1. The Fibonacci series can be computed as follows, F(n) = F (n − 1) + F (n − 2) (1)
   In class, we showed how this can be done in O (log n) computation time. Now suppose that the definition is changed in the following way, F ' (n) = F ' (n − 1) + F ' (n − 2) + F ' (n − 3) + F ' (n − 4) (2)
   Can F ' (n) be computed in O (log n)? If yes, please show how it can be done. If no, show a counterexample where this fails. Please provide your rationale for both. Assume that F ' (0) = 0, F' (1) = 1, F' (2) = 1, F' (3) = 1.

Ans:

For solving the matrix exponentiation, we are assuming a
linear recurrence equation like below:

F(n) = F(n-1) + F(n-2) + F(n-3) + F(n-4) for n >= 4
. . . . . Equation (1)

For this recurrence relation, it depends on four previous values.
Now we will try to represent Equation (1) in terms of the matrix.

[First Matrix] = [Second matrix] * [Third Matrix]

F (n)	= Matrix 'C' *	F(n-1)
F(n-1)		F(n-2)
F(n-2)		F(n-3)
F(n-3)		F(n-4)

Dimension of the first matrix is 4 x 1 .
Dimension of the third matrix is also 4 x 1.

So the dimension of the second matrix must be 4 x 4
[For multiplication rule to be satisfied.]

Now we need to fill the Matrix 'C'.

So according to our equation.
F(n) = F(n-1) + F(n-2) + F(n-3) + F(n-4)
F(n-1) = F(n-1)
F(n-2) = F(n-2)
F(n-3) = F(n-3)

C = [1 1 1 1
1 0 0 0
0 1 0 0
0 0 1 0]

Now the relation between matrix becomes:

```
[First Matrix]  [Second matrix]      [Third Matrix]
| F(n)   |  =  | 1 1 1 1|  *        | F(n-1) |
| F(n-1) |     | 1 0 0 0 |          | F(n-2) |
| F(n-2) |     | 0 1 0 0 |          | F(n-3) |
| F(n-3) |     | 0 0 1 0 |          | F(n-4) |
```

Lets assume the initial values for this case :-
F(0) = 0
F(1) = 1
F(2) = 1
F(3) = 1

So, we need to get F(n) in terms of these values.
So, for n = 4 Equation (1) changes to

```
| F(4) |  = | 1 1 1 1|  *        | F(3) |
| F(3) |     | 1 0 0 0 |          | F(2) |
| F(2) |     | 0 1 0 0 |          | F(1) |
| F(1) |     | 0 0 1 0 |          | F(0) |
```

Now similarly for n = 5

```
| F(5) |  = | 1 1 1 1|  *        | F(4) |
| F(4) |     | 1 0 0 0 |          | F(3) |
| F(3) |     | 0 1 0 0 |          | F(2) |
| F(2) |     | 0 0 1 0 |          | F(1) |
```

- - - - 2 times - - -

```
| F(5) |  = | 1 1 1 1|  *  | 1 1 1 1|     *    | F(3) |
| F(4) |     | 1 0 0 0 |    | 1 0 0 0 |        | F(2) |
| F(3) |     | 0 1 0 0 |    | 0 1 0 0 |        | F(1) |
| F(2) |     | 0 0 1 0 |    | 0 0 1 0 |        | F(0) |
```

So for n, the Equation (1) changes to

- - - - - - - - n -2 times - - - - -

```
| F(n)   |  =  | a b c | * | a b c | * ... * | a b c | * | F(2) |
| F(n-1) |     | 1 0 0 |   | 1 0 0 |         | 1 0 0 |   | F(1) |
| F(n-2) |     | 0 1 0 |   | 0 1 0 |         | 0 1 0 |   | F(0) |
```

```
| F(n)   |  =  [ | 1 1 1 1 | ] ^ (n-3)   *  | F(3) |
| F(n-1) |     [ | 1 0 0 0 | ]               | F(2) |
| F(n-2) |     [ | 0 1 0 0 | ]               | F(1) |
| F(n-3) |     [ | 0 0 1 0 | ]               | F(0) |
```

So we can simply multiply our Second matrix n-2 times and then multiply it with the third matrix to get the result. **Multiplication can be done in (log n)** time using Divide and Conquer algorithm

2. In class we have discussed the Quick Sort algorithm. Answer the following questions regarding quick sort.

• When does the worst case for Quick Sort occur?

1) Array is already sorted in the same order.
2) Array is already sorted in reverse order.
3) All elements are the same.

• What would be the worst-case recurrence for Quick Sort? Solve the recurrence and get its time complexity in worst case.

The worst-case time complexity of Quicksort is: $O(n^2)$
Assuming the turn component is generally the littlest or biggest component of the (sub)array (for example since our feedback information is now arranged and we generally pick the final remaining one as the turn component), the cluster wouldn't be separated into two roughly similarly estimated parts, yet one of length 0 (since no component is bigger than the turn component) and one of length n-1 (all elements except the pivot element)
Thusly we would require n dividing levels with a parceling exertion of size n, n-1, n-2,.

The parceling exertion diminishes directly from n to 0 - by and large, it is, subsequently, ½ n. Consequently, with n apportioning levels, the complete exertion is $n * ½ n = ½ n^2$.

So,
$T(n) = T(n - 1) + \Theta(n)$ Recurrence
The worst-case time complexity of Quicksort is: $O(n^2)$

• What would be the best-case recurrence for Quick Sort? Solve the recurrence and get its time complexity in best case.

The best-case time complexity of Quicksort is: O(n log n)
Quicksort accomplishes ideal execution assuming we generally partition the clusters and subarrays into two allotments of equivalent size.
Since then, at that point, assuming the quantity of components n is multiplied, we just need one extra partitioning level p.

So the quantity of dividing levels is log2 n. At each partitioning level, we have to divide a total of n elements into left and right partitions

This apportioning is finished - because of the single circle inside the parceling - with straight intricacy: When the cluster size duplicates, the dividing exertion copies also. The complete exertion is, subsequently, the equivalent at all apportioning levels.
So we have n elements times log2 n partitioning levels

$T(n) = 2T(n /2) + cn$ Recurrence
So, **The best-case time complexity of Quicksort is: O(n log n)**

• Suppose you have an algorithm which can give the median of a set on numbers in O(n) time. How can this be used to improve the worst-case time complexity of Quick sort?
Ans:

The best case occurs when every partition half the list equally. ---- 1
Median by definition is the middle element of sorted list.

So if we have algorithms which gives us median, we can put is at middle and it can equally divide set in two parts. -----2
So from 1 and 2, by using median, we can achieve best case scenario Time complecity i.e. **(nlogn)**

• In the execution of Quick sort suppose every time the n/5 th element is picked as the pivot element among n elements, what would be the recurrence for quick sort? What would be its time complexity?

Ans: lets divide at n/5 th position -
N = n/5 + 4n/5
n/5 + 4n/5 = n/25 + 4n/25 + n/25+ 4n/25 =n

At certain point the function slowly reaches=> 1

So n/(5)^k and n/(5/4)^k
'K' is number of level

n/(5/4)^k = 1
n = (5/4)^k

log n = k log (5/4)

3. If k is a non-negative constant, then prove that the recurrence:
T(n) = k , for n = 1 and T(n) = 3T(n/2) + kn, for n > 1
has the following solution (for n a power of 2):
T(n) = 3knlog23 − 2kn

Ans:
T(n) = 3T(n/2) + kn
T(n/2) = 3T(n/4) + kn/2
.
.
.
T(n/2 m-1) = 3T(1) + kn/2 m-1

Substituting,
$T(n) = 3\,mk + (3/2)\,m\text{-}1kn + (3/2)\,m\text{-}2kn + \ldots\ldots + (3/2)\,0kn$
Where $n = 2\,m$
$= 3 \log_2(n)k + kn\,[\,(3/2)m\text{-}1 + \ldots\ldots. + (3/2)0\,]$
$= 3 \log_2(n)\,k + 2kn[(3/2)m - 1]$
$= 3 \log_2(n)k + 2kn[(3\log_2(n) / 2 \log_2(n)\,) - 1]$
$= 3 \log_2(n)k + 2kn[3\log_2(n) / n - 1]$
$= 3 \log_2(n)k + 2k[3\log_2(n) - n]$
$= 3 \log_2(n)k + 2k3 \log_2(n) - 2kn$
$= 3k3 \log_2(n) - 2kn$
$= 3knlog_2(3) - 2kn$

4. Design an algorithm to compute the 2nd smallest number in an unordered (unsorted) sequence of numbers {a1, a2, ..., an} in $n + \lceil \log_2(n) \rceil - 2$ com parisons in the worst case. If you think such an algorithm can be designed, then show how it can be done. If your answer is no, then explain why it cannot be done.
Ans:

Algo Approach:

We start with tracking down the biggest number in a cluster. We can find the biggest component by contrasting the components as tuples, until only one component remains which is again the biggest component in the exhibit.

We can track down the second biggest component as follows:

Track down the biggest component.
Gather all components of the exhibit that were straightforwardly contrasted with the biggest component.
Track down the biggest component among them.
While finding the biggest component, we additionally store every one of the components, the biggest component was contrasted with in a cluster Compared[].Every time any component loses in a correlation with the biggest component, we store it in this exhibit. Second biggest component is one of the component of this cluster as it prevails upon any remaining components yet loses to the biggest component and afterward view as the biggest among these components.

Likewise, as examined prior, the second biggest component wins the examination fight against any remaining components with the exception of the biggest component and in this way the second biggest component should be the biggest component of the rundown of washouts cluster which has log n components. Thusly, the second call to FindMaxTournament() (to track down the second biggest component) passes a variety of size all things considered log n and accordingly, it utilizes $\log n - 1$ correlations.

As FindSecondMax() utilizes n - 1 + log n - 1 examinations i.e n + log n - 2 correlations. Subsequently to track down the second biggest component, least n + log n - 2 examinations are required.

The time intricacy of both the talked about techniques is same i.e O(n) yet the subsequent strategy utilizes lesser number of correlations.
In this manner, we presume that the base number of correlations expected to find the second biggest component is n + log n - 2.