> 2018年12月23日

归并排序

1，算法思想

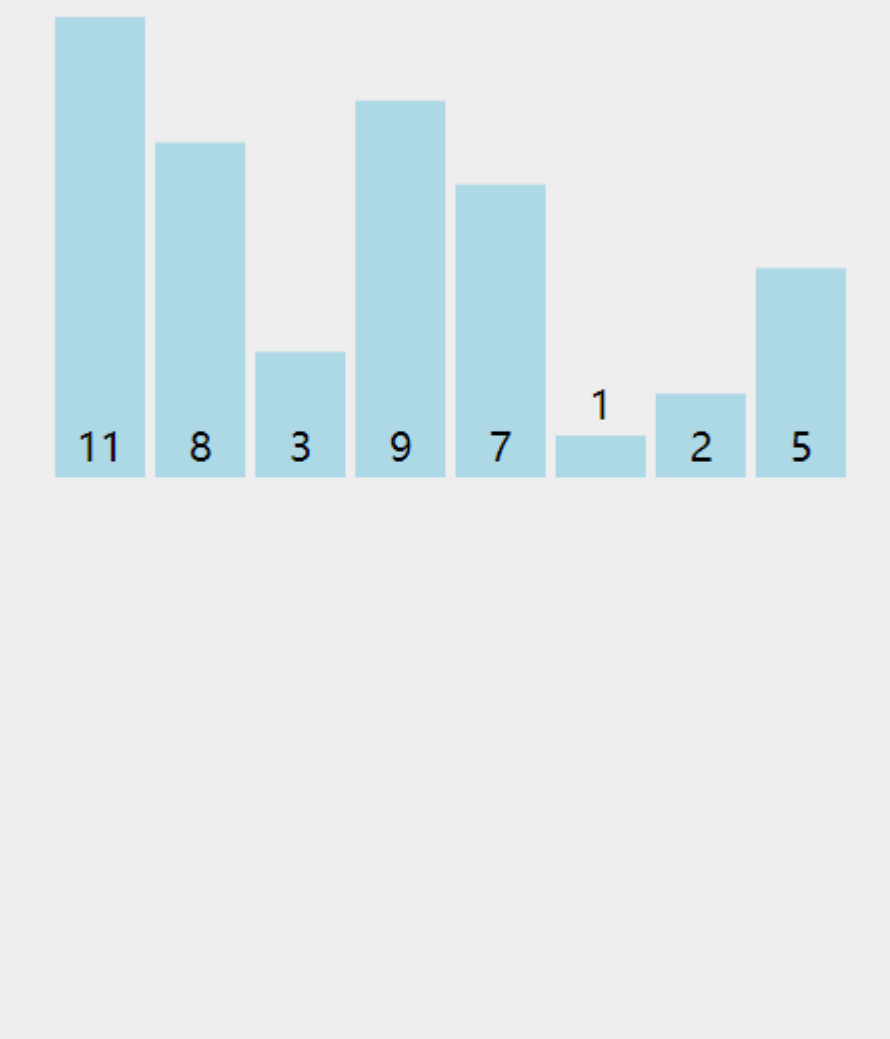
递归法（自上而下）

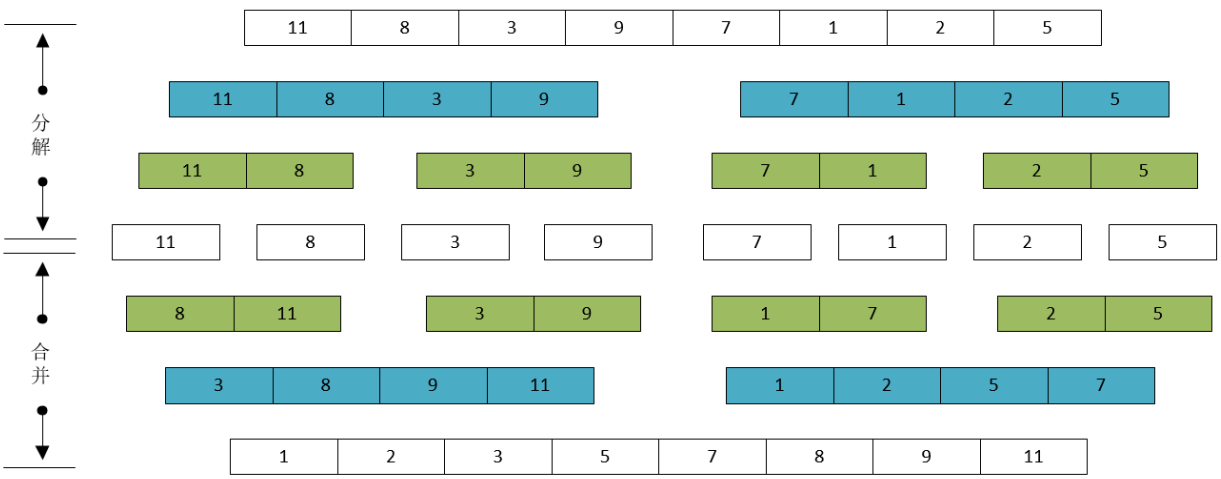
1. 申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列
2. 设定两个指针，最初位置分别为两个已经排序序列的起始位置
3. 比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置
4. 重复步骤3直到某一指针到达序列尾
5. 将另一序列剩下的所有元素直接复制到合并序列尾

迭代法（自下而上）

1. 将序列每相邻两个数字进行归并操作，形成$ceil(\frac{n}{2})$(向上取整)个序列，排序后每个序列包含两/一个元素
2. 若此时序列数不是1个则将上述序列再次归并，形成$ceil(\frac{n}{4})$个序列，每个序列包含四/三个元素
3. 重复步骤2，直到所有元素排序完毕，即序列数为1

2，算法图解





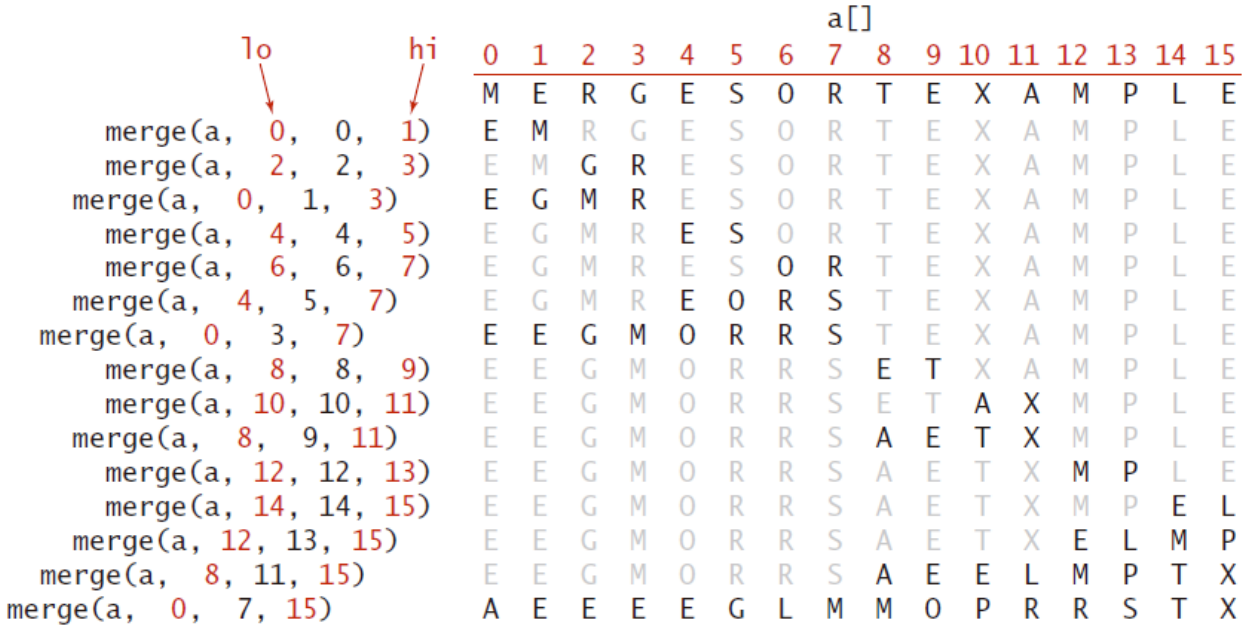
3，算法实现

```
1  public class Merge<AnyType extends Comparable<? super AnyType>> {
2      private AnyType[] aux;
3
```

```
    /**
     * 自上向下
     */
    public void sort(AnyType[] a) {
        aux = (AnyType[]) new Comparable[a.length];
        sort(a, 0, a.length - 1);
    }

    public void sort(AnyType[] a, int lo, int hi) {
        if (lo >= hi) {
            return;
        }
        int mid = lo + (hi - lo) / 2;
        sort(a, lo, mid);
        sort(a, mid + 1, hi);
        /**
         * 因为左右部分各自有序
         * 则只需a[mid]>a[mid+1]，才进行排序
         */
        if (a[mid].compareTo(a[mid + 1]) > 0) {
            merge(a, lo, mid, hi);
        }
    }

    public void merge(AnyType[] a, int lo, int mid, int hi) {
        /**
         * i从左半部分开始
         * j从右半部分开始
         */
        int i = lo;
        int j = mid + 1;
        for (int k = lo; k <= hi; k++) {
            aux[k] = a[k];
        }
        /**
         * 左右两半部分各自有序
         * 若左半部分用尽，则直接取右半部分元素
         * 若右半部分用尽，则直接去左半部分元素
         * 若右半部分当前元素小于左半部分当前元素，则取右半部分元素
         * 若右半部分当前元素大于左半部分当前元素，则取左半部分元素
         */
        for (int k = lo; k <= hi; k++) {
```

```
            if (i > mid) {
                a[k] = aux[j++];
            } else if (j > hi) {
                a[k] = aux[i++];
            } else if (aux[j].compareTo(aux[i]) < 0) {
                a[k] = aux[j++];
            } else {
                a[k] = aux[i++];
            }
        }
    }

    /**
     * 自下向上
     * 比较适合用链表组织的数据
     */
    public void sortBU(AnyType[] a) {
        int N = a.length;
        aux = (AnyType[]) new Comparable[a.length];
        for (int sz = 1; sz < N; sz *= 2) {
            for (int lo = 0; lo < N - sz; lo += 2 * sz) {
                /**
                 * mid=lo+((lo+sz+sz-1)-lo)/2=lo+(sz+sz-1)/2=lo+sz-1
                 * 之所以使用Math.min，是因为当N为奇数时的情况
                 */
                merge(a, lo, lo + sz - 1, Math.min(lo + sz + sz - 1, N
- 1));
            }

        }
    }
}
```

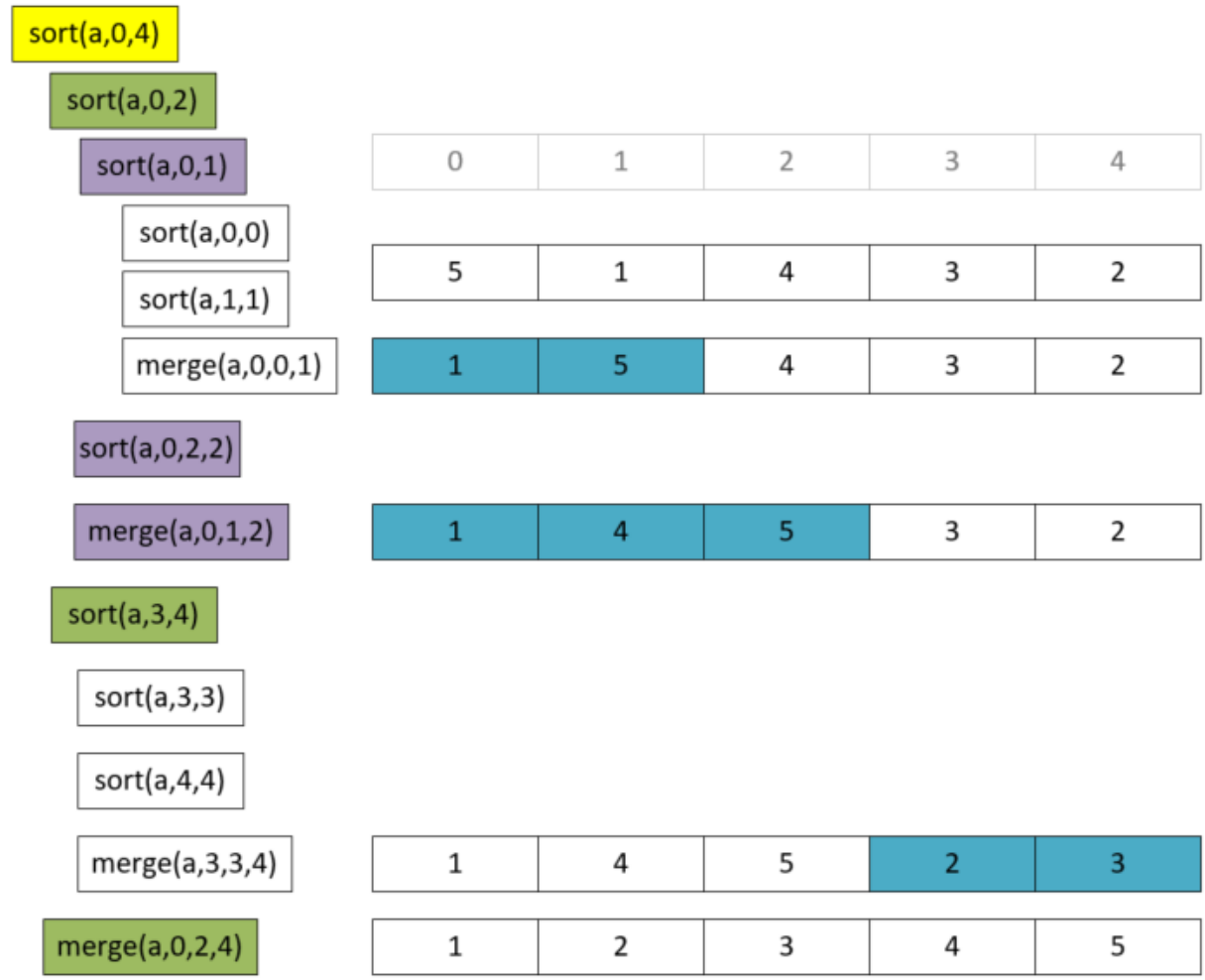
采用自上而下的方法其图解：序列个数为偶数




```
sort(a, 0, 15)
sort left half
  sort(a, 0, 7)
    sort(a, 0, 3)
      sort(a, 0, 1)
        merge(a, 0, 0, 1)
      sort(a, 2, 3)
        merge(a, 2, 2, 3)
      merge(a, 0, 1, 3)
    sort(a, 4, 7)
      sort(a, 4, 5)
        merge(a, 4, 4, 5)
      sort(a, 6, 7)
        merge(a, 6, 6, 7)
      merge(a, 4, 5, 7)
    merge(a, 0, 3, 7)
sort right half
  sort(a, 8, 15)
    sort(a, 8, 11)
      sort(a, 8, 9)
        merge(a, 8, 8, 9)
      sort(a, 10, 11)
        merge(a, 10, 10, 11)
      merge(a, 8, 9, 11)
    sort(a, 12, 15)
      sort(a, 12, 13)
        merge(a, 12, 12, 13)
```

```
          merge(a, 12, 12, 13)
        sort(a, 14, 15)
          merge(a, 14, 14,15)
        merge(a, 12, 13, 15)
      merge(a, 8, 11, 15)
merge
results  merge(a, 0, 7, 15)
```

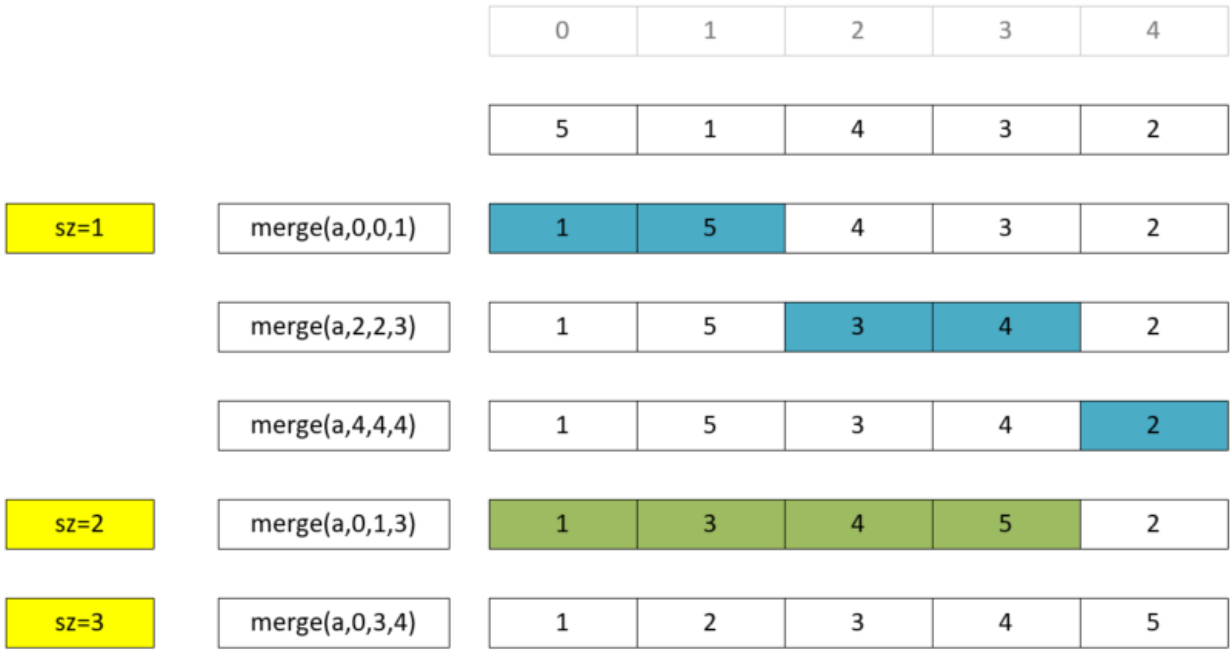
当序列的个数为奇数时：



采用自下而上的方法其图解：当序列个数为偶数时

		a[i]														
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sz = 1																	
	merge(a, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
	merge(a, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
	merge(a, 4, 4, 5)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
	merge(a, 6, 6, 7)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
	merge(a, 8, 8, 9)	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
	merge(a, 10, 10, 11)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
	merge(a, 12, 12, 13)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
	merge(a, 14, 14, 15)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L
sz = 2																	
	merge(a, 0, 1, 3)	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L
	merge(a, 4, 5, 7)	E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L
	merge(a, 8, 9, 11)	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
	merge(a, 12, 13, 15)	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P
sz = 4																	
	merge(a, 0, 3, 7)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
	merge(a, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
sz = 8																	
	merge(a, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

当序列个数为奇数时：



4，复杂度分析

- `merge`方法在将两个有序数组合并为一个有序数组时，需要借助额外的存储空间，其空间复杂度为$O(N)$。
- `merge`方法在合并数组时，不改变相同元素间的前后顺序，因此，归并排序时稳定的。 对N个数归并排序的用时等于完成两个大小为$\frac{N}{2}$的递归排序所用时间加上合并所用的时间，

对于`merge`方法，其时间复杂度为$O(N)$，当$N=1$时，归并排序所用时间为常数，记为1，那么有：$T(1)=1$ $T(N)=2T(\frac{N}{2})+N$ 令$N=\frac{N}{2}$，再将两边乘2，那么：$2T(\frac{N}{2})=2(2T(\frac{N}{4})+\frac{N}{2})=4T(\frac{N}{4})+N$ 因此可以得到：$T(N)=4T(\frac{N}{4})+2N$ 同理，令$N=\frac{N}{4}$，并在两边乘4，那么：$4T(\frac{N}{4})=4(2T(\frac{N}{8})+\frac{N}{4})=8T(\frac{N}{8})+N$ 又可以得到：$T(N)=8T(\frac{N}{8})+3N$ 总结规律有：$T(N)=2^kT(\frac{N}{2^k})+kN$ 其中$k=logN$ 最后有：$T(N)=NT(1)+NlogN=NlogN+N$ 因为归并排序的执行效率与待排序数组的有序程度无关，所以其时间复杂度是非常稳定的，因此，无论是最好、最坏还是平均情况下的时间复杂度都为$O(NlogN)$。

快速排序

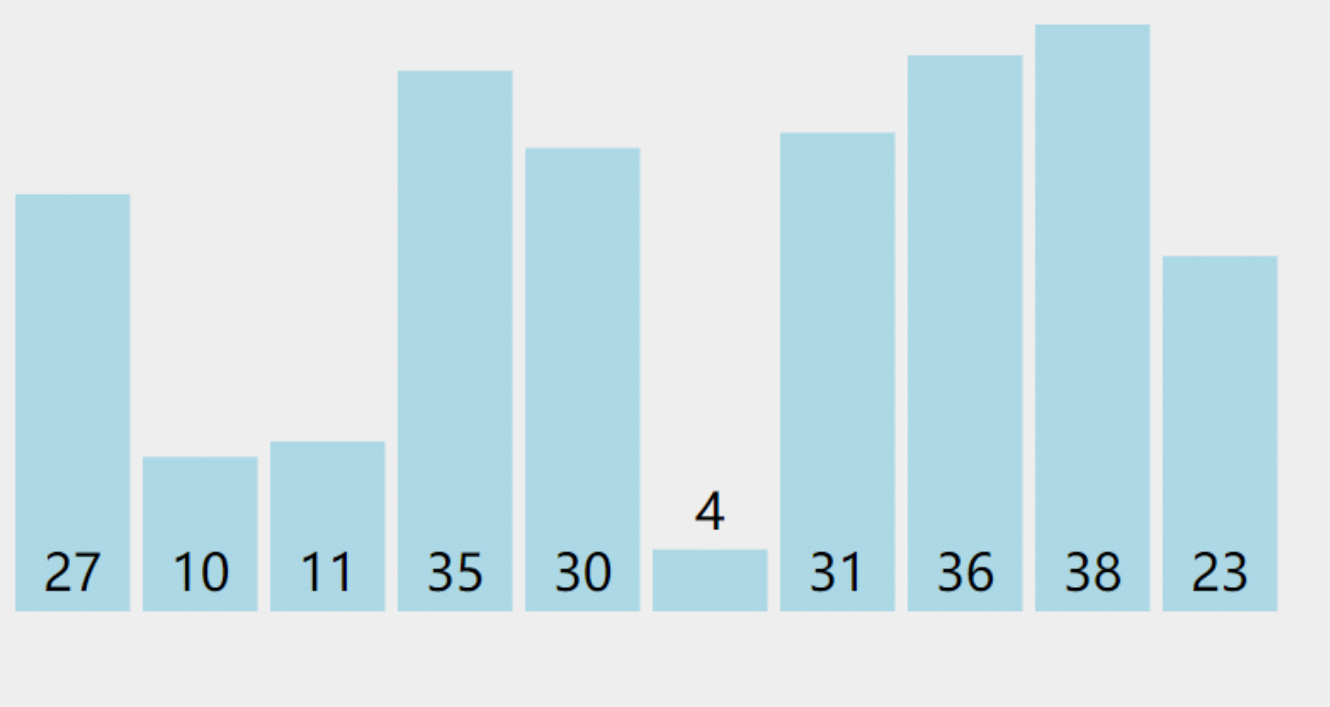
1，算法思想

快速排序使用分治法（Divide and conquer）策略来把一个序列（list）分为两个子序列（sub-lists）。

步骤为：

1. 从数列中挑出一个元素，称为“基准”（pivot），
2. 重新排序数列，所有比基准值小的元素摆放在基准前面，所有比基准值大的元素摆在基准后面（相同的数可以到任何一边）。在这个分割结束之后，该基准就处于数列的中间位置。这个称为分割（partition）操作。
3. 递归地（recursively）把小于基准值元素的子数列和大于基准值元素的子数列排序。 递归到最底部时，数列的大小是零或一，也就是已经排序好了。这个算法一定会结束，因为在每次的迭代（iteration）中，它至少会把一个元素摆到属于它的位置去。

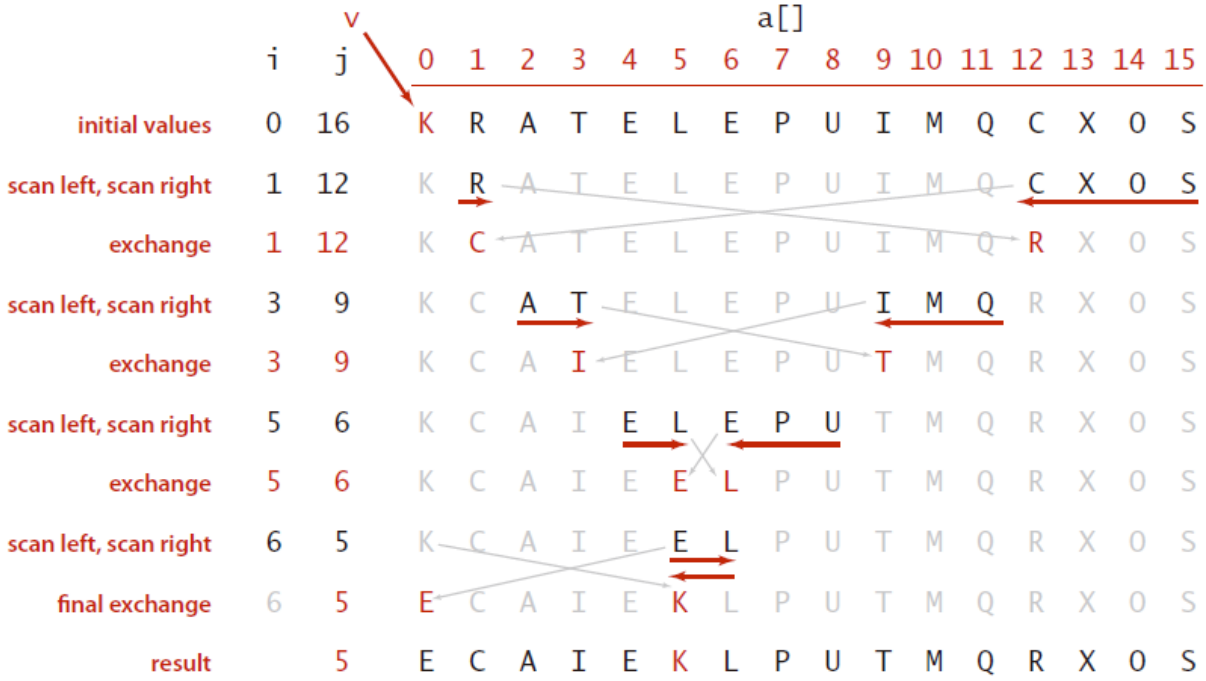
2，算法图解



3，算法实现

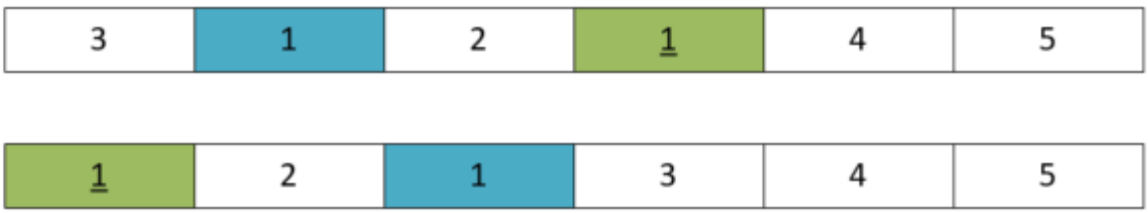
```
1  public class Quick<AnyType extends Comparable<? super AnyType>> {
2      public void sort(AnyType[] a) {
3          sort(a, 0, a.length - 1);
4      }
5
6      private void sort(AnyType[] a, int lo, int hi) {
7          if (lo >= hi) {
8              return;
9          }
10         int j = partition(a, lo, hi);
11         sort(a, lo, j - 1);
12         sort(a, j + 1, hi);
13     }
14
15     private int partition(AnyType[] a, int lo, int hi) {
16         int i = lo;
17         int j = hi + 1;
18         AnyType pivot = a[lo];
19         int size = hi - lo + 1;
20         if (size >= 3) {
21             pivot = medianOfThree(a, lo, hi);
22         }
23         while (true) {
```

```
            while (a[++i].compareTo(pivot) < 0) {
                if (i == hi) {
                    break;
                }
            }
            while (pivot.compareTo(a[--j]) < 0) {
                if (j == lo) {
                    break;
                }
            }
            if (i >= j) {
                break;
            }
            swap(a, i, j);
        }
        swap(a, lo, j);
        return j;
    }

    private AnyType medianOfThree(AnyType[] a, int lo, int hi) {
        int mid = lo + (hi - lo) / 2;
        if (a[lo].compareTo(a[hi]) > 0) {
            swap(a, lo, hi);
        }
        if (a[mid].compareTo(a[hi]) > 0) {
            swap(a, mid, hi);
        }
        if (a[mid].compareTo(a[lo]) > 0) {
            swap(a, mid, lo);
        }
        return a[lo];
    }

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



4，复杂度分析

- 最好的情况最多需要 $O(logN)$次的嵌套递归调用，所以它需要 $O(logN)$的空间。最坏情况下需要 $O(N)$次嵌套递归调用，因此需要 $O(N)$的空间。
- 如图，以3位基准，两个1前后顺序改变了，因此快速排序时不稳定的



快速排序的运行时间等于两个递归调用的运行时间加上`partition`方法运行时间：
$T(N) = T(i) + T(N-i-1) + N$ 令$T(0) = T(1) = 1$

- 最好情况下，基准`pivot`正好位于中间，那么$T(N) = 2T(\frac{N}{2}) + N$，该情况与上面归并排序分析相同，那么快速排序在最好情况下的时间复杂度为$O(NlogN)$。
- 最坏情况下，每次选取的基准`pivot`为最小元素，此时$i = 0$ $T(N) = T(N-1) + N$ $T(N-1) = T(N-2) + N - 1$ $T(N-2) = T(N-3) + N - 3$ . . . . . $T(2) = T(1) + 2$ 将两边相加后得到：$T(N) = \sum_{i=1}^{N} i = \frac{N(N-1)}{2}$ 因此，最坏情况下快速排序的时间复杂度为$O(N^2)$。为了避免这种极端情况，尽量避免基准`pivot`为最小元素，可以采用三数取中来选取`pivot`，即上述`medianOfThree`方法，该方法取头中尾三个数之间的中位数。

- 平均情况下，选取基准大小概率为$\frac{1}{N}$,那么$T(i)$的平均值为$\frac{1}{N}\sum_{j=0}^{N-1}T(j)$，那么$T(N)=T(i)+T(N-i-1)+N=\frac{2}{N}\sum_{j=0}^{N-1}T(j)+N$ 那么：$NT(N)=2\sum_{j=0}^{N-1}T(j)+N^2$ $(N-1)T(N-1)==2\sum_{j=0}^{N-2}T(j)+(N-1)^2$ 由上面式子可得：$NT(N)-(N-1)T(N-1)=2T(N-1)+2N-1$ $NT(N)=(N+1)T(N-1)+2N$ 又可以得到：$\frac{T(N)}{N+1}=\frac{T(N-1)}{N}+\frac{2}{N+1}$ $\frac{T(N-1)}{N}=\frac{T(N-2)}{N-1}+\frac{2}{N}$ $\frac{T(N-2)}{N-1}=\frac{T(N-3)}{N-2}+\frac{2}{N-1}$ $\cdots\cdots$ $\frac{T(2)}{3}=\frac{T(1)}{2}+\frac{2}{3}$ 最终可得：$\frac{T(N)}{N+1}=\frac{T(1)}{2}+2\sum_{i=3}^{N+1}\frac{1}{i}=ln(N+1)+\gamma-\frac{3}{2}$ 其中$\gamma\approx0.577$叫做欧拉常数，于是$\frac{T(N)}{N+1}=O(logN)$ $T(N)=O(NlogN)$ 所以快速排序的平均时间复杂度为$O(NlogN)$。

5，应用

```
LeetCode 215.数组中第K个最大元素
在未排序的数组中找到第 k 个最大的元素。
请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

示例 1:
输入: [3,2,1,5,6,4] 和 k = 2
输出: 5

示例 2:
输入: [3,2,3,1,2,4,5,5,6] 和 k = 4
输出: 4
```

使用快速排序思想，将K左右两边分区，大的在左边，小的在右边

```
public class FindKthLargest {
    public static int findKthLargest(int[] nums, int k) {
        if (nums == null || nums.length == 0) {
            return Integer.MAX_VALUE;
        }
        return findKthLargest(nums, 0, nums.length - 1, k-1);
    }

    private static int findKthLargest(int[] nums, int start, int end,
int k) {
        if (start > end) {
            return Integer.MAX_VALUE;
        }

```

```
14          int pivot = nums[end];
15          int left = start;
16          for (int i = start; i < end; i++) {
17              if (nums[i] > pivot) {
18                  swap(nums, left++, i);
19              }
20          }
21          swap(nums, left, end);
22
23          if (left == k) {
24              return nums[left];
25          } else if (left < k) {
26              return findKthLargest(nums, left + 1, end, k);
27          } else {
28              return findKthLargest(nums, start, left - 1, k);
29          }
30
31      }
32
33      private static void swap(int[] nums, int i, int j) {
34          int tmp = nums[i];
35          nums[i] = nums[j];
36          nums[j] = tmp;
37      }
38  }
```

第一次分区查找，需要对大小为N的数组进行分区，即需要遍历n个元素，平均情况下，在第二次分区查找时，需要对大小为$\frac{N}{2}$的数组进行分区，以此类推，分区遍历元素的个数分别为N、$\frac{N}{2}$、$\frac{N}{4}$.....1，我们知道等比数列求和公式： $S_n = \frac{a_1(1-q^n)}{1-q}$ 令$N = 2^k$，那么：$N + \frac{N}{2} + \frac{N}{4} + ... + 1 = 2^k + \frac{2^k}{2} + \frac{2^k}{4} + ... + 1 = 2^k + 2^{k-1} + 2^{k-2} + ... + 1 = 2^{k+1} - 1 = 2N - 1$ 所以该方法的时间复杂度为$O(2N-1) = O(N)$。

总结

排序算法	空间复杂度	稳定性	最好时间复杂度	最坏时间复杂度	平均时间复杂度
归并	$O(N)$	√	$O(NlogN)$	$O(NlogN)$	$O(NlogN)$
快速	$O(logN)$~$O(N)$	×	$O(NlogN)$	$O(N^2)$	$O(NlogN)$