4000个数，所有数的范围均为0～200，则首先生成200个容器（数组，列表都可以），针对每一个数是多少就放进第几个容器中，然后依次从每个容器中将数据全部倒出来。

```
public static void BucketSort(int[] arr) {
```

```
  if(arr == null || arr.length < 2) {
    return;
  }
  int max = Integer.MIN_VALUE;    // 获取数据状况
  for(int i = 0; i < arr.length; i++) {
    max = Math.max(max, arr[i]);
  }
  int[] bucket = new int[max + 1];    // 确定桶的数目
  for(int i = 0; i < arr.length; i++) {
    bucket[arr[i]]++;
  }
  int i = 0;
  for(int j = 0; j <bucket.length, j++) {
    while(bucket[j]-- > 0) {
      arr[i++] = j;
    }
  }
}
```

计数排序

类似于桶排序，每次不需要事先生成容器，直接记录数据出现的次数即可，最后按照次数打印。

基数排序

练习题

针对一个无序数组，计算排序后数组相邻元素的最大差值，要求时间复杂度为O(N)

> 利用桶排序的思想，遍历得到数组的最大最小值，根据数组的size建立size+1个桶，并且均分数组的最大值和最小值代表的范围，则最小的桶中一定有值，最大的桶中也一定有值。size个数，size+1个桶，则一定有一个桶为空，并且空桶左边桶的最大值和空桶右边桶的最小值在排序后一定相邻，则可以排除掉每个桶内部产生最大差值的可能性，则最大差值值可能在桶间产生。

```
public static int maxGap(int[] arr) {
  if(arr == null || arr.length < 2) {
    return;
  }
```

```
  int len = arr.length;
  int min = Integer.MAX_VALUE;
  int max = Integer.MIN_VALUE;
  for(int i = 0; i < len; i++) {
    min = Math.min(min, arr[i]);
    max = Math.max(max, arr[i]);
  }
  if(min == max) {
    return 0;
  }

  boolean[] hasNum = new boolean[len + 1];
  int[] maxs = new int[len + 1];
  int[] mins = new int[len + 1];
  int bid = 0;
  for(int i = 0; i < len; i++) {
    bid = bucket(arr[i], len, min, max);
    mins[bid] = hasNum[bid] ? Math.min(mins[bid], arr[i]) : arr[i];
    maxs[bid] = hasNum[bid] ? Math.max(maxs[bid], arr[i]) : arr[i];
    hasNum[bid] = true;
  }

  int res = 0;
  int lastMax = maxs[0];
  int i = 1;
  for(; i <= len; i++) {    //计算左桶最大值和右桶最小值的差值
    if(hasNum[i]) {
      res = Math.max(res, mins[i] - lastMax);
      lastMax = maxs[i];
    }
  }
  return res;
}

public static int bucket(long num, long len, long min, long max) {
  return (int)((num - min) * len / (max - min));
}
```