# 一、实验题目与要求

1．Implement exercise 2.3-7.

2．Implement priority queue.

3． Implement Quicksort and answer the following questions. (1) How many comparisons will Quicksort do on a list of n elements that all have the same value? (2) What are the maximum and minimum number of comparisons will Quicksort do on a list of n elements, give an instance for maximum and minimum case respectively.

4． Give a divide and conquer algorithm for the following problem: you are given two sorted lists of size m and n, and are allowed unit time access to the ith element of each list. Give an O(lg m + lgn) time algorithm for computing the kth largest element in the union of the two lists. (For simplicity, you can assume that the elements of the two lists are distinct).

# 二、算法思想

1. 两数之和

最快捷的算法就是先将数列排序，然后用双指针算法，在数列一头一尾设置两个指针，不断向中间汇聚直到相遇，期间根据和的情况选择头指针向右移还是尾指针向左移，或者输出结果并同时往中间移。

1. 优先队列

堆是一种完全二叉树结构，对于大顶堆，根节点的值将大于等于子节点的值。实践中往往 不建立完全二叉树结构，而是以连续元素序列作为树结构，用下标模拟父子关系。其中，对于 索引以 0 起始的元素序列，下标为 *n* 的元素的左右子节点的下标分别为 2*n* + 1 和 2*n* + 2。

为了实现堆排序和优先级队列，需要实现几个堆操作算法：

·add：在队列尾部加入新元素；

·sift\_up：将序列尾部的元素向上移动到正确的位置（即重新调整为小顶堆）；

·sift\_down：将序列首部的元素向下移动到正确的位置；

·extract：删除队列首部的元素。

堆调整算法实现为sift\_up和sfit\_down。该算法不断地比较父节点和子节点的元素值大小关系： 如果父节点的值小于任一子节点的值，则将最大的节点值调整为父节点，原父节点下沉为子节点，并继续将其作为新的子树的根节点做堆调整处理。

Extract函数通过先记录队首元素，然后交换其与队尾元素（此时相当于已经将队首元素删除），然后将此时的队首元素（即原队尾元素）用sift\_dowm下移，最后返回记录的原队首元素。

1. 快速排序

快速排序算法通过选取一个基准元素，将所有比基准元素小的元素移动到基准元素左侧， 比基准元素大的元素移动到右侧，随后继续对左右两端序列递归求解，实现时间复杂度为Θ(*n* lg *n*) 的排序。然而，若基准元素选取不佳（如选择到了最小或最大元素），则算法的时间复杂度将退化为 Θ(*n*2)，因为经过划分后，一段序列长为 *n* − 1，另一段序列为空。

随机快速排序每次随机选择基准元素，则降低了选择到不好的基准元素（最大或最小元 素）的可能性，从而减少算法时间复杂度退化的可能。

然而，若序列所有元素均一致，随机快速排序算法的时间复杂度依然为 Θ(*n*2)。后文将描述解决此种情况的优化算法，同时描述另一种选取基准元素的方法，以进一步减少选择最差基 准元素的概率。

1. 两个有序数列的第k小值

可以通过二分法将问题规模缩小，假设p + q = k，A[p -1]对应A中第p个元素，总序列是由A，B组成的有序序列 。

1.如果序列A中的第p个元素小于序列B中的第q个元素，则序列A的前p个元素肯定都小于总序列的第k个元素，即序列A中的前p个元素肯定不是总序列的第k个元素，所以将序列A的前p个元素全部抛弃，形成一个较短的新序列；然后，用新序列替代原先的A序列，再找其中的第k - p个元素（因为已经排除了p个元素，k需要更新为k-p），依次递归；

2.同理，如果A序列中的第p个元素大于序列B中的第q个元素，则抛弃序列B的前q个元素，k = k - q；

3.如果序列A的第p个元素等于序列B的第q个元素，则第k个元素为A(p-1) ；

递归终止条件:

1. 如果一个序列为空，那么第k个元素就是另一个序列的第k个元素；
2. 如果k = 1，那么直接返回min(A[0],B[0]) ；
3. 如果A[p - 1] == B[q - 1]，则第k个数就为A[p - 1]；

# 三、算法步骤与核心代码

1. 两数之和

按照上文的具体分析，代码如下:

|  |
| --- |
| **while** i**<**j**:**  **if** lst**[**i**]+**lst**[**j**]==**t**:**  **if not** f**:print(**"满足要求的元素组为："**)**  f**=1**  **print(**lst**[**i**],**lst**[**j**])**  i**+=1**  j**-=1**  **elif** lst**[**i**]+**lst**[**j**]<**t**:**i**+=1**  **else:**j**-=1**  **if not** f**:print(**"不存在满足要求的元素组"**)** |

1. 优先队列

按照上文所述，堆算法的核心是调整堆算法 \_adjust\_heap 的实现。所有堆算法都实现于

优先队列.py。

add函数的核心代码如下：

|  |
| --- |
| def add**(**self**,**value**):**  self**.**heap**.**append**(**value**)**  self**.\_**siftup**(len(**self**.**heap**)-1)** |
|  |

其中的sift\_up函数利用递归思想，核心代码为

|  |
| --- |
| def\_siftup**(**self**,**index**):**  if index**>0:**  parent**=int((**index**-1)/2)**  if self**.**heap**[**parent**]>**self**.**heap**[**index**]:**  self**.**heap**[**parent**],**self**.**heap**[**index**]=**self**.**heap**[**index**],**self**.**heap**[**parent**]**  self**.\_**siftup**(**parent**)** |

extract函数具体步骤上文已讲：

|  |
| --- |
| **def** sift\_down**(**self**,** index**):**  **if** index **< len(**self**.**heap**):**  left **= 2 \*** index **+ 1**  right **= 2 \*** index **+ 2**  **if** left **< len(**self**.**heap**) and** right **< len(**self**.**heap**)** \  **and** self**.**heap**[**left**] <=** self**.**heap**[**right**]** \  **and** self**.**heap**[**left**] <=** self**.**heap**[**index**]:**  self**.**heap**[**left**],** self**.**heap**[**index**] =** self**.**heap**[**index**],** self**.**heap**[**left**]**  self**.\_**siftdown**(**left**)**  **elif** left **< len(**self**.**heap**) and** right **< len(**self**.**heap**)** \  **and** self**.**heap**[**left**] >=** self**.**heap**[**right**]** \  **and** self**.**heap**[**right**] <=** self**.**heap**[**index**]:**  self**.**heap**[**right**],** self**.**heap**[**index**] =** self**.**heap**[**index**],** self**.**heap**[**right**]**  self**.\_**siftdown**(**left**)**  **if** left **< len(**self**.**heap**) and** right **> len(**self**.**heap**)** \  **and** self**.**heap**[**left**] <=** self**.**heap**[**index**]:**  self**.**heap**[**left**],** self**.**heap**[**index**] =** self**.**heap**[**index**],** self**.**heap**[**left**]**  self**.\_**siftdown**(**left**)** |

|  |
| --- |
| def extract**(**self**):**#删除堆顶元素并返回此元素值  if not self**.**heap**:print(**'队列已空！'**)**  value**=**self**.**heap**[0]**  self**.**heap**[0]=**self**.**heap**[len(**self**.**heap**)-1]**  self**.\_**siftdown**(0)**  return value |

1. 随机快速排序

随机快速排序的递归函数用python写可以写出很简洁的代码。该算法实现于快速排序.py 中。

随机化过程即为获取随机数后将其与尾部元素交换，不再赘述。现以尾部元素为基准元素，从首元素开始遍历：若当前元素小于尾部元素，则将其移动到序列的前面（所有小于尾部 元素的元素序列）。该部分迭代的核心代码为

**def** qs**(**array**):**

**if len(**array**) < 2: return** array

t **=** array**[0]**

l **= [**i **for** i **in** array**[1:] if** i **<=** t**]**

r **= [**i **for** i **in** array**[1:] if** i **>** t**]**

**return** qs**(**l**) + [**t**] +** qs**(**r**)**

1. 两个有序数列的第k小值

根据上文的具体步骤，核心的函数见下：

|  |
| --- |
| def find\_kth**(**llst**,** llen**,** rlst**,** rlen**,** k**):**  **if** llen **>** rlen**:**  **return** find\_kth**(**rlst**,** rlen**,** llst**,** llen**,** k**)**  # 长度小的数组已经没有值了,从rlst找到第k大的数  **if not** llen**:**  **return** rlst**[**k**-1]**  # 找到第1 大的数,比较两个列表的第一个元素,返回最小的那个  **if** k **== 1:**  **return min(**llst**[0],** rlst**[0])**  middle **= min(**k **>> 1,** llen**)**  middle\_ex **=** k **-** middle  # 舍弃llst的一部分  **if** llst**[**middle**-1] <** rlst**[**middle\_ex**-1]:**  **return** find\_kth**(**llst**[**middle**:],** llen**-**middle**,** rlst**,** rlen**,** k**-**middle**)**  # 舍弃rlst 的一部分  **elif** llst**[**middle**-1] >** rlst**[**middle\_ex**-1]:**  **return** find\_kth**(**llst**,** llen**,** rlst**[**middle\_ex**:],** rlen**-**middle\_ex**,** k**-**middle\_ex**)**  **else:**  **return** llst**[**middle**-1]** |

# 四、总结

学会了多种排序算法并将其进行了时间复杂度上的比较。第四题在思想上有很大的启发。