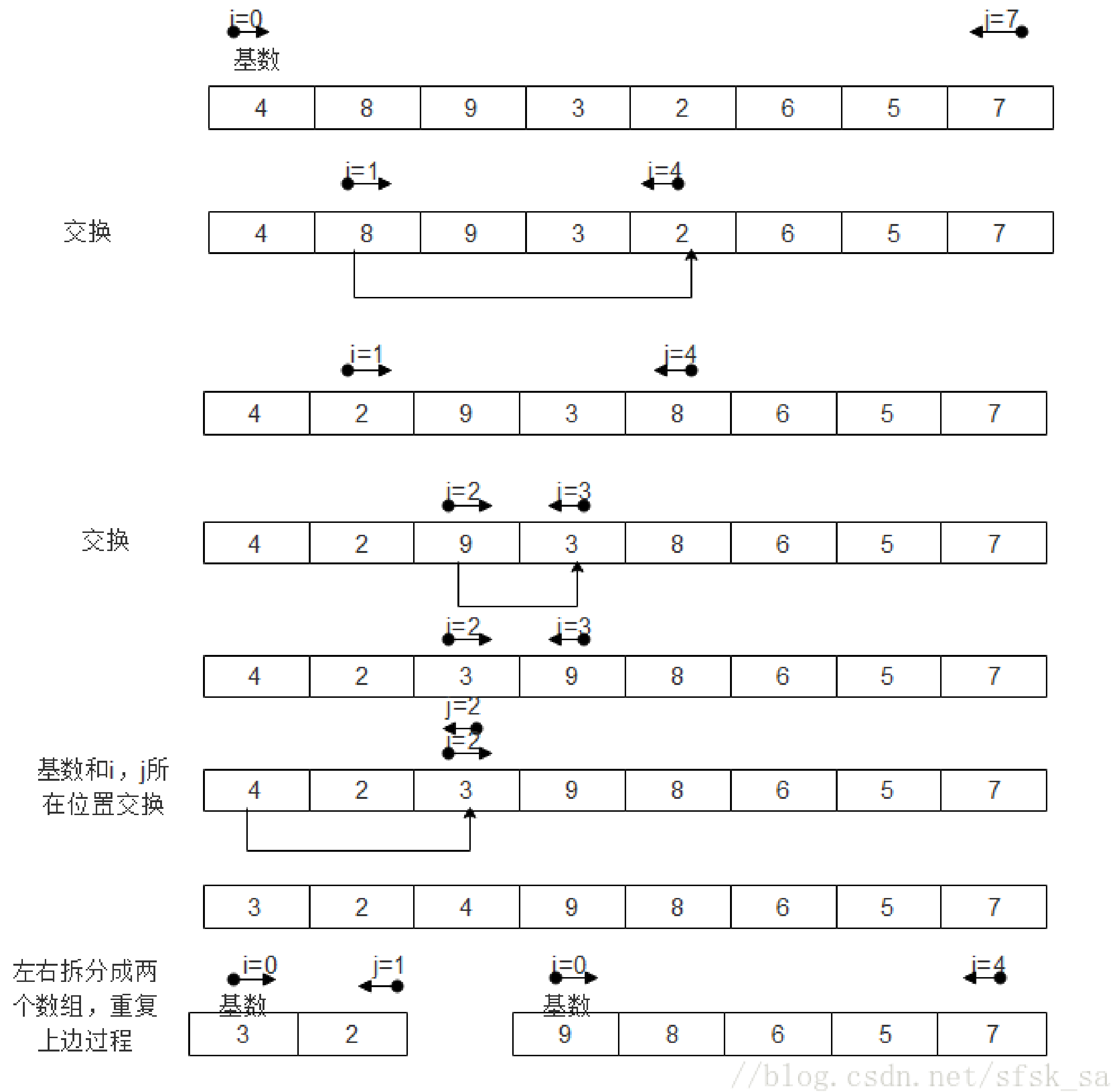
找数组中第K大的值（基于快速排序和最大堆排序两种）

一、快速排序

1. 基本的快速排序

关键是分而治之，找到pivot，然后分成两段（l，pivot-1）和（pivot+1，r）递归，主算法partition的作用就是，每次比pivot小的都放前面，比它大的丢后面，最后要保证pivot位置是两段的中间，左边都是比它小，右边都是比它大，分成两段后再重复此规律。



我们对数组 a[l … r]a[l⋯r] 做快速排序的过程是（参考《算法导论》）：

分解： 将数组 a[l … r]a[l⋯r] 「划分」成两个子数组 a[l … q - 1]a[l⋯q−1]、a[q + 1 … r]a[q+1⋯r]，使得 a[l … q - 1]a[l⋯q−1] 中的每个元素小于等于 a[q]a[q]，且 a[q]a[q] 小于等于 a[q + 1 … r]a[q+1⋯r] 中的每个元素。其中，计算下标 qq 也是「划分」过程的一部分。

解决： 通过递归调用快速排序，对子数组 a[l … q - 1]a[l⋯q−1] 和 a[q + 1 … r]a[q+1⋯r] 进行排序。

合并： 因为子数组都是原址排序的，所以不需要进行合并操作，a[l … r]a[l…r] 已经有序。 上文中提到的「划分」 过程是：从子数组 a[l … r]a[l…r] 中选择任意一个元素 xx 作为主元，调整子数组的元素使得左边的元素都小于等于它，右边的元素都大于等于它， xx 的最终位置就是 qq。

```
# 快速排序，有三种优化，搜索'优化'查看，封装排序算法
def quick_sort(A):
    if not A:
        return []
    size = len(A)
    qSort(A, 0, size - 1)
    return A

# 分成两段递归
# 优化3：size小于7是，插入排序效率更高，更大的时候用快速排序效率更高
def qSort(A, l, r):
    if l < r:
        pivot = partition(A, l, r)
        qSort(A, l, pivot - 1)
        qSort(A, pivot + 1, r)
# 主算法第一种写法
def partition(A, l, r):
    # 可以优化，取基准点，默认取第一个
    pivot = A[l]
    # 其实和人比较时，l始终代表的是pivot 的值；和l比较时（l，r互换了，pivot被换到右边了），r
代表的还是pivot 值
    # 最终l==r，即pivot应该在的位置
    while l < r:
        # >=等于号不可省略，否则左右区间内数相等会死循环，下同
        while l < r and A[r] >= pivot:
            # 意味着右边的值都是合理位置的，不需要变动，慢慢缩小区间，下同
            r -= 1
        # 上述条件之外的就需要交换了，丢到左边去
        A[r], A[l] = A[l], A[r]
        while l < r and A[l] <= pivot:
            l += 1
        A[l], A[r] = A[r], A[l]
    return l

# 主算法另外一种写法
def random_partition(nums, l, r):
    pivot = nums[l]
    j = l
    for i in range(l+1, r+1):
        if nums[i] < pivot:
            j +=1
            nums[i], nums[j] = nums[j], nums[i]
    nums[l], nums[j] = nums[j], nums[l]
    return l
```

2. 几种优化后的快速排序

   a. 基于基准pivot选择的优化，默认是选择左或右区间值，可以随机取或者取[l,mid,r]三个数再按大小取中间的那个值

   b. pivot 是换来换去的，实际上可以直接改成赋值，但是最后要记得把pivot放在他最终的位置（即l==r，分割两段区间的位置）

   c. size小于7是，插入排序效率更高，更大的时候用快速排序效率更高。略...

```python
# 主算法
def partition(A, l, r):
    # 优化1: 取基准。随机选择[l,r]区间中的一个数作为基准点，效果更好
    # random 选择pivot
    # import random
    i = random.randint(l,r)
    nums[i], nums[l] = nums[l], nums[i]

    # 优化1：取基准。取数组中左右中三个值，然后排序，pivot取中间那个数
    # m相当于加入第三个数可以选择，三数排序选中间值，(保证A[l]是中间大的值)
    m = (l + r) >> 1
    if A[l] > A[r]:
        A[r], A[l] = A[l], A[r]
    if A[m] > A[r]:
        A[r], A[m] = A[m], A[r]
    if A[m] > A[l]:
        A[m], A[l] = A[l], A[m]

    # 保证A[l]是我们要取得，优化的都放A[l]
    pivot = A[l]

    #最终l==r，即pivot应该在的位置
    while l < r:
        # >=等于号不可省略，否则左右区间内数相等会死循环，下同
        while l < r and A[r] >= pivot:
            r -= 1
        # 需要交换了
        # A[r], A[l] = A[l], A[r]
        # 优化二:  避免不必要的交换，直接覆盖值，但是最后再把pivot值赋值回去
        A[l] = A[r]
        while l < r and A[l] <= pivot:
            l += 1
        # A[l], A[r] = A[r], A[l]
        # 优化2:  避免不必要的交换，直接覆盖值，但是最后再把pivot值赋值回去
        A[r] = A[l]
    # 最后再赋值回来
    A[l] = pivot
    return l

# 插入排序
def insert_sort(A):
    if not A:
        return []
    size = len(A)
    for i in range(1,size):
        while i > 0 and A[i] < A[i-1]:
            A[i], A[i-1] = A[i-1], A[i]
            i -= 1
    return A
```

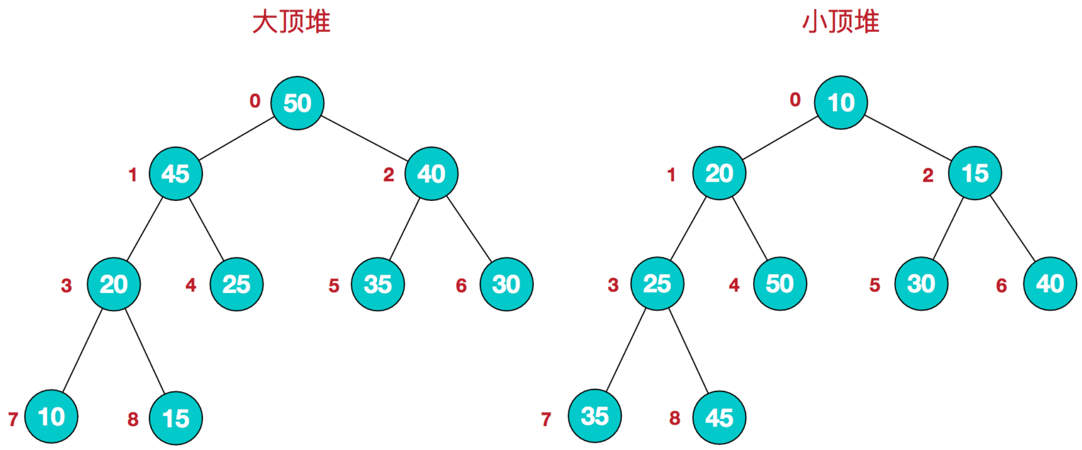
3. 找第K个大的数，快速选择法。每次pivot所在位置即右边区间全是大于它的数，所以只要找到pivot在index = size-k的位置，此时pivot所在位置就是第k大的数了（一般从小到大排序嘛..）。当pivot下标p小于index，即当前pivot是第p大，不够第index大的，在右区间（更大数里面）找；反之则在左区间找。

```python
# 找到最大的第K个数，使用快速排序思想,只要pivot下标等于size-k即可
```

```python
def findKthLargest( nums, k):
    size = len(nums)
    return quickSelect(nums, 0, size - 1, size - k)

# 当pivot下标p小于index，即当前pivot是第p大，不够第index大的，在右区间（更大数里面）找；反之则在左区间找。
def quickSelect(nums, l, r, index):
    # 写法一：迭代
    while  True:
        q = random_partition(nums, l, r)
        if q == index:
            return nums[q]
        elif q > index:
            r = q-1
        else:
            l = q+1
    # 写法二：递归
    # q = random_partition(nums, l, r)
    # if q == index:
    #     return q
    # else:
    #     return quickSelect(nums, l, q-1, index) if q > index else quickSelect(nums, q+1, r, index)

# 和上面的快速排序是一模一样，只是用random的优化方法，改了个名字
def random_partition(nums, l, r):
    # random 选择pivot
    i = random.randint(l,r)
    nums[i], nums[l] = nums[l], nums[i]
    pivot = nums[l]
    j = l
    for i in range(l+1, r+1):
        if nums[i] < pivot:
            j +=1
            nums[i], nums[j] = nums[j], nums[i]
    nums[l], nums[j] = nums[j], nums[l]
    return l
```

二、堆排序

1. 基本堆排序。最大堆（ai >= 2*ai and* >= 2ai+1），父节点值永远比子节点大，i是父节点index，2*ai 就是左子。最小堆（ai* <= 2ai and <= 2*ai+1）。每次维护一个最大堆，保证正最顶上那个节点是最大值，然后把它丢到最后一个节点的位置，剩下的元素中再维护最大堆得到新的最大值的头结点，依次重复。




```
# heap sort 堆排序， 从最底下开始，构建最大堆（i>= 2i 和2i+1）或最小堆
def heap_sort(A):
    if not A:
         return []
    size = len(A)
    # size//2 是层数i（父节点位置）， size//2-1表示小标从0开始，python方便操作
    # 确认最深最后的那个根节点的位置
    first_root = size//2 -1
    # 先调整，第一次构建最大堆
    for i in range(first_root, -1, -1):
        heap_adjust(A, i, size -1 )
    # 此时刚构建第一个最大堆，需要把第一个值和最后一个值交换，
    # 之后在剩下元素中再构建最大堆
    for end in range(size-1, 0, -1):
        # 把最大的元素（索引为1）与最后一个元素互换
        A[end], A[0] = A[0], A[end]
        # 在剩下元素中再构建最大堆
        heap_adjust(A, 0, end-1)
    return A

# root 代表当前父节点所在下标，2root就是它的左节点，2root+1是又节点，下一个父节点是2 *
root
# end 表示当前剩下元素个数
def heap_adjust(A, root, end):
    while True:
        # 左子节点的位置
        child = 2 * root + 1
        # 若左子节点超过了最后一个节点，则终止循环
        if child > end:
            break
        # child 值不需要更新，一直找到最大的child
        if child+1 <= end and A[child] < A[child+1]:
            child += 1
        # 孩子比父节点大，交换值，并且将根节点指针，移到这个孩子节点上
        if A[child] > A[root]:
            A[child], A[root] =  A[root], A[child]
            # 很关键，相当于把当前父节点移动到需要判断的下一个父节点
            root = child
        else:
            # 父节点最大，没有子节点比它大，直接跳出
            break
```

2. 找最大的K个值，基于最大堆思想。

   每次调整最大堆都是把最大堆顶点和尾值互换，第k次调整就是第k大的数。

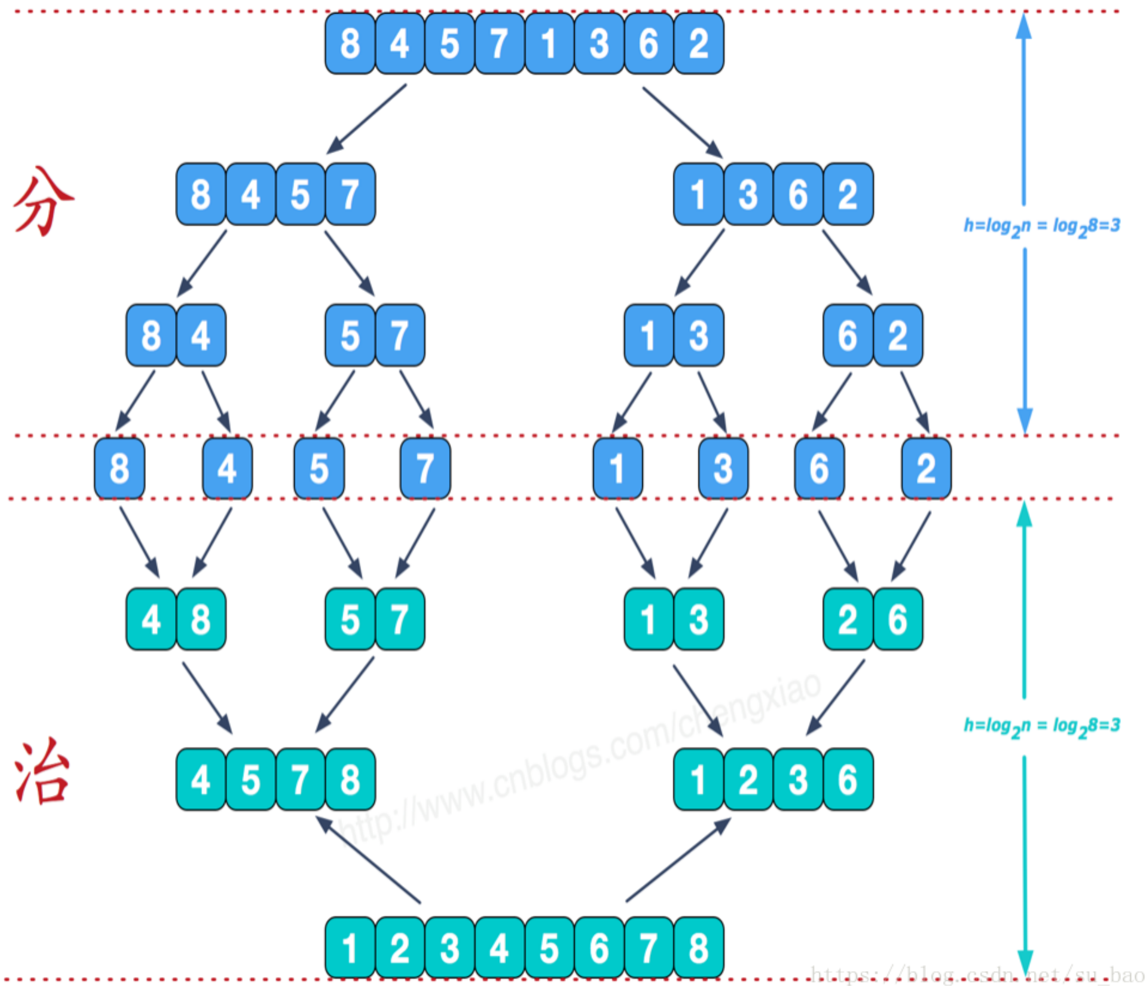
```
# 找到第K个最大的元素
# 建立一个大根堆，做 k - 1k-1 次删除操作后堆顶元素就是我们要找的答案
def findKthLargestHeap(A, K):
    size = len(A)
    return heapSelect(A, size, K)

# 第k次调整就是第k大的数
def heapSelect(A, size, K):
    first_root = size//2 -1
    for i in range(first_root, -1, -1):
        heap_adjust(A, i, size-1)
    # count记录调整的次数，已经调整过一次了，初始为1
    count = 1
    for end in range(size-1, 0, -1):
        # 调整到第k次，最顶上那个就是第k大的数，每次最顶上都是现有节点中最大的数
        if count == K:
            break
        # 每次调整，次数加1
        count += 1
        A[end], A[0] = A[0], A[end]
        heap_adjust(A, 0, end-1)
    return A[0]


# root 代表当前父节点所在下标，2s就是它的左节点，2s+1是又节点，下一个父节点是i*=2
# end 表示当前剩下元素个数
def heap_adjust(A, root, end):
    while True:
        # 左子节点的位置
        child = 2 * root + 1
        # 若左子节点超过了最后一个节点，则终止循环
        if child > end:
            break
        # child 值不需要更新，一直找到最大的child
        if child+1 <= end and A[child] < A[child+1]:
            child += 1
        # 孩子比父节点大，交换值，并且将根节点指针，移到这个孩子节点上
        if A[child] > A[root]:
            A[child], A[root] =  A[root], A[child]
            # 很关键，相当于把当前父节点移动到需要判断的下一个父节点
            root = child
        else:
            # 父节点最大，没有子节点比它大，直接跳出
            break
```

三、归并排序

顺带记下归并排序。归并排序（MERGE-SORT）采用分治法（Divide and Conquer），将原问题划分成n 个规模较小而结构与原问题相似的子问题；递归地解决这些问题，然后再合并其结果，就得到原问题的解。归并排序思想：

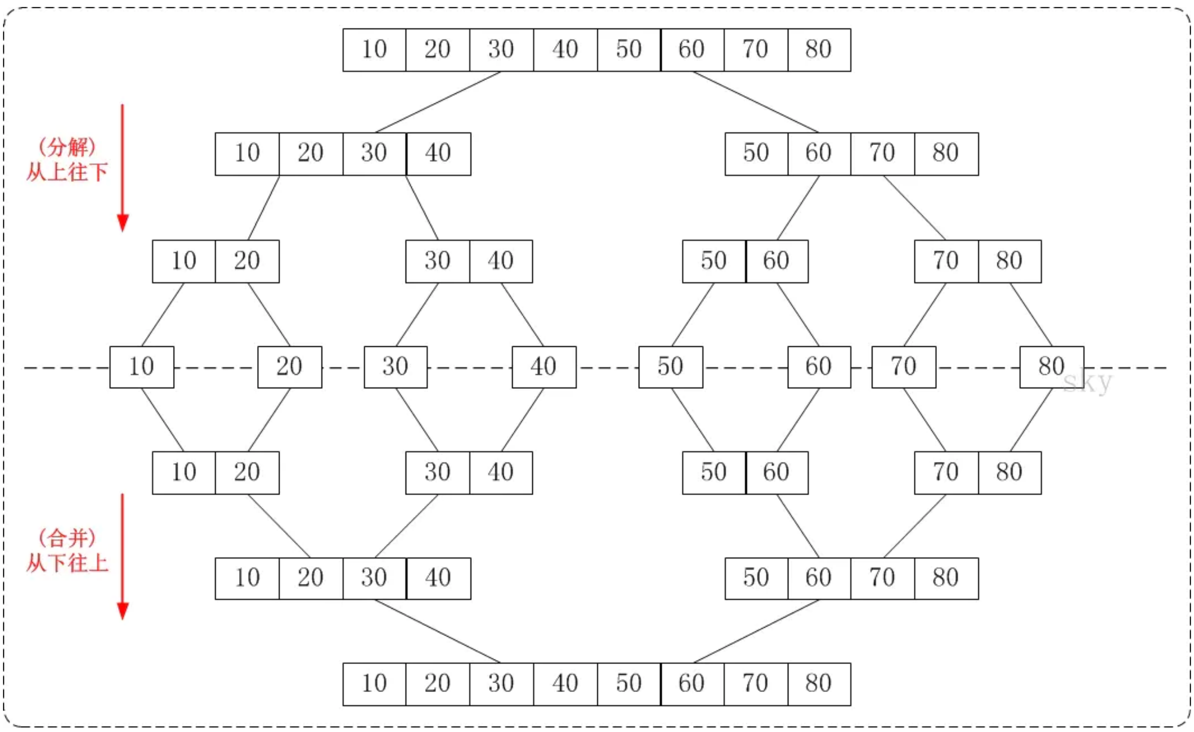
1. 将一个序列从中间位置分成左右两个序列；
2. 在将这两个子序列按照第一步继续二分下去；

3. 直到所有子序列的长度都为1，也就是不可以再二分截止。这时候再两两合并成一个有序序列即可。



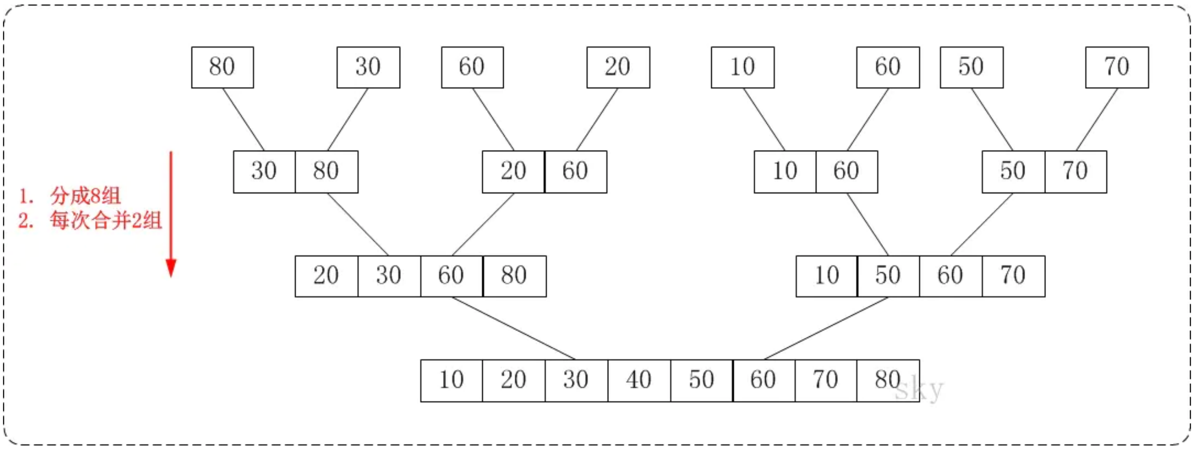
归并排序」比「快速排序」好的一点是，它借助了额外空间，可以实现「稳定排序」。归并排序的最好，最坏，平均时间复杂度均为**O(nlog^n)**。两种写法，一种自下向上的递归形式（先分到不能分，然后从最底下的小问题开始操作归并），在合并子列时需要申请临时空间空间复杂度为**O(n)**。另外一种自上向下的迭代写法，空间复杂度**O(1)**。

a. 递归写法



```python
# 归并排序，递归写法，每次去mid把数组分成左右两部分直到不能分，然后最后实现归并比较
# 从下至上的递归，最底下是A[0] 和 A[1]比较...
def merge_sort(A):
    if len(A) < 2:
        return A
    size = len(A)
    mid = size//2
    left = merge_sort(A[:mid])
    right = merge_sort(A[mid:])
    return merge(left,right)

# 实现左右比较并且归并
def merge(left, right):
    a, b = len(left), len(right)
    # 用来遍历左数组 和右数组，每次比较完一个就把小的加入结果并且跳到该数组的下一个
    i, j = 0, 0
    # 存放结果，所以递归需要额外空间
    res = []
    while i < a and j < b:
        # 左数组i的值小于等于当前右数组j的值（稳定性），那把它丢进去，index跳到下一个。反之丢右数组
        if left[i] <= right[j]:
            res.append(left[i])
            i += 1
        else:
            res.append(right[j])
            j += 1
    # 上述条件是左数组或右数组中一个遍历完了就停止，此时另外一个数组可能还有货
    # 因为从下至上的递归，左右单个数组是已经排好顺序了，所以直接把另一数组添加到末尾即可
    res += left[i:]
    res += right[j:]
    # 返回最终排序好的归并后数组即可
    return res
```

b. 迭代写法。貌似不太常用，了解一下。




```python
# 非递归的归并排序
def merge(A, low, mid, high):
    left = A[low: mid]
    right = A[mid: high]
    k = 0
    j = 0
    result = []
    while k < len(left) and j < len(right):
```

```python
        if left[k] <= right[j]:
            result.append(left[k])
            k += 1
        else:
            result.append(right[j])
            j += 1
    result += left[k:]
    result += right[j:]
    A[low: high] = result

def merge_sort_no(A):
    i = 1 # i是步长
    while i < len(A):
        low = 0
        while low < len(A):
            mid = low + i #mid前后均为有序
            high = min(low+2*i,len(A))
            if mid < high:
                merge(A, low, mid, high)
            low += 2*i
        i *= 2
```