1: recurrence best O(.) bound

(a) it has 4^k copies of $\frac{n}{4^k} + 4^{k-1}$ terms of $\frac{n}{4^{k-1}}$ at kth level

=> $4^k T(\frac{n}{4^k}) + 4^{k-1} \sum_{i=1}^{k} \frac{n}{4^{i-1}}$ where $k = \log_4 n$

=> $T(1).4^k + \sum_{i=1}^{k} 4^{i-1}.\frac{n}{4^{i-1}}$

=> resubstituting k value

=> $4^{\log_4 n} + \sum_{i=1}^{\log_4 n} n$

=> $n + nlogn$

=> $O(nlogn)$

(b) based on the explanation from part (a) of the question, it has 4^k copies of $n/4^k + 4^{k-1}$ terms of 1 at kth level

=> $4^k T(\frac{n}{4^k}) + 4^{k-1} \sum_{i=1}^{k} 1$ where $k = \log_4 n$

=> $T(1).4^k + \sum_{i=1}^{k} 4^{i-1}$

=> resubstituting k value - second factor is a geometric progression

=> $4^{\log_4 n} + 4^{\log_4 n} n$

=> $n + n$

=> $O(n)$

(c) this is a case of linear recurrence

=> $T(n-2) + n + n$

=> $T(n-3) + n + n + n$

at kth level we will have => $T(n-k) + kn$

substituting k=n

=> $T(0) + n^2$

=> $O(n^2)$

(d) the first expression $\frac{n}{3}$ in a recursion tree lasts shorter than $\frac{n}{2}$. the worst case complexity of $\frac{n}{3}$ can be approximated to be equal to $\frac{n}{2}$ if not more than that.

hence the expression reduces to -

=> $T(\frac{n}{2}) + T(\frac{n}{2}) + \sqrt{n}$

=> $2T(\frac{n}{2}) + \sqrt{n}$

effectively we have 2^k terms of $\frac{n}{2^k}$ + 2^{k-1} terms of $\frac{\sqrt{n}}{2^{\frac{k-1}{2}}}$

=> $2^k T(\frac{n}{2^k}) + 2^{k-1} \sum_{i=1}^{k} \frac{\sqrt{n}}{2^{\frac{i-1}{2}}}$ where $k = \log_2 n$

=> $2^{logn} T(\frac{n}{2^{logn}}) + \sqrt{n} \sum_{i=1}^{k} \frac{2^{i-1}}{(2^{i-1})^{\frac{1}{2}}}$

=> $O(n) + \sqrt{n} \sum_{i=1}^{logn} (2^{i-1})^{\frac{1}{2}}$

=> $O(n) + \sqrt{n}(2^{logn})^{\frac{1}{2}}$

=> $O(n) + \sqrt{n}\sqrt{n}$

=> $O(n) + O(n)$

=> $O(n)$

(e) $T(n) = T(\sqrt{n}) + 4$.

number of stages $k = logn$

$n = 2^k$

first expression in the equation cannot have a complexity which is worse than 2. hence equating the value to 2.

=> $n^{\frac{1}{2^k}} = 2$

=> $\frac{1}{2^k} logn = log2$

=> $logn = 2^k$

=> $n = 2^{2^k}$

=> solving the above equation for k also gives us - $k = log_2 log_2 n$

=> recalculating the T(n) function now with the substitution $T(n^{1/2}) = (T(n^{1/4}) + 4) + 4$

effectively we have k terms of $T(n^{\frac{1}{2^k}})$ + k terms of constant 4

at k^{th} stage => $= T(2^{2^0}) + k * 4$.

=> $= T(2) + k * 4$.

=> $= 2 + k * 4$.

Thus the complexity turns out to be O(k)

Since $n = 2^{2^k}$ and k = $log_2 log_2 n$,

complexity = O($log_2 log_2 n$)

(f)

a)

$g(n) = n^2$

$T(n) = 3T(\frac{n}{2}) + n^2$

Total layers = $\log n$

At layer k, we have 3^k times $(\frac{n}{2^k})$ and 3^{k-1} times $(\frac{n^2}{4^{k-1}})$ Let k $= \log n$

$T(n) = 3^k + \sum_{i=1}^{k} \frac{3^{i-1}.n^2}{4^{i-1}}$ which is a Geometric series.

$=> \sum_{i=1}^{k} \frac{3^{i-1}.n^2}{4^{i-1}}$ can be written as $n^2 \sum_{i=1}^{k} \frac{3^{i-1}}{4^{i-1}}$

$=> n^2 \sum_{i=1}^{k} \frac{3^{i-1}}{4^{i-1}} = n^2.(1)$ since $\frac{3}{4} < 1$

Re-substituting the value of $k = \log n$, we get

Time Complexity $= 3^{\log n} + n^2 = O(n^2)$

b)

$g(n) = n$

$T(n) = 3T(n/2) + n$

$T(n/2) = 3(3T(n/2^2) + (n/2)) + n = 3^2T(n/2^2) + 3(n/2) + n.$

After k steps ....

$= 3^kT(n/2^k) + n(1 + 3/2 + 9/4 + ...).$

If k is the last layer then $n/2^k$ is approximately $= 1$

number of layers $k = log_2 n$

Here the ratio of geometric series is 3/2 and if the ratio in a geometric series is greater than 1 then complexity of the series is approximately $ratio^k = (3/2)^{log_2 n}$

Replacing value of k and the geometric series in the step k equation ...

$=> 3^{log_2 n}T(1) + n * (3/2)^{log_2 n}.$

$=> n^{log_2 3}T(1) + n * (n)^{log_2 (3/2)}.$

$=> n^{1.58}T(1) + (n)^{1.58}.$

Thus the complexity turns out to be O($n^{1.58}$)

c)

$g(n) = n^{\log_2 3}$

$T(n) = 3T(n/2) + n^{\log_2 3}$

$T(n/2) = 3(3T(n/2^2) + (n/2)^{\log_2 3}) + n^{\log_2 3}$

$=> 3^2T(n/2^2) + 3(n/2)^{\log_2 3} + n^{\log_2 3}.$

After k steps .... $=> 3^kT(n/2^k) + n^{\log_2 3}(1 + 3/(2)^{log_2 3} + 3^2/(2^2)^{log_2 3} + ...).$

$=> 3^kT(n/2^k) + n^{\log_2 3}(1 + 3/3 + 3^2/(3^2) + ...).$

$=> 3^kT(n/2^k) + n^{\log_2 3}(k).$

If k is the last layer then $n/2^k$ is approximately $= 1$

number of layers $k = log_2 n$

Replacing value of k and the geometric series in the step k equation ...

$=> 3^{log_2 n}T(1) + n^{\log_2 3} * log_2 n.$

Thus the complexity turns out to be O$(n^{n^{1.58}} * log_2 n)$

2: Sorting Nearby Numbers

Algorithm: : Create a new array $B[0 \ldots M]$ where $M = max_i A[i] - min_i A[i]$, with all elements set to the value 0.
- For each element $A[i]$ in the input list, increment the count of $B[A[i] - min(A[i])]$.
- Create a new array C for storing the sorted numbers.
- For each element in the array B, if $B[i]$ is not NULL, append $B[i + min(A[i])]$ to the array C, $B[i]$ times

Correctness: : For each element in array A, we are increasing the count of the respective index in array B. This ensures that all the elements in the array A have unique element count entry in array B. Then you read the array B from index 0 and add the the relative index to an array C "count" number of times. Since we are indexing from lowest to highest value adding entries as they occur, we are sure that the list is sorted. Since we are inserting the values "count" number of times from array B, we are ensuring that all entries are present as well.

Running time: The runtime for searching the min and max element in the array is $O(n)$ as we have to traverse n elements in the given array. The runtime for indexing through the input array A to insert count in array B is O(n) The runtime for indexing through array B to append values in array C is (M) Therefore, total time complexity of this algorithm is $O(2n + M) = O(n + M)$

3: selecting k^{th} smallest element in union of two sorted arrays

from the given data - first sorted array = A[ ] and second sorted array = B[ ].

1->find $(\frac{k}{2} - 1)^{th}$ index in arrays A[ ] and B[ ]

2->set $A(\frac{k}{2} - 1)$ as index i and $B(\frac{k}{2} - 1)$ as index j

3->check for comparison conditions as given below

a-> if $A[i] > B[j]$

this says that all elements to the left of A[i] and right of B[j] are included in our interested set and rest of array space can be ignored. create two new arrays for A[ ] and B[ ] with these interested elements respectively from each sets. the new k value will be $k = k - \frac{k}{2}$

b-> if $A[i] < B[j]$

this says that all elements to the right of A[i] and left of B[j] are included in our interested set and rest of array space can be ignored. create two new arrays for A[ ] and B[ ] with these interested elements respectively from each sets. the new k value will be $k = k - \frac{k}{2}$

4->now check if any of arrays are empty, if that is the case the new k^{th} element of the other array will give us the overall k^{th} element of union of sorted arrays.

a-> if sizeof(A[]) == 0, then B[k-1] = smallest element of union Array; break;

b-> if sizeof(B[]) == 0, then A[k-1] = smallest element of union Array; break;

5->if bullet 4 was not satisfied, then we need to find new field of interested space among both new arrays A[ ] and B[ ]. find i and j such that

a-> In new A[ ] i= minimumOf($\frac{k}{2}$, sizeOf(A[]))

b-> In new B[ ] j= minimumOf($\frac{k}{2}$, sizeOf(B[]))

6->now jump to step 3 and check for A[i] v/s B[j] comparisons and run this recurrence upto step 5 until you find the kth smallest element.

Correctness: : This algorithm considers that the input arrays are sorted. The k^{th} value being asked should be less than the total size of union of sorted input arrays. unless the value of k is reduced to 1 or until one of the arrays reduces to a null set, the k^{th} smallest element of Union will be found in one of the recursively divided arrays. So it is guaranteed to return the k^{th} smallest element but this algorithm will only work for input arrays which are pre-sorted.

Running time: : complexity of creating recursive arrays from $A[]$ is $O(\log n)$, since we are dividing the input array in k/2 parts. complexity of creating recursive arrays from $B[]$ is $O(\log n)$, since we are dividing the array in k/2 parts. Complexity of comparing the two elements is a constant. So, the total Time Complexity = $O(\log n + \log n) = O(2\log n) = O(\log n)$

4: Closest pair of restaurants in Manhattan

(a) If the closest distance happens to be between two points such that one point is present in L part of the grid and the other is present in R part of the grid, then that distance will not be taken into account if we skip algorithm Steps 3-4. In such a case, the distance presumed initially will be wrong.

(b) In the grid of size dxd, we can prove that there exists a minimum distance between two points and that proves that our algorithm extracts that minimum distance.

Lemma : There are 2 points (i,j) in the grid of size dxd that is less than d.

Proof : We can assume that no other points can have same co-ordinates as (i,j) since the grid paths are rectangular. If i and j exist on the same rectangle of the grid, then it is given that we have the minimum possible distance. If i and j exist on 2 different rectangles then by the rectangular nature of grid we can assume that there is some other point k between them which connects them. So the distance between k and j may be less than i and j or the distance between k and i may be less than i and j. Thus we have a pair of points with smaller distance than the distance between i and j.

Now if we consider that i and j existed on the two corners of the grid then by above proof we can infer that there exist some other point k between them which has $distance <= d$ either with i or j. Hence the lemma is proved.

By proof of above lemma, we can say that a minimum distance exists in the grid which is $<= d$, so our algorithm is guaranteed to find a $distance <= d$.

: discussed with Sagar/Madhur

(c)

Algorithm: For given set of points, place them into two sub-arrays where the sub-arrays represent set of points of points on either side of $x - axis$ (such that $x - axis$ is a vertical line that divides the set of points equally). Next, find the closest pair of points in each sub-array and take the lesser of the two as $distancec$. Now, there might be pair of points where each point lies in region L and R grids respectively. So, find the smallest distance between such pairs by considering the region bounded by $x - axis + distance \; and \; x - axis - distance$ and let $distance'$ be this distance. Finally, to find the pair of points having the least distance overall in the given input set, compare $distance \; and \; distance'$ and return the smaller of the two.

Running time: Finding the least distance in each of the two sub-arrays (or regions on either side of x) would take a total of $2T(n/2)$ time and the step where we need to sort the points in the region bounded by $x - axis + distance \; and \; x - axis - distance$ will take $O(nlogn)$ time. Thus, we can say that the run time satisfies the recurrence $T(n) = 2T(n/2) + O(nlogn)$

proof:

$T(n) = 2T(n/2) + nlogn$

$=> 2[2T(n/4) + n/2(log(n/2))] + nlogn$

$=> 4T(n/4) + nlog(n/2) + nlogn$

$=> 4[2T(n/8) + n/4(log(n/4))] + nlog(n/2) + nlogn$

$=> 8T(n/8) + nlog(n/4) + nlog(n/2) + nlogn$

$=> 2^K T(n/2^K) + \sum_{k=1}^{logn} nlog(n/2^{K-1})$

$=> \; n/2^K = 1$

$=> \; n = 2^K$

$=> \; log_2 n = K$

substituting K, we get:

$=> \; n + nlog^2 n$

$=> \; O(nlog^2 n)$

5: Linear time median

(a)

For a given array A of size n and integer this is how much time we take accoring to the given procedure

dividing input into $(\frac{n}{5})$ groups of 5 takes o(n)

array of each groups meadian takes o(n)

selecting a median out of all the sets takes $t(\frac{n}{5})$

finding 3rd element in every array takes constant time o(1)

total complexity is given as $T(n) = T(\frac{n}{5})_{median} + O(n)$

(b)

the complexity needs to be proved using deterministic select.

given that M is a median of sorted elements from group sets of 5

1. We can compare each n-1 numbers with the median and find two sets A and B such that all elements in A are smaller than the median and all elements in B are greater than the median.

2. The size of A will be $> .3n$ and $< .7n$ and same will be the size of B.

3. Thus we can say there exist at least $n/4(< .3n)$ numbers less than the median and at least $n/4(> .7n)$ numbers that are greater than the median.