Q2. Purpose: Practice solving recurrences. For a-c, use the Master Theorem to derive asymptotic bounds for T(n) or argue why the Master Theorem does not apply. You don't have to solve the recurrence if the MT does not apply. Please assume that small instances need constant time c if not explicitly stated. Justify your answers, i.e., give the values of a, b, n^log_b(a), ? for case 3 of the Master Theorem also show that the regularity condition is satisfied. (3 points each)

a) T(n)=8T(n/2) +n^4+n

$a = 8\ b = 2$ $\log_b a = \log_2 8$ n^ $\log_b a$ = n^ $\log_2 8$ = n^3

$f(n) = n^4$ € $\Omega(n^{3+\mathcal{E}})$

Hence, here the case 3 applies.

$\mathcal{E} = 0.1$

Therefore T(n) € Θ(f(n))

T(n) € Θ (n^4)

Regularity Condition: There is c<1 and n>=0 such that $af(n/b) <= c.f(n)$

8* (($n/2)^4$ + n/2) <=c (n^4+ n)

8*(n^4/16+ n/2)<=c*n^4+n

8*(n^4/16+ n/2)/ n^4+n<=c

As n tends to infinity the dominant terms in numerator and denominator will be n^4.

8*(n^4/16)/n^4<=c

Dividing by n^4 on both sides

½ <=c

But c<1 hence 0.5<=c<1 for n>0

b) T(n)=3T(n/3)+n/lg(n)

a=3 b=3 $\log_b a = \log_3 3$ n^ $\log_b a$ = n^ $\log_3 3$ = n^1

f(n)=n/lg(n)

The driving function is n / lg(n)=o(n) which means that it grows asymptotically more slowly than the watershed function n. But n= lg n grows only logarithmically slower than n, not polynomially slower. So, lg(n) € o($n^{\mathcal{E}}$) so 1/lg(n) € ($n^{-\mathcal{E}}$) and also n/lg(n) € ὼ($n^{1-\mathcal{E}}$)= ὼ($n^{\log_b a-\mathcal{E}}$). Thus there is no constant $\mathcal{E}$>0 exists such that n/lg(n) € ὼ($n^{\log_b a-\mathcal{E}}$) which is required for case 1 to apply. . Case 2 fails to apply as well, since n/lg(n)= $\Theta(n^{\log_b a} \lg^k n)$ also fails in this k=-1 but for case 2 to apply we need k must a non-negative number.

c) T(n)=2T(n/4)+nlg(n)

$a = 2\ b = 4$ $\log_b a = \log_2 4$ n^ $\log_b a$ = n^ $\log_2 4$ = n^2

$f(n) = n\lg n$€ $\Omega(n^{2+\mathcal{E}})$

Hence, here the case 3 applies.

$\mathcal{E} = 0.1$

Therefore T(n) € Θ(f(n))

T(n) € Θ (n^2)

Regularity Condition: There is c<1 and n>=0 such that $af(n/b) <= c.f(n)$

2*((n/4)*lg(n/4)<=cnlgn
2*((n/4)(lg(n)-lg(4)))<=cnlg(n)
Dividing both sides by n
½(lg(n)-2)<=clg(n)
lg(n)(1/2 -c)<=1

As lg(n)> 0 for n>2. c>1/2 for these inequality to be true. But c<1 hence 0.5<=c<1 n>2

d) Assume T(n) = (n+1)T(n-1)/n+c(2n-1)/n for n>1 and T(1) = 0. Show T(n) = c(n+1)$\sum_{i=2}^{n}$ $\frac{2i-1}{i(i+1)}$ by forward iteration.

T(n)=(n+1)T(n-1)/n + c(2n-1)/n

T(n)/n +1 = T(n-1)/ n + c(2n-1)/n(n+1)

T(n)/n+1-T(n-1)/n= c(2n-1)/n(n+1)

T(2)/3-T(1)/2=C(2*2)-1/2(2+1)

T(3)/4-T(2)/3=C(2*4)-1/3(3+1)

T(n)/n+1-T(n-1)/n= c(2n-1)/n(n+1)

Add all the terms. As we can see all the elements on the LHS will get cancelled only T(n)/n+1 and T(1)/2=0 will remain. The RHS will be a sum from i=2 to n

Therefore

T(n)/n+1-0= c∑ (2i-1)/i(i+1) where is i=2 to n

T(n) = c(n+1) ∑ (2i-1)/i(i+1) where is i=2 to n

Q3. Purpose: Practice algorithm design. This problem was an interview question! To avoid deductions, please follow Eric Demaine's instructions about how to "give/describe" an algorithm. Let X[1..n] and Y[1..n] be two arrays, each containing n numbers already in sorted order. Give an O(lgn)-time algorithm to find the median of all 2n elements in arrays X and Y. You may assume that all 2n numbers are distinct, and that n is a power of 2.

1. In these algorithm, the idea is by comparing the medians of both arrays , we reduce the sample space by half recursively.

Pseudo Code:

```
def get_median(arr,n):
  if(n%2==0):
    return ((arr[n//2-1] +arr[n//2])/2)
  else:
    return arr[n//2]
def median(x,y):
  print(x,y)
  n=len(x)
  if(n==1):
    return (x[0]+y[0])/2
  x_middle = get_median(x,n)

  y_middle = get_median(y,n)
  print(x,y,x_middle,y_middle)

  if(x_middle==y_middle):
    return x_middle
  if(x_middle<y_middle):
    return median(x[n//2:], y[:(n//2)])
  else:
    return median(x[:(n//2)], y[(n//2):])
```

2. First we find of the medians of both arrays which can be done in O(1) time. Let's assign them m1 and m2

   CASE 1: if m1 is equal to m2.

   It means that there are n-1 elements greater than m1 and m2 and n-1 elements greater than m2. Which means m1 and m2 will lie in the middle of merged array. Hence we return m1 or m2 as both of them are equal.

   CASE 2: if m1<m2.

   In the merged array the median array will lie between m1<=median of merged array <=m2. So, we ignore the elements to left side of m1 in x because all the elements less than m1 in x will lie in the in range 0 to n-2 and never in the middle. Also, we ignore the elements to right side of m2 in y, because all the elements greater than m2 will lie in the rang of n-2 to 2n in the merged array and not in the middle.

   CASE 3: if m1>m2.

   In the merged array the median of the merged array will lie between m2<=median of merged

Array<=m1.So we ignore the elements to right side of m1 in x because all the elements more than m1 in x will lie in the range of n-2 to 2n in merged array and never in the middle. We will ignore the elements to left side of m2 in y because all the elements less than m2 in y will lie in the range of 0 to n-2 and never in the middle.

3. Worked Out Example:
   X=[5,6,7,8]
   Y=[1,2,3,4]

   Step1: Let m1 be the median of x and m2 be the median of y. m1=6.5 and m2=2.5. As m1>m2 so case 3 will apply here. The median will lie between 2.5<= median of merged array<=6.5

   Step2: So, will discard the elements greater than m1 in x. hence x will be x= [5,6]. We will discard the elements less than m2=2.5. hence y= [3,4].

   Step 3: So, the m1 now will be m1=5.5 and m2 will be m2=3.5. So m1>m2 so case 3 will apply Here in x, we will discard 6 as it is larger than m1=5.5 and in y we will discard 3 as 3 is smaller than 3.5. therefore x= [5] and y= [4].

   Step 4: As n=1, the best case will execute. We will return the average of 5 and 4. Hence the median of merged array will be 4.5.
4. Correctness: The loop invariant here is that the correct median must always be between the partitioned elements of X and Y. We maintain this invariant by adjusting the partition positions based on the comparisons of the maximum and minimum elements around the partition points.
5. Time Complexity:
   The recursion divides the problem in the size 2. The sub problems computes one sub problem per recursive call. Hence the recurrence relation for the above problem would be

   $T(n) = \{c_0 \text{ if } n < s$
   $\{a*T(n/b) + D(n) + C(n) \text{ otherwise}$

   a=1……no of sub problems in per recursive call.
   b=2 ……size of sub problem
   D(n)=1 …….. time required for deciding which sub problem to solve (checking if m1ix greater or m2)
   C(n)=0 ……not combining solution.
   G(n)=2*(1+1) …….. time required for finding the median of two arrays in every recursion. 1 is for deciding whether n is even or odd and other for calculating the median.

   $T(2n) = \{1 \quad \text{if } n==1\}$
   $\{1* T(2n/2) +1+4 \text{ otherwise}\}$
   We have got 2n because our sample space is 2 arrays of size n.

   $T(x) = \{1 \quad \text{if } n==1\}$
   $\{1*T(x/2) +1+4 \text{ otherwise}\}$
   Where x=2n

   By using master theorem, we will find big-theta of the code

$$T(n) = \begin{cases} c_0 & \text{if } n < s \\ aT(n/b) + f(n) & \text{otherwise} \end{cases}$$

Master Theorem

$f(n) \in O(n^{\log_b a - \epsilon})$ (for some $\epsilon > 0$)	$T(n) \in \Theta(n^{log_b a})$
$f(n) \in \Theta(n^{\log_b a})$	$T(n) \in \Theta(n^{log_b a} \lg n)$
(for some $\epsilon > 0$) $f(n) \in \Omega(n^{\log_b a + \epsilon})$ and (*)	$T(n) \in \Theta(f(n))$

Our $\log_b a = \log_1 1 = 0$ and $n^{\log_b a} = n_0 = 1$ and $f(n)=4$. The master theorem case 2 applies here.
So, $T(n)=\Theta(n_0 * \lg n)$
$T(n)=\Theta(\lg n)$

Q4. (9 points) Purpose: Practice algorithm design and the use of data structures. This problem was an interview question! To avoid deductions, please follow Eric Demaine's instructions about how to "give/describe" an algorithm. Consider a situation where your data is almost sorted—for example, you are receiving time-stamped stock quotes, and earlier quotes may arrive after later quotes because of differences in server loads and network traffic routes. Focus only on the time stamps. Assume that each time stamp is an integer, all time stamps are different, and for any two time stamps, the earlier time stamp corresponds to a smaller integer than the later time stamp. The time stamps arrive in a stream that is too large to be kept in memory. The time stamps in the stream are not in their correct order, but you know that every time stamp (integer) in the stream is at most a hundred positions away from its correctly sorted position. Design an algorithm that outputs the time stamps in the correct order and uses only a constant amount of storage, i.e., the memory used should be independent of the number of time stamps processed. Solve the problem using a heap.

Description:

We know the 1st element is at most 100 position away for its correct position. Hence at its worse, it will be 101th position away for its correct position. Therefore, we can sort this timestamp using a min-heap of size k+1 where k =100.

We will first add first 101 elements of the stream in the min-heap. Then we will pop out the element (smallest element). And the next timestamp in stream.

We will continue these until all the timestamps in the stream are over. And then will also pop the remaining 101 elements one by one and add in the stream.

Pseudo Code:

```
1.SORT_TIME_STAMP (stream, k=100):
2. i = 0
3. kMinHEAP
4.
5. WHILE i < MIN (k+1, length(stream)):
6.    addNode(kMinHEAP, stream[i]))
7.
8. sortedindex = 0
9.
10. for i in range(k+1, len(stream):
11.   minElem = ExtractRoot(kMinHeap)
12.   stream[sortedindex] = minElem
13.   sortedindex += 1
14.   addNode(kMinHeap, stream[idx])
15.
16. WHILE length(kMinHeap) != 0:
17.   minElem = ExtractRoot(kMinHeap)
18.   stream[sortedindex] = minElem
19.   sortedindex += 1
20.
21. return stream
```

Worked Out Example: We assume a stream=[3,5,1,2] and will assume that k=2 which means the value at the stream is at most two positions away from it.

Stream = [3,5,1,2]

Steps:

1. We will construct a min-heap of k+1 elements in these case k+1=3.
2. Hence, we construct a heap of 3 elements.
3. Initialize i=k+1 and sortedindex=0
4. In this heap will be at the root. And it will have 3 and 5 as their children.
5. Then, I extract 1 from root and add it to the stream at sortedindex=0.
6. And add 2 to heap and increment sortedindex by 1.
7. Which again get placed at root because it is the smallest value.
8. Then, as I again extract min from heap which is 2 and add it to stream[sortedindex] and again increment the sortedindex by 1
9. As elements in the stream are over, we will extract all the elements from heap until heap is null.
10. The sorted stream will be stream= [1,2,3,5]

Correctness:

Loop Invariant: The loop invariant property is that the elements in the stream[0: sortedindex] will contain “sortedindex” number of smallest elements of stream in a sorted manner.

Initialisation: In the start the sortedindex is 0, so the sub array contains 0 smallest element hence the loop invariant property holds true.

Maintenance: During each iteration of loop a timestamp is added to the heap. When the size of min heap gets more than 101, we extract the minimum element from the min heap and assign it to its correct position. The correct position this case will stream[sortedindex]. The sorted index is incremented by 1 in every iteration. The loop invariant property is maintained in each iteration, as after each iteration we get subarray stream[0:sortedindex] has smallest “sortedindex” in a sorted order.

Termination: The loop terminates when all the timestamps from the stream have been processed. After this also the last k+1 elements are remaining in the min heap. So, the algorithm keeps on popping the element for min heap until it is empty again. This ensures that all the elements in the stream are in their correct positions. Hence in the end stream will contain all the elements in the sorted order.

<u>Time Complexity:</u>

```
1.SORT_TIME_STAMP (stream, k=100):
2. i = 0                                                    1
3. kMinHEAP
4.
5. WHILE i < MIN (k+1, length(stream)):
6.    addNode(kMinHEAP, stream[i]))                          k+1*log(k)
7.
8. sortedindex = 0                                          1
9.
10. for i in range(k+1, len(stream):                        n-(k+1)
11.   minElem = ExtractRoot(kMinHeap)                       (n-(k+1)*logk
12.   stream[sortedindex] = minElem
```

```
13.   sortedindex += 1
14.   addNode(kMinHeap, stream[idx])                    (n-(k+1)*logk
15.
16. WHILE length(kMinHeap) != 0:                        K+1*logk
17.   minElem = ExtractRoot(kMinHeap)
18.   stream[sortedindex] = minElem
19.   sortedindex += 1
20.
21. return stream
```

T(n)€O(nlogk) where k=100

T(n)€ O(n)

Space Complexity:

Since we use only extra space of min heap which is fixed size of size k irrespective of size stream.

S(n) = O(k)
since k = 100,
S(n) € O(1)