Homework 1.
Due: Thursday, May, 27, 2021 before 11:59PM via Gradescope. Late submission with no penalty by Friday, May 28, 2021 before 11:59AM.

Problem 1 (Finding a bit).

Design:
Define our algorithm $findOne(i, j)$, as:
1. Base case scenario: if i = j, run $I(i, j)$, if $I(i, j)$ returns i, return 1, otherwise return 0.
2. If i $\neq$ j, run $I(i, j)$, if $I(i, j)$ returns 0, return 0. Otherwise if $I(i, j)$ returns 1, return $findOne(i, \frac{i+j}{2})$ if it's not zero, otherwise return $findOne(\frac{i+j}{2} + 1, j)$. The fraction $\frac{i+j}{2}$ is rounded down if $i + j$ is odd. Having this definition, the wanted ouput is returned by calling $findOne(1, n)$.

Correctness:
The idea of the algorithm is by using Divide-and-Conquer and having $I(i, j)$, we do not need to iterate through the entire list to check if $x_k = 1$ exists.

The base case is when the list has been divided into only 1 number, having index i = index j. Now if $I(i, j)$ returns 1, we know $x_k = 1$ is found where k = i (or j), so we return i. Otherwise, if it returns 0, meaning that this number is not a 1, simply return 0 and move on.

The recursive part of the algorithm first runs $I(i, j)$, if it returns 0, we know that there can't be any existing 1 in the interval $[i, j]$, so we return 0. Otherwise, if it returns 1, although we know that 1 exists in the interval, we still can't be sure where it is.
So we call $findOne(i, \frac{i+j}{2})$ and $findOne(\frac{i+j}{2} + 1, j)$.
We divide the list into half by splitting the list from index $\frac{i+j}{2}$, which is the middle. The reason we choose the middle is that the list given has no significant pattern so there is no way to choose a more efficient pivot and dividing it by half is the optimial choice.
We understand that the two recursive call will eventually be divided into the base case or return 0 if no 1 exists in the sublist, so we know the return value is either a 0 or the index where we found the 1.
So we return the first function checking the first half of the list, return its returned value if it's not zero. If the returned value is 0, we move on to the second half of the list and return its returned value. Since we already checked the list using $I(i, j)$, we know if the first half does not have 1, then the second half must have 1.
By doing this, we limits the number of subproblem to 1 because we proceed only if function $I(i, j)$ returns 1 and if the first half of the list has a 1, we don't need to check the second half of the list anymore.

Runtime analysis:
Because we divides the list into half, contains 1 subproblem in each recurrence, and having the checking function $I(i, j)$ of time complexity $O(1)$,
we get a recurrence relation of:

$$T(n) = T(n/2) + O(1)$$

By Master Theorem, the algorithmś runtime complexity is $O(\log n)$.

Problem 2 (Buying/Selling stock).

2a)
Design:
Define our algorithm $simpleBP(A = [a_1, ..., a_n])$, as:
1. Create variables named min and max with initial value a_1 and $a_{n/2+1}$.
2. Iterate from a_1 to $a_{n/2}$, update variable min if a_i is less than min. Similarily, itrate from $a_{n/2+1}$ to a_n, update variable max if a_i is greater than max.
3. Return $max - min$ if it's a non-negative number, otherwise return 0.
Having this definition, the wanted output is returned by calling $simpleBP(A)$.

Correctness:
The idea of this algorithm is to iterate through each single value of the list and find a minimum value in the first half of the list and a maximum value in the second half.
By keeping track of the current minimum/maximum value and update them when a less/greater value is found, after the iteration is done, the we are able to find out the min of the first half elements and the max of the second half elements.
Then, we substract the max with the min, we will find the greatest difference between the first $\frac{n}{2}$ days and the last $\frac{n}{2}$ days. In case if such pair does not exist (the input list is in descending order), we simply just return 0 implying that such pair does not exist.

Runetime Analysis:
Because the algorithm iterate through the input array one time, the runtime complexity is $O(n)$.

2b)
Design:
Define our algorithm $bestPair(A = [a_1, ..., a_n])$, as:
1. Base case scenario: if the length of A = 2, return $a_2 - a_1$.
2. If the length of A $\neq$ 2, create varibale max with initial value $simpleBP(A)$.
3. Return the maximum value among max, $bestPair(A_1 = [a_1, ..., a_{n/2}])$, and $bestPair(A_2 = [a_{n/2}, ..., a_n])$.

Correctness:
The idea of the algoithm is by using Divide-and-Conquer to calculate the maximum profit to sell of divided sub-arrays and find the maximum of all the possible best options.

The base case is when the list has been divided into only 2 number. There is only one possible buy/sell option, which is the second day price minus the first day price. In this case, just simpy return their difference.
We are ensured that a base case that takes in a one-element array won't exist because the length of the list is assumed to be a power of 2.

The recursive part of the algorithm first runs $simpleBP(A)$ from **2a** and stores the return value into a created variable max. This could be the possible most profitable outcome we make so save this for later use.
Now, to perform divide-and-conquer, we need to divide the input array A into two arrays $A_1 = [a_1, ..., a_{n/2}]$ and $A_2 = [a_{n/2}, ..., a_n]$. The reason we choose to split it from the middle is that the array given has no significant pattern so there is no way to choose a more efficient ipvot and dividing it by half is the optimal choice. After that, we call $bestPair(A_1)$ and $bestPair(A_2)$.
We understand that the two recursive call will provide us with the existing best-pair price of the arrays, so what we need to do is to compare the return values of these two function and the max value and return the maximum among the 3 values.
Now the algorithm will eventually output the highest possible best-pair price of the input array.

Runtime analysis:
Because the array is divided into half, each recurrence calls 2 subproblems, and each recurrence calls $simpleBP()$ of time complexity $O(n)$ one time,
we get a recuurence relation of:

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

By Master Theorem, the algorithm's runtime complexity is $O(n \log n)$.