CSCI B505 - Fall 2018
Written Assignment 2:

1. ⓐ $T(n) = T(n/2) + n$

$T(1) = \text{constant}$

Size	Tree	Work
n	$T(n)$	n
$\frac{n}{2}$	$T(\frac{n}{2})$	$\frac{n}{2}$
$\frac{n}{4}$	$T(\frac{n}{4})$	$\frac{n}{4}$
$\frac{n}{n}$	$T(1)$	const.

Total Work $= n + \frac{n}{2} + \frac{n}{4} + \dots + \text{const.}$

$= n(1 + \frac{1}{2} + \frac{1}{4} + \dots) + \text{const.}$

$\simeq 2n$

$= O(n)$

ⓑ $T(n) = T(n/5) + n^2$

Size	Tree	Work
n	$T(n)$	n^2
$\frac{n}{5}$	$T(\frac{n}{5})$	$(\frac{n}{5})^2 = \frac{n^2}{5^2}$
$\frac{n}{25}$	$T(\frac{n}{25})$	$(\frac{n}{25})^2 = \frac{n^2}{5^4}$
$\frac{n}{125}$	$T(\frac{n}{125})$	$(\frac{n}{125})^2 = \frac{n^2}{5^6}$
$\frac{n}{n}$	$T(1)$	const.

Total work $= n^2(1 + \frac{1}{5^2} + \frac{1}{5^4} + \frac{1}{5^6} + \dots) + \text{const.}$

$\simeq O(n^2)$

(c) $T(n) = T(n/3) + \text{const.}$

Size	Tree	Work
n	$T(n)$	const.
$\frac{n}{3}$	$T(\frac{n}{3})$	const.
$\frac{n}{9}$	$T(\frac{n}{9}) = T(\frac{n}{3^2})$	const.
$\frac{n}{n}$	$T(1)$	const.

$\log_3 n$

Total work = Const. * ($\log_3 n$)

$\simeq O(\log_3 n)$

2. We can write a recursive function to solve the problem of "compute x^n", given x, n.

Power (x, n) .... (divide & conquer method is used)

if $n == 0$ then return 1 ... $x^0 = 1$ for $x > 1$

elseif $(n \% 2) == 0$: (if n is even)

return((power $(x, \frac{n}{2})$) * (power $(x, \frac{n}{2})$))

else : ...... if n is odd

return (x * power$(x, \frac{n}{2})$ * power $(x, \frac{n}{2})$)

Time complexity for the above problem :-

Size	Tree	Work
n	$T(n)$	n
$\frac{n}{2}$	$T(\frac{n}{2})$	$\frac{n}{2}$

$(n + \frac{n}{2} + \ldots)$

$= n(1 + \frac{1}{2} + \ldots)$

$= O(n)$

The power(x,n) is taking O(n) time Complexity. But it can be Optimized to produce better time complexity.

Power (x, n): (Optimized Version)

```
temp = 0        ... initializing temp variable.
if n==0 return 1
temp = power(x, n/2)
if (n%2 = =0)        ...(if n is even)
    return temp* temp
else
    return x * temp * temp
```

Time Complexity of above function :-

Size	Tree	Work
n	n	Const.
$\frac{n}{2}$	$\frac{n}{2}$	Const.
⋮	⋮	⋮

logn times

total work = cost * (log n)

$\simeq O(\log n)$

So by storing the value in a variable we can reduce the time complexity from O(n) to O(logn).

3. Finding Median of two sorted arrays with equal length n in $O(n)$ time.

let my 2 arrays be A_1, A_2 and m_1, m_2 are the medians of the 2 arrays respectively.
Let m be the median of the 2 arrays.

if $(m_1 == m_2)$: return m_1 (or m_2) ... as our work is done (m_1 or m_2 is the median)

if $(m_1 > m_2)$:

then m lies in either of the 2 subarrays below

(i) 1st element of A_1 to m_1.

(ii) m_2 to last element of A_2.

else if $(m_2 > m_1)$:

then m lies in either of the 2 subarrays below

(i) m_1 to last element of A_1.

(ii) 1st element of A_2 to m_2.

We will repeat the above process until the size of both the sub arrays becomes 2.

if size of sub array == 2:

$$m = (\max(A_1[0], A_2[0]) + \min(A_1[1], A_2[1]))/2$$

Proving by example.

let $A_1 = [1, 12, 15, 26, 38]$
$A_2 = [2, 13, 17, 30, 45]$
> 2 sorted arrays Of equal length.

$m_1 = 15$, $m_2 = 17$ for the above 2 arrays.

since $m_2 > m_1$ ∴

The 2 new sub arrays will be -

$A_1 = [15, 26, 38]$ $A_2 = [2, 13, 17]$

$m_1 = 26$ $m_2 = 13$.

Now $m_1 > m_2$ ∴

So 2 new sub arrays are -

$A_1 = [15, 26]$, $A_2 = [13, 17]$

Now size of both sub arrays == 2 :-

$m = \{max(15, 13) + min(26, 17)\} / 2$

$= (15 + 17)/2$

$= 32/2 = 16$

So our algorithm is proved to be correct.

4. (a) A[] is nearly sorted :-

We can consider this array as K sorted array, where each element is at most K moves away from its correct position.

We can sort the above array with insertion sort with time log(nk) complexity. But for better time complexity we can use min heap (heap data structure). In this case, we need to construct a k+1 size of min heap and insert 1st k+1 elements.

Then we can remove min element from heap and add next element from array to the heap until both heap and the array is exhausted. The time complexity for the above solution will be O(n log k).

* repeat (until all the elements covered)

(1) Pop min element from min heap and assign it to next available array index

list.append (pq.pop(0)) → pq is priority queue

pq.append (list(index+1)) → next element

while (pq != 0):

list.append (pq.pop(0

b) A[] consists of random numbers :-

I would choose Mergesort algorithm to sort the array of random numbers.

I ~~too~~ choose merge sort because of its time complexity O(n log n) which is faster than most of the other sorting algorithms. I didn't choose bucket sort algorithm, which has a better time complexity O(n), because it has 3 issues.

1. Need to handle duplicates
2. Need to maximum number of the array
3. It takes too much space.

→ Quick sort and heap sorts are discarded because of worst case & instability

Merge sort works like divide and conquer.

Steps :-

1. Divide the array into 2 subarrays
2. Sort each subarray (Conquer)
3. Merge them into one.

Merge sort:

1. For i = 1 to n_1 (n_1 is the max number of 1st sub array)

~~L[i] = A[i]~~

Sort the 1st array.

For j = 1 to n_2 (n_2 the max number of 2nd sub array)

Sort the 2nd array.

How to sort :-

L[i] ≤ R[j] (L is the 1st subarray, R is the 2nd subarray)

put L[i] into the main array.

else

put R[j] into the main array.

→ Merge the 2 subarrays with the above process. We can call recursively this algorithm to sort all the numbers.