CIS: 606 BONUS ASSIGNMENT.

AMIT TERDAL
CSUID: 2887869

ANSWER 1: Functions in increasing order asymptotically:

$2^{900} < \log(\log n)^2 < \log n = \log_4 n < \sqrt{n} < 2^{\log n} < n \log n <$

$n^2 \log^5 n < n^3 < 2^n < n!$.

ANSWER 2: For each recurrence, $T(n)$ is constant for $n \leq 2$

(a) $T(n) = 4.T\left(\frac{n}{2}\right) + n\sqrt{n}$.

Expanding the Recurrence:

$T(n) = 4T\left(\frac{n}{2}\right) + n\sqrt{n}$.

$= 4\left[4T\left(\frac{n}{4}\right) + \frac{n\sqrt{n}}{2}\right] + n\sqrt{n}$

$= 16\, T\left(\frac{n}{4}\right) + 2n\sqrt{n}$.

continuing the expansion, we get

$= 4^i.T\left(\frac{n}{2^i}\right) + in\sqrt{n}$.

We stop expanding when the argument as T becomes 1. Similar to $\frac{n}{2^i} = 1$; $i = \log n + \log_2 n$.

Substituting 'i' with $\log_2 n$ & $\frac{n}{2^i} = 1$.

$T(n) = 4^{\log_2 n}\, T(1) + (\log_2 n)\, n\sqrt{n}$

$4^{\log_2 n} = n^2$ and $T(1) = O(1)$

$T(n) = n^2.\, O(1) + n\sqrt{n}.\log_2 n$

$= O(n^2) + n^{1.5} \log_2 n$.

Between $O(n^2)$ & $n^{1.5}\log_2 n$, $O(n^2)$ term grows faster than $n^{1.5}\log_2 n$ for large n.

Scanned with CamScanner

$\therefore$ The Solution to the recurrence.

$T(n) = 4.T\left(\frac{n}{2}\right) + n\sqrt{n}$ is $T(n) = O(n^2)$

(b) $T(n) = 2T\left(\frac{n}{2}\right) + n\log n$.

For this problem, none of the case of master theorem can apply.

By expanding the recurrence, we have

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

$$= 4T\left(\frac{n}{4}\right) + n\log\left(\frac{n}{2}\right) + n\log n$$

$$= 8T\left(\frac{n}{8}\right) + n\log\left(\frac{n}{4}\right) + n\log\frac{n}{2} + n\log n$$

$$= \ldots\ldots$$

$$= 2^i.T\left(\frac{n}{2^i}\right) + n\log\frac{n}{2^{i-1}} + \ldots\ldots + n\log\frac{n}{2} + n\log n.$$

The expanding will not stop until $\frac{n}{2^n} = 1$, $k = \log n$.

$$T(n) = 2^k.T\left(\frac{n}{2^k}\right) + n\log\frac{n}{2^{k-1}} + \ldots + n\log\frac{n}{2} + n\log n$$

Given $2^k = n$ & $T(1) = O(1)$

$$2^k.T\left(\frac{n}{2^k}\right) = n.O(1)$$

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

we can derive the following:

$$n\log\frac{n}{2^{k-1}} + \ldots + n\log\frac{n}{2} + n\log n$$

$$\vdots$$

$$n(\log n - (k-1)) + \ldots + n(\log n - 1) + n\log n$$

$$k.n\log n - n(k-1) + \ldots + 2 + 1$$

$$k.n\log n - n(k-1)$$

Scanned with CamScanner

$K\, n(\log n - \frac{K-1}{2})$

Since K is $\log n$, $\log n - \frac{K-1}{2} = (\log n+1)/2$

$T(n) = n.\ O(1) + n \log n \cdot \frac{\log n+1}{2} = O(n \log^2 n)$.

ANSWER 3:

(a) A → array of 'n' distinct number (no two numbers are equal)

A(1 ---- n)

if $i<j$ & $A[i] > A[j]$, then the pair $A[i]$, $A[j]$ is called an inversion of A.

In other words a pair of element where first element is greater than second element & first element appears before the second in array

list all inversions of array {4, 2, 9, 1, 7}

(4,2) (4,1) (2,1) (9,1) (9,7)

(b) if we arranged array in decreasing order then the array with most inversions.

for example: n=7.

The array would be {7, 6, 5, 4, 3, 2, 1}

The total number of inversions in this array can be calculated as follows:

Array of size 'n'. First element will form (n-1) inversions, Second element will for (n-2) inversions & so on until the second to last element which form 1 inversion with the last element

The total number of inversions is the sum of first (n-1) integers, which will be $\frac{n(n-1)}{2}$

$\frac{n(n-1)}{2}$ modify of $\frac{n(n+1)}{2}$ this come from sum of first 'n' natural numbers.

Scanned with CamScanner

Since we are summing upto $\frac{n-1}{2}$ the formula modifies to $\frac{n(n-1)}{2}$.

Therefore, the array with elements from the set $\{1, 2, \ldots n\}$ that has the most inversions is array, arranged in decreasing order & it has $\frac{n(n-1)}{2}$ inversions

(c) Pseudo code for the function that counts inversions in an array.

Main function to count inversions in the array:

```
def count inversion (arr) :
    if len(arr)<=1                Base case : If the array is
        return arr,0.                         empty or has no
                                              elements, no inversion.
```

Dividing the array into 2 halves:

```
mid = len (arr)/2
left_half = arr[: mid]
right_half = arr (mid: ].
```

Recursively sort & count inversions in each half:

```
Sorted left , inversion left = count Inversion (left_half)
Sorted right, inversion right = count Inversion (right_half)
```

Merge the two halves & count inversions b/w them.

```
Sorted_Array, cross inversions = merge count( Sorted_left, Sorted right)
```

Total inversions are sum of all inversion in each half & cross inversions.

Total Inversion = inversion left + inversion right + cross Inversion

```
return Sorted_Array , Total Inversion .
```

This function to merge two sorted halves & count cross inversions:

Scanned with CamScanner

```
def mergecount (left, right)
    merged = []                 Resulting merged array
    cross inversion = 0         count of cross inversions.
    i,j = 0,0

    while i<len(left) & j<len(right)
        if left[i] <= right[j]
            merged.append(left[i])
    else : Right element is smaller means an inversion merged
    append(right[j])
    cross inversion += len(left) - i     count inversion &
    j++

    Adding remaining elements from both values, if any :
    merged.extend(left[i:])
    merged.extend(right[j:])
```

The time complexity of the inversion count Alg, which is modification of merge sort is $O(n \log n)$

Divide : The array is divided in half in each recursive call.

The number of levels is proportional to $\log n$.
Dividing an array of size 'n' repeatedly in half will take $\log_2 n$ to reach arrays of size '1'.

<u>Conquer</u> : In each recursive call, the algorithm sorts & counts inversions in the sub arrays.

The part is done in linear time relative to the size sub array.

<u>combine</u> : Merge & count : main work happens. This is $O(n)$ operation for each Merge.

Putting it all together, at each level of recursion: the Algorithm performs $O(n)$ work for merging & counting inversions across all subproblems at that level. Since there are 'log n' levels as the array is halved at each level

So, total work done is $O(n \log n)$.

Scanned with CamScanner

ANSWER 4: (1) $\frac{n}{6}$: Medians of half groups are smaller than Pivot.

$= \frac{n}{3} \times \frac{1}{2}$ groups $\Rightarrow \frac{n}{6}$.

(2) $\frac{n}{3}$: In each such groups, atleast 2 elements are similar than Pivot.

$\Rightarrow$ at least $\frac{n}{3} \times \frac{1}{2} \times 2 = \frac{n}{3}$.

(3) $\frac{n}{3}$.

(4) $\frac{2n}{3}$.

(5) $\frac{n}{3}$.

(6) $\frac{n}{3}$

(7) $\frac{2n}{3}$

(8) $\frac{2n}{3}$: $x = \frac{2n}{3}$

(9) $T(n) \leq c\frac{2n}{3} \log \frac{2n}{3} + c\frac{n}{3} \log \frac{n}{3} + n$

(10) 3 : $c = 3$.

ANSWER 5: Dynamic Programming Algorithm: To find a subset of a given set of integer whose sum is closest to target 'M'.

Function closest_ss (A, M) — A → array of integers

n = length of A — M → integer

K = sum of all elements in A.

(1) Initialising:

create a 2D array 'Sol' with dimensions

$(n+1) \times (K+1)$

Set all values to false

Sol[0][0] = True // Base Case

Scanned with CamScanner

for i from 1 to n.
For j from 0 to K.

dol[i][j] = dol[i-1][j]
if A[i-1] <= j then
dol[i][j] = dol[i][j] OR dol[i-1][j-A(i-1)]

closet = 0
For j from 0 to M
if dol[n][j] is true
closest = j.
Return closet.

closest_ss = Take an array 'A' & an integer 'M' as input.
A 2D array 'dol' is created, where dol[i][j] will be true if there will a subset of ist 'i' elements of 'A' that add up to 'j'.

<u>Base case</u>: dol[0][0] → True ; no elements sum '0'
Table is filled based on: dol[i][j] = dol[i-1][j] || dol[i-1][j-A[i-1]] if A[i-1] <= j.

It iter from '0' to 'M' in the last row of 'dol' table to find largest 'j' for dol[n][j] = True
This value is the closest sum to 'M'.

The function will return the closest sum as an integer.

<u>Time complexity analysis</u>: The overall time complexity of the algorithm is governed by the filling of the 'dol' table which is O(nK) or in other words time complexity of this algorithm is O(nK) because it involves filling a 2D array of size "n*K".

Scanned with CamScanner