QUESTION 1:

You're given an array A of n integers, and must answer a series of n queries, each of the form: "How many elements a of the array A satisfy Lk ≤ a ≤ Rk?", where Lk and Rk (1 ≤ k ≤ n) are some integers such that Lk ≤ Rk. Design an O(n log n) algorithm that answers all of these queries.

- The question is asking us to find the number of element that satisfy the queries in the question
- To find the Lk and Rk we need a SEARCHING ALGORITHM
    - The searching algorithm that we learnt before are Binary Search and Linear Search, the time complexity of Binary search O(log n) is smaller than the upper bound O(nlogn) and better than the time complexity of Linear Search which is O(n)
    - Thus we use Binary Search in this question
- Binary Search requires the array to be sorted, and it repeatedly dividing the search interval in half, so we need a SORTING ALGORITHM
    - The two typical sorting algorithm that we learnt before are quick sort and merge sort, the worst case of quick sort is O(n^2) which is greater than the upper bound that mentioned in the question O(nlogn), and the time complexity of merge sort is O(nlogn).
    - So we use Merge Sort in this question for searching algorithm
- Binary Search also can't solve few problems in this question
    1. Binary search only return the Index of the element that matches the searching target
        i. Because the [Lk,Rk] is a range and the question didn't mention that they has to be the exact element in the array, so they might not existing in the array.
        ii. If duplicate element exist in the array and the Lk or Rk is equals to the duplicate element, which index would the binary search algorithm return.

    SOLUTION: find the FIRST element that is equals or greater than Lk and find the LAST element that is equals or smaller than Rk.

- We need an algorithm to find the INDEX of the FIRST element that is equals or greater than Lk

```
# Find the INDEX of FIRST element that is euqlas or greater than Lk
# the array is SORTED# now, low is the index of the first element in the array, and high is the index of the last
FindLeft(array, Low, High, Lk)
    # If the FIRST element is greater and equals to Lk
    if (array[low] >= Lk)
        then return 'low'
    # If the LAST element is smaller than Lk then theres no a exist in the range
    elif (array[high] <= Lk)
        then theres no such a exist in the range [Lk,Rk]

# based on binary search, repeatedly dividing the search interval in half
# and ignore half of the element after one comparison
    mid = (low + high)/2
    # start to compare the Lk to array[mid]
    if array[mid] > Lk
        then ignore the right part and keep searching the left side
        FindLeft(array, Low, Mid, Lk)
    elif array[mid] == Lk
        if array[mid-1] < Lk
            return mid
        else
            FindLeft(array, Low, mid, Lk)
    elif array[mid] < Lk
        if array [mid+1] >= Lk
            return mid+1
        else
            FindLeft(array, mid, high, Lk)
```

- And another algorithm to find the INDEX of the LAST element that is equals or smaller than Rk

```
# Find the INDEX of LAST element that is euqlas or smaller than Rk
# the array is SORTED
# now, low is the index of the first element in the array, and high is the index of the last
FindRight (array, low, high, Rk)
    // if the first element is greater than Rk, then no solution
    if array[low] > Rk
        then no such a exist in the range [Lk,Rk]
    # set mid for binary search
    mid = (low + high)/2
    if (array[mid] < Rk)
        if (array[mid+1] > Rk)
            return mid
        else
            FindRight(array, mid, high, Rk)
    elif (array[mid] > Rk)
        if (array[mid-1] <= Rk)
            return mid-1
        else
            FindRight(array, low, mid, Rk)
    elif (array[mid] == Rk)
        if (array[mid-1] != Rk)
            return mid
        else
            FindRight(array, low, mid, Rk)
```

- After we find the index of the First element equals or greater than Lk and the index of the element equals or smaller than Rk

```
CountElements (LeftIndex, RightIndex)
    # because Lk<=Rk
    # if they are the same then theres only one element is in the range [Lk,Rk]
    if (LeftIndex == RightIndex)
        Count = 1
    else
        Count = RightIndex - LeftIndex + 1
    return Count
```

- The time complexity would be O(nlogn)

QUESTION 2 (A):

Describe an O(n log n) algorithm (in the sense of the worst case performance) that determines whether or not there exist two elements in S whose sum is exactly x.

- As we want to find if element in S whose sum is exactly x exist or not we can use the integer that we already known, X, to minus every element in the array, Diff = (x – n), and to check if the value Diff exist in the array or not.
- Binary Search (Time complexity O(logn)) algorithm can find out if one value is exist in an array or not, which is faster than Linear Search as well as mentioned in question 1
- Solution:
    - Diff = (x – n) till we find one diff exist, then return True. If we can't find any Diff exist till the last n then return False, means there no two elements in the array's sum can be exactly x. For this step, we need to do (x - n) maximum n times, and binary search the Diff for n times, which is O(n) * O(logn).
    - Before binary search we need to do a merge sort to make sure the array is sorted, which is O(nlogn).
    - Then this algorithm satisfies the upper bound O(nlogn).

```
# use forloop to make sure that every element in the array S
# is tested till the last one or the Diff has been found
# array S got n elements and X is the given integer
for (i=0; i<n; i++) in array S
    Difference = (X - array[i])
    # low is the first index, and high is the index of last element
    FindDifference(array, low, high, Difference)
        if high >= 1
            mid = (high + low)/2
            if array[mid] == Difference
                return True
            elif array[mid] > Difference
                return FindDifference(array,low,mid-1,Difference)
            elif array[mid] < Difference
                return FindDifference(array,mid+1,high,Difference)
        else
            return False
```

QUESTION 2 (B):

Describe an algorithm that accomplishes the same task, but runs in O(n) expected (i.e., average) time.

- Because of the requirement of time complexity changed to O(n), so we don't do sorting and searching anymore.
- With Hashing, we get O(1) search time on average (under reasonable assumptions), and O(n) is the worst case according to the definition of hash table
- However, we still need to check if Diff = (X – n) exist or not.
- Solution:

    If (X-array[n]) doesn't exist in the hash table, then insert the array[n] into the table, if (x-array[n]) existing in the table already means that there's two element's sum is exactly x, the current array[n] and the one already existing in the table

```
Create Hashtable H = dict()
    # theres total N elements in array S
    for(i=0; i<n; i++) in array S
        if (x - array[i]) is not exiting in hashtable H
            then insert(put) array[i] into the hashtable H
        else
        # if the differnece already existing in the hashtable
            return True
```

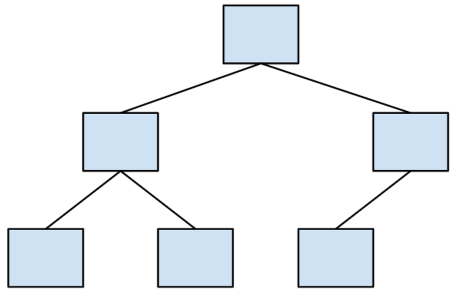
QUESTION 3 (A):

- the party attended by n people (not including yourself)
- celebrity doesn't know anyone but themselves
- all the people except celebrity knows celebrity
- Solution:
    - We ask each person question about the next person (n-1) times to get the celebrity candidate
        - because of in this question, celebrity has to know no one else but themselves, and everyone else in the party knows the celebrity so there can only be one celebrity candidate
    - We ask the celebrity candidate (n-1) times if he knows anyone
    - We ask (n-1) people if they know the potential celebrity
    - 3(n-1) = 3n-3

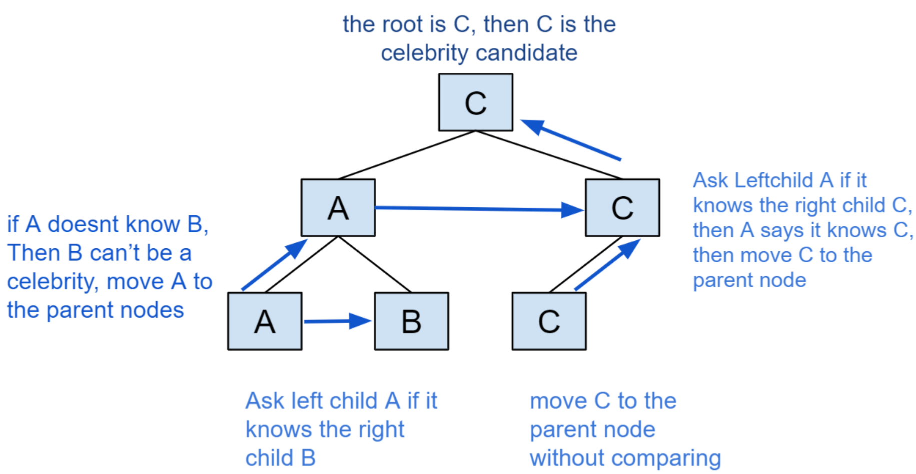
```
# there are total N people in the array
while (after we ask about everyone, its (n-1) times because the
        last person doesnt need to answer the question)
    initially i = 0, and j = 1
    if person[i] knows person[i+j]
        person[i] is not celebrity
        keep asking, i++
    elif person[i] doesnt know person[i+j]
        person[i] might be celebrity
        keep asking about the person after
        j++
# after we ask everyone, there's one person left, A
# we need to make sure that the person doesnt know anyone
# because celebrity doesnt know anyone
while (n-1) {
    if person[A] knows anyone
        then person[A] is not celebrity
        theres no celebrity in this party
    else
        then we need to check if everyone knows A
        because everyone knows celebrity
        if everyone which is (n-1) people knows A
            then A is celebrity
        else A is not celebrity
}
```

QUESTION 3 (B):

- Use the binary tree where every node has 0 or exactly 2 children, and all layers are filled except the last level
- Use every leaf represents one person in the party (not include yourself), if its odd number, the right most node of the second last level can only have one child, make sure it's a left child, like the following binary tree.



- All the nodes (including root) except the nodes in last level are empty (because the last level nodes/leaves store the 'person' that attended the party.
- Always ask the left child if it knows the the right child, if it knows then left child if not celebrity, then move the right child to the parent node, if there's only one left child, then move the left child to the parent node directly, shown in the following graph:



- After we find the celebrity candidate after the first round asking, we need to make sure that the celebrity doesn't know anyone else and everyone else in the party knows celebrity, from the part(A), we say that the three rounds questioning process, the worst case would be asking 3(n-1) times, but in this case:
  - For ask if the celebrity candidate knows everyone else:
    - If the celebrity candidate used to be the left child and the other person used to be the right child (in the same level with the same parent node), means we asked the question already, then we don't need to ask it again
  - For ask if everyone else knows the potential celebrity:
    - If the celebrity used to be the right child and the other person used to be the left child (in the same level with the same parent node), means we asked the question already, then we don't need to ask it again
- The depth of this type of binary tree is ($\log_2(n)$), and in every level of the tree we can ask at least one less question round 2 and round 3 asking
- Then we got at least ($\lfloor \log_2(n) \rfloor$ - 1) times of asking question
- Thus the total time of asking question would be $3(n-1) - (\lfloor \log_2(n) \rfloor - 1)$ which equals to $3n - \lfloor \log_2(n) \rfloor - 2$.

QUESTION 4:

Read the review material from the class website on asymptotic notation and basic properties of logarithms, pages 38-44 and then determine if f(n) = Ω(g(n)), f(n) = O(g(n)) or f(n) = Θ(g(n)) for the following pairs.

Big O Notation	• $f(n) = O(g(n))$ means that f(n) doesn't grow substantially faster than g(n) because a multiple of g(n) eventually dominates f(n) • $0 \le f(n) \le c * g(n)$ for all n>=n0 with positive constant c • $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0$
Omega Notation	• $f(n) = \Omega(g(n))$ says that f(n) grows at least fast as g(n), because f(n) eventually dominates a multiple of g(n). • $0 \le c * g(n) \le f(n)$ for all n>n0 with positive constant c • $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = \infty$
Theta Notation	• $f(n) = \Theta(g(n))$ means that g(n) is both an asymptotical upper bound and an asymptotical lower bound for f(n) • $0 \le c1 * g(n) \le f(n) \le c2 * g(n)$ for all n>n0 with positive constant c • $0 < \lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) < \infty$

PART 1:

$f(n) = \left(1og_2(n)\right)^2$

$g(n) = \log_2\left(n^{\log_2(n)} + 2\log_2(n)\right)$

we simplify the f(n) and g(n) get $f(n) = (log_2(n)) * (log_2(n))$ $g(n) = (log_2(n)) * (log_2(n)) + 2\log_2(n)$ n has to be greater than 0 because of log when n = 1, 0 = f(n) = g(n) then $0 \le f(n) \le c * g(n)$ when n >= 1, c >= 1 also $0 \le c * g(n) \le f(n)$ when n >= 1, c <= ½ so the answer is $f(n) = \Theta(g(n))$	$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = \lim_{n\to\infty}\left(\frac{((log_2(n)) * (log_2(n))}{(log_2(n)) * (log_2(n)) + 2\log_2(n)}\right)$ $= \lim_{n\to\infty}\left(\frac{((log_2(n)) * (log_2(n))}{log_2(n) * (log_2(n) + 2)}\right)$ $= \lim_{n\to\infty}\left(\frac{log_2(n)}{log_2(n) + 2}\right) = \lim_{n\to\infty}\left(\frac{1}{1 + \frac{2}{log_2(n)}}\right) = 1$ Which satisfies $0 < \lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) < \infty$ So the answer is $f(n) = \Theta(g(n))$

PART 2:

$f(n) = n^{100}$

$g(n) = 2^{\frac{n}{100}}$

$f(n) = log_2(n^{100}) = 100\, log_2(n)$ $g(n) = log_2\left(2^{\frac{n}{100}}\right) = \frac{n}{100}$ then $0 \le log_2(n) \le c * n$ when n >= 1, c >= 1 then $0 \le f(n) \le c * g(n)$ when n >= 1, c >= 1 so the answer is $f(n) = O(g(n))$	$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = \lim_{n\to\infty}\left(\frac{log_2(n^{100})}{log_2\left(2^{\frac{n}{100}}\right)}\right) = \lim_{n\to\infty}\left(\frac{100\, log_2(n)}{\frac{n}{100}}\right)$ $= \lim_{n\to\infty}\left(\frac{log_2(n)}{n}\right) = \lim_{n\to\infty}\left(\frac{1}{\ln(2) * n}\right)$ $= \frac{1}{\ln(2)} * \lim_{n\to\infty}\left(\frac{1}{n}\right) = 0$ Which satisfies $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0$ So the answer is $f(n) = O(g(n))$

PART 3:

$f(n) = \sqrt{n}$

$g(n) = 2^{\sqrt{log_2(n)}}$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{\sqrt{n}}{2^{\sqrt{log\,2(n)}}} = \lim_{n\to\infty}\frac{n}{2^{2^*\sqrt{log_2(n)}}} = \lim_{n\to\infty}\frac{log_2(n)}{2^*\sqrt{log_2(n)}}$$
$$= \frac{1}{2} * \lim_{n\to\infty}\frac{log_2(n)}{\sqrt{log_2(n)}} = \frac{1}{2} * \lim_{n\to\infty}(log_2(n))^{1-\frac{1}{2}}$$
$$= \frac{1}{2} * \lim_{n\to\infty}(log_2(n))^{\frac{1}{2}} = \frac{1}{2} * \lim_{n\to\infty}\sqrt{log_2(n)} = \frac{1}{2} * \infty$$
$$= \infty$$

Which satisfies $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = \infty$

So the answer is $\mathrm{f(n)} = \Omega(g(n))$

PART 4:

$\mathrm{f(n)} = n^{1.001}$

$\mathrm{g(n)} = n * log_2(n)$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{n^{1.001}}{n * log_2(n)} = \lim_{n\to\infty}\frac{n^{0.001}}{log_2(n)} = \lim_{n\to\infty}\frac{0.001 * n^{-0.999}}{\frac{1}{\ln(2) * n}}$$
$$= \lim_{n\to\infty} 0.001 * \frac{1}{n^{0.999}} * \ln(2) * n$$
$$= \ln(2) * 0.001 * \lim_{n\to\infty}\frac{n}{n^{0.999}}$$
$$= \ln(2) * 0.001 * \lim_{n\to\infty}(n)^{1-0.999} = \ln(2) * 0.001 * \infty$$
$$= \infty$$

Which satisfies $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = \infty$

So the answer is $\mathrm{f(n)} = \Omega(g(n))$

PART 5:

$f(n) = n^{(1+sin(\pi n/2))/2}$

$g(n) = \sqrt{n}$

For the f(n) the exponent of n has sin in there the interval of sin(x) is [-1,1], then, $$\sin\left(\frac{\pi n}{2}\right) \in [-1,1]$$ $$1 + \sin\left(\frac{\pi n}{2}\right) \in [0, 2]$$ $$\frac{1 + \sin\left(\frac{\pi n}{2}\right)}{2} \in [0, 1]$$	• Thus f(n) oscillate • Then f(n) and g(n) are not comparable • No omega, theta or big O

QUESTION 5:

PART A:

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

- For ($n^{log_b(a)}$), b = 2, a = 2
  - $n^{log_b(a)} = n^{log_2(2)} = n^1 = n$
- f(n) = n(2+sin(n))
  - Because the interval for sin(x) is [-1, 1]
  - Thus we can get $1 \le (s + \sin(n)) \le 3$, so we can treat this part as a constant value
  - Then the power of f(n) is 1
- $f(n) = \theta(n^1)$
- Which satisfies the case 2, $f(n) = \theta(\ n^{log_b(a)}\ )$
- Thus the answer would be $T(n) = \theta\left(\ n^{log_b(a)} * log_2(n)\right) = \theta\left(n * log_2(n)\right)$

PART B:

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

- For ($n^{log_b(a)}$), b = 2, a = 2
  - $n^{log_b(a)} = n^{log_2(2)} = n^1 = n$
- f(n) = $\sqrt{n} + \log(n)$
  - because there are two parts for f(n) then we compare these two by using limit rule

$$\lim_{n\to\infty}\left(\frac{\sqrt{n}}{\log(n)}\right) = \lim_{n\to\infty}\left(\frac{\frac{1}{2} * n^{1/2}}{\frac{1}{\ln(d) * n}}\right) = \lim_{n\to\infty}\left(\frac{1}{2} * \ln(d) * \frac{1}{n^{1/2}} * n\right)$$
$$= \frac{1}{2} * \ln(d) * \lim_{n\to\infty}\left(\frac{n}{n^{1/2}}\right) = \frac{1}{2} * \ln(d) * \lim_{n\to\infty}\left(n^{1-1/2}\right)$$
$$= \frac{1}{2} * \ln(d) * \lim_{n\to\infty}\left(n^{1/2}\right) = \frac{1}{2} * \ln(d) * \infty = \infty$$

  - So $\sqrt{n}$ grows faster than log(n), then we pick $\Theta(\sqrt{n})$
- $T(n) = 2 * T\left(\frac{n}{2}\right) + \theta\left(\sqrt{n}\right) = 2 * T\left(\frac{n}{2}\right) + \theta\left(n^{1/2}\right)$
- Which satisfies the first case because of $f(n) = O\left(n^{log_b(a)-\varepsilon}\right)$ as $f(n) = O(n^{1-\varepsilon})$ for $\varepsilon < \frac{1}{2}$
- Thus the answer would be $T(n) = \theta\left(n^{log_b(a)}\right) = \theta(n)$

PART C:

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

- For ($n^{log_b(a)}$), b = 2, a = 8
  - $n^{log_b(a)} = n^{log_2(8)} = n^3$
- $f(n) = n^{\log(n)}$ we can't get the exact exponent of this function, then we use the limit rule to compare these two

$$\lim_{n\to\infty}\left(\frac{n^{\log(n)}}{n^3}\right) = \lim_{n\to\infty}\left(n^{\log(n)-3}\right) = \infty$$

  - after the comparison we can get $n^{\log(n)}$ grows faster than n^3
- then we can get $f(n) = \Omega(n^{3+\varepsilon})$
- thus the answer would be $T(n) = \theta\left(f(n)\right) = \theta(n^{\log(n)})$

YAWEN LUO z5134924

PART D:

$\mathrm{T(n)} = T(n-1) + \mathrm{n}$

- Master theorem doesn't apply here for this function, so we use the inductive method

  $\mathrm{T(n)} = T(n-1) + n$
  $\mathrm{T(n-1)} = T(n-2) + (n-1)$
  $\mathrm{T(n-2)} = T(n-3) + (n-2)$
  $\mathrm{T(n-3)} = T(n-4) + (n-3)$
  $\mathrm{T(n-4)} = T(n-5) + (n-4)$
  …
  $\mathrm{T(n)} = T(n-1) + n$
  $= T(n-2) + (n-1) + n$
  $= T(n-3) + (n-2) + (n-1) + n$
  $= T(n-4) + (n-3) + (n-2) + (n-1) + n$
  …
  Then we can get

  $$\mathrm{T(n)} = T(0) + 1 + 2 + 3 + \cdots + n = \frac{n(n-1)}{2}$$

  As the term with highest exponent is n^2, so the answer would be T(n) = $\theta(n^2)$