assignment2 - 排序军训

Chapter 4 分治排序

4.1

证明假设一颗二叉树的高度为 h，叶节点个数为 L, 证明 $L \leq 2^h$
Proof:
Use induction.
Base: when h=1, it' s obvious that $L = 1 \leq 2^1$
I.H: we make an assumption that for $h \leq k$, we hold that $L \leq 2^h$
Step: we now need to prove $h = k+1$ still hold that $L \leq 2^h$

For an tree fit $h \leq k$, we add node to this tree to consider this problem.

Case 1 : add a node to leaf node who higher than h, then,$k = k'$, $h = h'$. OK
Case 2 : add a node to leaf node in h floor, then $h = h+1, L = L+1$, easily to prove it still hold $L \leq 2^h$
Case 3: based on case 2, when the tree' s height +1, we continue to add a node to leaf node in h floor, that is $h = k+1$, then L will plus 1 each time while h is fixed.
$L' \leq 2L$, we have $L \leq 2^h$, so $L' \leq 2L \leq 2 \times 2^h = 2^{h+1}$
So, for $h = k+1$, still hold that $L \leq 2^h$
Q.E.D

4.4

5 COMPARISONS TO SORT 4 ELEMENTS

```
def sort4elements(a,b,c,d):
    A=min(a,b)
    B=a+b-min(a,b)
    C=min(c,d)
    D=c+d-min(c,d)
    # cost 2 COMPARISONS, info: A<B, C<D
    C=min()
```

4.1

证明假设一颗二叉树的高度为 h，叶节点个数为 L, 证明 $L \leq 2^h$
Proof:
Use induction.
Base: when h=1, it' s obvious that $L=1 \leq 2^1$
I.H: we make an assumption that for $h \leq k$, we hold that $L \leq 2^h$
Step: we now need to prove $h=k+1$ still hold that $L \leq 2^h$

For an tree fit $h \leq k$, we add node to this tree to consider this problem.

Case 1 : add a node to leaf node who higher than h, then, $k=k'$, $h=h'$. OK
Case 2 : add a node to leaf node in h floor, then
$h=h+1, L=L+1$, easily to prove it still hold $L \leq 2^h$

Case 3: based on case 2, when the tree' s height +1, we continue to add a node to leaf node in h floor, that is $h = k + 1$, then L will plus 1 each time while h is fixed.
$L' \leq 2L$, we have $L \leq 2^h$, so $L' \leq 2L \leq 2 \times 2^h = 2^{h+1}$
So, for $h = k + 1$, still hold that $L \leq 2^h$
Q.E.D

4.4

5 COMPARISONS TO SORT 4 ELEMENTS

```
def sort4elements(a,b,c,d):
    # cost 2 COMPARISONS, info: A<B, C<D
    A=min(a,b)
    B=a+b-min(a,b)
    C=min(c,d)
    D=c+d-min(c,d)
    #find the largest element,3rd comparison
    tmpD=max(B,D)
    B=B+D-max(B,D)
    D=tmpD

    #find the smallest element
    #4th comparison
    A=min(A,C)#D>A
    C=A+C-min(A,C)
    #5th comparison. to find B(2nd smallest). B in{B,C}
    B=min(B,C)#B>A
    C=B+C-min(B,C)
    #finished. ABCD is sorted abcd
    return [A,B,C,D]
```

7 COMPARISONS TO SORT 5 ELEMENTS

```
def sort5elements(a,b,c,d, e):
    #我们在4 element基础上维护一个E
    A=min(a,b)
    B=a+b-A
    #E>D
    E=max(d,e)
    d=d+e-E
    #E>C
    C=min(c,d)
    D=c+d-C

    #C<D<E
    E=max(D,E)
    D=E+D-max(D,E)

    #A<B,A<C<D<E
    tA=min(A,C)
    C=A+C-min(A,C)
    A=tA
    #B<D,B<E,A<B<C<D, C<E
    tB=min(B,D)
    D=B+D-min(B,D)
    tB=B
    #by above analysis,we know E>C and E>B and,A<B<C<D, so E only can be between C and D, or larger than D, one and only one more comparison needed.
    if(E<D):
        swap(E,D)
8   return [A,B,C,D,E]
```

4.8 k-sorted

每次找到一个 part 中最大的元素作为 Pivot. 然后进行类似于 mergesort 的分治排序。

```
def ksort(arr[],int start,int end, int k)
    #if arr need to be k-sorted, then half of arr need
to be k/2 - sorted
    #pose a cursive equatation
    #cursive end condition:
    if(k<=1):
        return
    n=end-start+1
    mid=n/2
    leftMax=max in arr[:n/2]
    rightMax=max in arr[n/2:]

    if leftMax>rightMax:#swap two part to be 2-sorted
        swap arr[:n/2] and arr[n/2:]
    ksort(arr,start,mid,k/2)
    ksort(arr,start,mid,k/2)
```

分析复杂度:

$f(n) = \text{找到每一段最大值的比较次数} = k \times \frac{n}{k} = n$

递归表达式:

$T(k) = 2T\left(\frac{k}{2}\right) + n$

And we hold that $n = ck$ for some constant c

By Master Theorem:

$T(k) \in O(n \log k)$

4.9 Bolts and Nuts

先选一个螺母，然后遍历螺钉找到与其匹配的螺钉。然后通过螺母将螺钉分两类：大的，小的. 通过螺钉将螺母分类. 然后分治的继续做下去直到全部匹配
代码框架如下：

```
def bolts_nuts(bolts[], nuts[]):
    bolt=bolts[0] #pick one bolt in bolts
    small nuts=[i:i in nuts and i<bolt]
    large nuts=[i:i in nuts and i >bolt]
    nut=some nut in nuts that nut fit bolt
    small bolts=[i:i in bolts and i<nut]
    large bolts=[i:i in bolts and i>nut]

    bolts_nuts(small nuts, small bolts)
    bolts_nuts(large nuts, large bolts)
```

4.11

已知数组 $A[1,...,n]$ 至多有 2 个逆序对。
1）证明若 （i,j） 为逆序对，则 $j-i<=2$
2）请设计一个算法将数组中的元素排序，要求算法在最坏情况的比较次数不超过 n
Solution：
1):
用反证法。假设 (i,j) 是逆序对，这里不妨假设升序是正序,且 $j-i>2$, 那么根据逆序对的定义，我们知道 $A[i]>A[j]$, 且 index i 和 j 之间的其他元素都符合正序

则至少有：$A[j] < A[i] < A[i+1] < A[i+2] < \dots$
而我们知道, $j-i>2$，即 $j-(i+1)>0 \wedge j-(i+2)>0$
则 $(i+1,j),(i+2,j)$ 也构成逆序对，有 3个逆序对，矛盾。Q.E.D

2):
由 1) 结论，我们利用一个 width=3 的 slilding window 即可查询所有逆序对并更改，从而完成排序
只需要遍历数组一遍，故比较次数不超过 n。下面是实现

```
def sort(A[]):
    #the start of sliding window
    int i=0;
    int wid=3;
    while(i+wid<len(A)):
        for(j=0;j<wid;j++):
            if(A[i+j]<A[i]):
                SWAP(A[i+j],A[i])
```

该循环最多执行 $len(A)-wid+wid=len(A)=n$ 次. 符合要求

4.14（易位词）

易位词：改变单词字母顺序组成另外一个单词。构造算法找出篇幅很大的英文文件中的所有易位词。

原理：将文章中的每个单词，统计他们各个字母出现的次数，如果没出现就记为 0. 则每类易位词可以获得一个唯一确定的编码. 我们利用哈希表将每个单词映射到哈希表，编码相同的就是同一个易位词，通过这个方法我们就可以找到所有的易位词。

Chapter 7

7.1

设计算法计算广义逆序对。
思路：只需要在普通逆序对算法基础上，将比较条件做一下更改即可。

```
def count_generalized_inversions(A, C):
    def merge_and_count(arr, temp, left, mid, right):
        i, j, k = left, mid + 1, left
        inv_count = 0
        # 将右侧区间预处理为乘以C后的值，方便判断.注意到这不影响右侧区间内部的计算逆序对。
        right_scaled = [C * arr[x] for x in range(mid + 1, right + 1)]

        # 按照归并的方式统计广义逆序对
        while i <= mid and j <= right:
            if arr[i] > right_scaled[j - mid - 1]:
                # 如果左边的值比右边满足条件，统计逆序对的个数
                inv_count += (right - j + 1)
                temp[k] = arr[i]
                i += 1
            else:
                temp[k] = arr[j]
                j += 1
            k += 1

        # 将剩余的元素归并
        while i <= mid:
            temp[k] = arr[i]
```

```
            i += 1
            k += 1
        while j <= right:
            temp[k] = arr[j]
            j += 1
            k += 1

        # 更新原数组
        for i in range(left, right + 1):
            arr[i] = temp[i]

        return inv_count

    def merge_sort_and_count(arr, temp, left, right):
        if left >= right:
            return 0

        mid = (left + right) // 2
        inv_count = 0

        # 递归统计左侧、右侧以及跨区间的广义逆序对
        inv_count += merge_sort_and_count(arr, temp,
left, mid)
        inv_count += merge_sort_and_count(arr, temp,
mid + 1, right)
        inv_count += merge_and_count(arr, temp, left,
mid, right)

        return inv_count

    n = len(A)
    temp = [0] * n
    return merge_sort_and_count(A, temp, 0, n - 1)
```

7.4

假设我们有 k 个数组，每个数组中有 n 个排好序的元素（总共有 nk 个元素）。现在需要将这些数组合并成一个排好序的数组。

1）：

分析：数组 1 和数组 2 merge. Merge 需要遍历两个数组，cost $2n$

接下来合并后的数组与数组 3 merge， cost $2n + 4n = 6n$

数组 4：cost $2n + 6n = 8n$

...

故 k 个数组完全合并，cost

$$\sum_{i=1}^{k} 2i \times n = 2n \times \sum_{i=1}^{k} i = k(k+1)n$$

2):

给出分治算法：

```
def divMerge(k个数组):
    int A1[]=divMerge("前k/2个数组")
    int A2[]=divMerge("后k/2个数组")
    merge(A1,A2)

def merge(A1[],A2[]):
    i=0,j=0
    list A3[]
    while(i<len(A1) and j<len(A2)):
        if(A1[i]<A2[j]):
            A3.append(A1[i])
            i+=1
        else:
```

```
            A3.append(A2[j])
            j+=1
```

对该算法而言：

$$T(k,n) = 2T\left(\frac{k}{2}, n\right) + f(n,k)$$

Where $f(n)$ 是将两个排好序的数组合并的代价，即 merge 两个长度为 kn/2 的数组.

$$f(n,k) = kn$$

由 Master Theorem

$$E = \frac{\log b}{\log c} = 1$$

而

$$n^E = \Theta(f(n)) = \Theta(n)$$

故：

$$T(k,n) = f(n,k)\log k = nk\log k$$

Q.E.D

7.5

给定一个 n 个 nodes 的二叉树：

1. 设计 $O(n)$ 算法计算树的高度
2. 设计 $O(n)$ 算法计算树的直径（树中节点距离的最大值。距离：之间最短路径的长度）
   1) 计算树的高度
   可以分治的来实现。

$$h(T)=\max\{h(\text{left subtree of }T),\ h(\text{right subtree of }T)+1\}$$

```
    由以上递归表达式设计算法：
    ```python
    def calHeight(tree T):
        if(size(T)==1)
            return 1
        if(size(T)==0)
            return 0
        int h1=calHeight(left subtree of T)
        int h2=calHeight(right subtree of T)
        return max(h1,h2)+1
    ```
    由于该算法并不会产生对树的节点的重复遍历，递归在树高度意
义上严格下降，所以其时间复杂度为 $O(n)$
2) 计算树的直径
    树的直径：设为 $D(T)$
```

$$D(T)=D(\text{left subtree of }T)+D(\text{right subtree of }T)+1$$

```
    证明：容易知道直径所对应的路径一定过根结点. 那么可以分别
计算两段的路径长度。由于左子树中的点到右子树的点的所有路径都
```

```
经过**rootnode**，那么该递归式成立。
```

```
def diameterCal(tree T):
    if(T only have one node):
        return 1
    if(T have no node):
        return 0
    int d1=diameterCal(left subtree of T)
    int d2=diameterCal(right subtree of T)
    return d1+d2+1
```

这样，每个点仍然只被遍历了一次，所以 $O(n)$ 时间内可以完成。//这实际上是一个 BFS?

7.8 (find maxima)

1. 设计算法找出 maxima
    1. 如果能对点排序：

```
def find_maxima(points):
    # 按 x 升序排列，x 相同时按 y 降序排列
    # 这样我们可以只关注y，因为x已经排好。
    sort points firtly increase x, secondly decrease y
    maxima = []
    current_max_y = -inf

    # 从右到左扫描
    for x, y in reversed(points):
        if y > current_max_y:
```

```
            maxima.append((x, y))
            current_max_y = y
    return sorted(maxima)
```

假如不允许排序：
我们可以不断地对 x 坐标二分，在合并时更新当前的 maxima

```
#划分最小窗口大小
int xmin=min{x_{i+1}-x_{i}}
#D是窗口宽度
def find_maxima_divide(points,start,end):
    D=end-start
    if(D<xmin):#最多只有一个点，自己是这部分的maxima
        return points
    else:
        points1=find_maxima_divide(points,start,D/2)
        points2=find_maxima_divide(points,D/2,end)
        #合并两部分的maxima
        return merge(points1, points2)

def merge(points1,points2)
    #已知points1中的x坐标都更小
    maxima=[]
    i=start of points1
    yMax=points2中的y坐标最大值
    #右边的点集，都比左边大，所以左边点集不影响它们仍然是maxima
    maxima.append(points2)
    while(i<len(points1)):
        if(i.y>yMax):
            maxima.append(i)
```

```
            i+=1
    return maxima
```

2. 整理算法
   这两种思路等价，我们按第二个来
   对于这个 O (n) 的算法整理如下：

```
def find_maxima(points,x1,x2,y1,y2):
    int xmid=sum of x in points DIV len(points)
    int ymid=sum of y in points DIV len(points)
    #分别计算第一  第二  第三  第四象限
    points_1=find_maxima(points,xmid,x2,ymid,y2)
    points_2=find_maxima(points,x1,xmid,ymid,y2)
    #第三象限不需计算
    points_4-find_maxima(points,xmid,x2,y1,ymid)
    yMax=max y in points_1
    xMax=max x in points_2
    #类似第一题,进行merge
    maxima.append(points_1)
    forall point in points_2:
        if(point.y>yMax):
            maxima.append(point)

    forall point in points_4:
        if(point.x>xMax):
            maxima.append(point)
```

错误：

本算法假设：第三象限全部舍弃，第一象限找到的 maxima

candidate 全部都是 maxima，但是我们可以基于对抗策略（adverse tragedy)来构造一种最坏的输入：

- 第三象限全部舍弃：算法按 x, y 二分，我们可以让第三象限一个点都没有，这样舍弃的优化相当于白做——也就是算法并不一定可以做到其递归表达式理想的平均分
- 这其实涉及到一个逻辑：尽管我们保证 x 轴上下各有 $\frac{n}{2}$ 个点，y 轴左右各有 $\frac{n}{2}$ 各点，但是不能保证他们具体在每个象限的分布。

一旦划分足够不平均，递归表达式可能成为 $T(n)=2T\left(\frac{n}{2}\right)+O(n)$ 由master theorm知道其时间复杂度为 $O(n\log n)$

3. 证明算法时间复杂度下界
   有 n 个点，要找出其中的 maxima，我们必须知道其和所有点的 x，y 的大小关系。答案空间（所有点 x，y 的排列）为 $n!\times n!$

每次比较，仅能 2 分答案空间

$$2^k>n!\times n!$$

且有斯特林公式

$$n!\approx\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

那么

$$k > \log(n! \times n!) = o(n \log n)$$

7.14 (缺失的比特串)

Ques:
给定一个二维比特数组，有k行n列，存放了所有可能的k比特串，仅仅有某一个k比特串被剔除，所以k和n满足$n = 2^k - 1$ 现在需要计算出缺失的比特串，关键操作是“检察数组的某一位是0或1”
Solution：

1. 我们不难发现：对于比特串的每一位而言（即每一行），其应该有 $n/2$ 个 0，$n/2$ 个 1. 所以我们按行按列二维遍历，检察缺失的 0 或 1，在 $O(nk)$ 时间内可以做完. 算法如下

```
def findBits(bits[][]):
    for i in row of bits:
        #to store the amt of overcount 1
        cnt=0
        for j in col of bits in row i:
            if(bits[i][j]==1):
                cnt+=1
            else:
                cnt-=1
        #when j loop end, if cnt==1, the i th bit of
ans-bits is 1
        #if cnt==-1(there' s only two case.),the i th
bit of ans-bits is 0
        if(cnt==1):
            ans_bits.append(1)
        else:
```

```python
            ans_bits.append(0)
    return ans_bits
```

2. 2025-03-16: 我不太理解如果能不检查某一位的数字，没有获得这个信息，如何确定缺失的比特串，所以我不知道如何在 $O(n)$ 内解决. 可能是我的理解有问题。
如果可以将 bit 串按整体做异或，那可以这样做：
异或操作满足交换律和结合律，而且对于任意数 xxx，有

$$x \oplus x = 0$$
$$x \oplus 0 = x$$

所以我们推导出：将给定的比特串全部做异或，再与 0 做异或. 可以得到结果。
按列遍历，将每一列的比特串做异或. 最后得到的结果与实际的
这个过程只需要按列遍历一遍，所以 $O(n)$

Chapter 14

14.1

请证明：对于所有整数 $h \geq 1$，$\left\lceil \log\left(\left\lfloor \frac{1}{2}h \right\rfloor + 1\right) \right\rceil + 1 = \lceil \log(h+1) \rceil$（结合堆结构）
Proof:
将 h 划分进 $[2^{k-1},\ 2^k - 1]$ 的区间方便分析.
观察得到：当 k=1 时，即 h=1 时，显然成立
假设 $k \leq n$ 时均成立，证明 $k = n + 1$ 也成立：
$k = n + 1$, then $h \in [2^n, 2^{n+1} - 1]$

$$\log\left(\left\lfloor\frac{1}{2}h\right\rfloor+1\right)\leq\log(2^n-1+1)=n$$

$$\log\left(\left\lfloor\frac{1}{2}h\right\rfloor+1\right)\geq\log(2^{n-1}+1)>n-1$$

$$\text{so we have:}$$

$$LHS=\left\lceil\log\left(\left\lfloor\frac{1}{2}h\right\rfloor+1\right)\right\rceil+1=n+1$$

$$\log(h+1)\leq\log(2^{n+1})=n+1$$
$$\log(h+1)\geq\log(2^n+1)>n$$
$$\text{so we have:}$$
$$\lceil\log(h+1)\rceil=n+1$$
$$LHS=RHS$$
$$Q.E.D$$

用堆的结构特性来解读：
堆的结构：是一颗完美二叉树或者仅仅比一棵完美二叉树少若干个节点且节点紧密排列。在这个意义上解读：
对于一个堆的某一个节点 h ，其 parent node 的下标是 $\left\lfloor\frac{1}{2}h\right\rfloor$，左边表示节点 h 的 Parent node 的下一个节点的 depth，右边表示节点 h 的下一个节点的 depth，二者相等

14.2 堆中第 k 大的元素

给定一个堆，其中有 n 个元素. 请选中其中第 k 大的元素。假设 $k\ll n$, 选择的代价要求是 k 的函数

Solution：
假设是大根堆。我们依次挪去 root，挪 k 次，就找到了第 k 大的元

素。但是此题要求 k 的函数代价来完成，又有 $k \ll n$. 我们需要优化。

经过分析可得，要找到第 k 大的元素，最坏情况我们需要遍历到堆的第 k 层. 而超过堆的第 k 层的话，任意一条 path $x_1, x_2, \ldots, x_{k+1}$ to k+1 th floor, we hold:

$$x_1 < x_2 < x_3 < \cdots < x_{k+1}$$

即第 k+1 层一定没有第 k 大元素.
所以我们截取堆的前 k 层一定可以找到答案。对于数组实现的堆，这一点只需要将堆的规模改为 $2^k - 1$

```
def heapFindKth(heap H[],k):
    H.size=2**k-1
    for(int i=0;i<k;i++):
        H.deleteMax()
        fixHeap()
    return H.getMax()
```

FixHeap in $O(k)$
总体在 $O(k)$ 内做完了.

14.3 d 叉堆

一维数组表示。根节点存放在 $A[1]$

1. 证明正确性:
   假设第 i 位，是堆的第 h 层（h 从 1 开始）中，按 d 个一组的第

k 组中的，第 j 个节点
等比数列求和可以得到前 h 层节点求和个数

$$\sum_{h=1}^{H} \text{amt of nodes} = \frac{1-d^h}{1-d}$$

则：

$$i = \frac{1-d^{h-1}}{1-d} + (k-1)d + j$$

Where $0 < j \le d$
容易知道其 Parent node，是堆的第 $h-1$ 层的，第 k 个节点

$$\text{Parent}(i) = \frac{1-d^{h-2}}{1-d} + k$$

设 $I = \frac{i-2}{d} + 1$

$$\begin{aligned}
I &= \frac{i-2}{d} + 1 \\
&= \frac{\frac{d-d^{h-1}}{1-d} + 1 + (k-1)d + j - 2}{d} + 1 \\
&= \frac{1-d^{h-2}}{1-d} + (k-1) + \frac{j-1}{d} + 1 \\
&= \frac{1-d^{h-2}}{1-d} + k + \frac{j-1}{d}
\end{aligned}$$

而 $0 \le j-1 < d$
故

$$\lfloor I \rfloor = \mathrm{Parent}(i)$$

Q.E.D

2. CHILD

根据 1）中分析，父节点位于第 $h-1$ 层，第 k 个节点

$$i = \frac{1-d^{h-2}}{1-d} + k$$

其第 j 个子女下标表示为：

$$\mathrm{Children}(i) = \frac{1-d^{h-1}}{1-d} + (k-1)d + j$$

$$\begin{aligned} d(i-1)+j+1 = & \\ & = \frac{d-d^{h-1}}{1-d} + kd - d + j + 1 \\ & = \frac{1-d^{h-1}}{1-d} + (k-1)d + j \\ & = \mathrm{Children}(i) \end{aligned}$$

Q.E.D

14.4

过程有点繁琐，用手写

先考虑Heap是完美二叉树

k层，有 2^k-1 个点，第i层有 2^{i-1} 个点，高度为 $k-i$

满足：节点的高度之和

$$S(k)=\sum h=\sum_{i=1}^{k} 2^{i-1}(k-i)$$

$$=\sum_{i=1}^{k} k\cdot 2^{i-1}-\sum_{i=1}^{k} i\,2^{i-1}$$

$$=k\cdot\frac{1(1-2^k)}{1-2}-T(k)$$

$$T(k)=1\times 2^0+2\times 2^1+\cdots+k\cdot 2^{k-1}$$

$$2T(k)=\quad 1\times 2^1+\cdots+(k-1)2^{k-1}+k\cdot 2^k$$

$$-T(k)=2^0+2^1+\cdots+2^{k-1}-k2^k$$

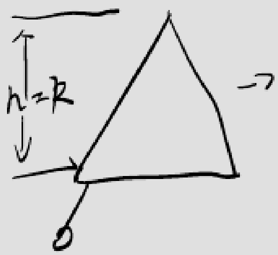
$$=\frac{1(1-2^k)}{1-2}-k2^k=2^k-1-k2^k$$

$$S(k)=k(2^k-1)-T(k)=k2^k-k+2^k-1-k2^k$$

$$=2^k-k-1$$

而总点数为 $n=2^k-1$　　显然有 $S(k)\le n-1=2^k-2$

下面我们证明在完美二叉树上加节点，仍满足。



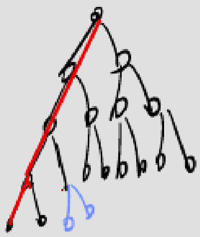
完美二叉树 (k+1层第1个)

case 1: 加的最新的节点: 有k个点高度 +=1,

sum_h += k

node_n += 1

而 $S(k) + k = 2^k - 1$, 仍 $\le n-1 = 2^k - 2 + 1 = 2^k - 1$



case 2: 以 n_R (Right Children 加入).

其路径上所有点仍已更新, 即

sum_h += 0

node_n += 1.

一旦新一层 n_R 已加入, 堆有 sum h = n-1, 且标红点 h 在 k+1 层不再更新. 可以递归地划分小规模的堆. 而 $n \le 3$ 时, I. Base 显然成立

则不难证明命题成立.

14.5

请给出一个时间为 $O(n \log k)$，用来将 k 个已排序链表合成一个有序链表的算法。这里 n 表示所有输入链表种元素的总数

使用分治法:

```
def list_merge(lists [])
    if size of left half lists <= 1 || size of right half lists<=1:
        #only one list, merge with other
        merge(left half lists,right half lists)
    left half lists=lists[:k//2]
```

```
    right half lists=lists[k//2:]
```

merge和归并排序中的merge实现相似，时间复杂度为 $O(n)$
$T(k) = 2T\left(\frac{k}{2}\right) + O(n)$
所以时间为 $O(n\log k)$

14.6 动态发现中值

用两个堆，同时尽量维护他们的高度相近（平衡的） 这两个堆一个是大根堆，一个是小根堆。
描述 ADT

```
class dynamicMidFind:
    heapLarge hL
    heapSmall hS

INSERT(K):
    if(hL is empty):
        hL.insert(K)

    #to insert in upper large heap or lower small heap?
    if(k<top(hL)):
        hL.insert(K)
    else:
        hS.insert(K)

    #维护size相差不超过1
    if(size(hL)>size(hS)):
        int maxTop=top(hL)
```

```
        hL.deleteMax
        hS.insert(maxTop)
    else:
        int minTop=top(hS)
        hS.deleteMax
        hL.insert(maxTop)

findMid(K)
    if(hL size == hS size):
        #return average of each root
        return (top(hL)+top(hS))/2

    if(hL size -hS size ==1):
        return top(hL)

    #only 3 cases
    return top(hS)
```