1.

1.1. Prototype:

1.1.1. *function* (S, n, queries[n]):

```
        S->quicksort (A, n)
        int i = 0
        while (i < n) {
            int left = queries[i].Lk
            int right = queries[i].Rk
            if(A[n-1] < Lk || A[0] > Rk) {
                queries[i].result = 0
            } else {
                int Rindex = bsearch(A, right, 0, n – 1)
                if(A[Rindex] > right) Rindex -= 1
                int Lindex = bsearch(A, left, 0, n – 1)
                if(A[Lindex] < left) Lindex += 1
                queries[i].result = (Rindex - Lindex + 1)
            }
            i += 1
        }
    End function
```

1.2. Explanation: I am giving my function the queries as an array of structs called ‘queries’, the structs have members ‘Lk’, ‘Rk’ and ‘result’. Lk and Rk are set when the structs are given to the function, and result is set, initially starting as -1.First quicksort the array ‘A’ giving O(logn). Then enter a loop that will ensure the values provided are actually in the range of the array at all. If not, set the result for that query to 0. If so, do 2 binary searches which is 2O(logn) which therefore accounts for O(n2logn) which is just O(nlogn). These **adjusted** binary searches are looking for the Lk and Rk values, however if the bounds aren’t found, instead of returning -1 or some other signal that the value isn’t in the array, it still returns the index of the isolated final element for possible adjustment after the search. Example with bounds included: A = [0, 1, 2, 3, 4], Lk = 1, Rk = 4. The binary search returns Lindex = 1 and Rindex = 4. Since 1 !< 1 and 4 !> 4, no adjustment is done. Therefore, the result of that query is 4 – 1 + 1 = 4, which is correct, spanning values 1, 2, 3, 4. Example with bounds not included: A = [0, 2, 3, 4, 6], Lk = 1, Rk = 5. Lindex will either return index 0 or 1 depending on the implementation of binary search, and Rindex either index 3 or 4. If Lindex is 1, A[1] = 2, 2 !< 1 so it returns Lindex = 1. However, if the search landed at 0, A[0] = 0, and 0 < 1, so we return the index + 1 (0 + 1 = 1). For Rindex, if we landed on index 3, A[3] = 4, and 4 !> 5, so we return index 3. If we landed on index 4, A[4] = 6, 6 > 5, so we take index – 1 (4 – 1 = 3). We then go as normal 3 – 1 + 1 = 3 which is correct, spanning values 2, 3, 4. This is just an O(1) increment or decrement adjustment that does not impact the speed of binary search, and so the overall cost of doing a logn search ‘n’ times, is still O(nlogn)

2.

2.1. Part A

2.1.1. Prototype: function (S, n, x):

```
            S->quicksort (S, n)
            int i = 0
            while (i < n) {
                int first = S[i]
                if(S[i] > x) return false
                int index = binary search (S, x – S[i], 0, n - 1)
                if(index >= 0 and index != i) return true
                if(index >= 0){ if(S[i+1] == S[i] || S[i] == S[i-1]) return true }
                i++.
            }
            return false.
        End function
```

2.1.2. *Explanation:* The function first does a quicksort on the given array which is O(nlogn). It then enters an outer loop which runs from i = 0 to i = n-1, n-times in total. However, in the loop I immediately check that the current element in the array is not greater than x, and return false if it is, since nothing could be added to S[i] to obtain x if S[i] is already larger than x and since the array is sorted, this would be the case for all the following elements. This is an O(1) check each time and so is O(n) over the n iterations of the outer loop. Here I am making the assumption that integers are >= 0. If it is not true, we perform a binary search on the array for the difference between x and S[i] i.e. the digit needed to complete the sum to x, given we have S[i]. Binary search is O(logn) and so over the course of the outer loop, this adds a factor O(nlogn). So, the dominators of this function are quicksort and 'n' iterations of binary search, which both iron out to O(nlogn), therefore the entire function is O(nlogn). The worst case for this algorithm would be where x is greater than any element in the array and is not the sum of any 2 elements in the array. In this case we quicksort [O(nlogn)] and go through the loop all 'n' times, doing n binary searches before eventually returning false. This would be O(nlogn) due to the domination of quicksort and n binary searches. I made the assumption that doubling a value is not allowed, but since I know the array is sorted, in the case that X is double a number, it would have to occur twice, and so would be adjacent in the sorted array, hence the extra little adjacency checks if my binary search wants to double the number at some position in the array.

2.2. Part B

2.2.1. *Prototype: function(S, n, x):*

```
        int i = 0
        while (i < n) { hashTable(S[i]) = i }
        i = 0
        while (i < n) { if (hashTable(X – S[i]) != i) return true }
        return false
    End function
```

2.2.2. *Explanation:*

2.2.2.1. First thing to note was that we can't sort unless it was key indexed. Instead use a hashTable. This is an O(n) average summing check. Again, I assume you cannot double a number that occurs once in the array to make the sum. Initially, there's just 2 O(n) loops so it's clearly O(n). Now for correctness: the value of S[i] is the key, and the index i is the value, and importantly we keep looping and overwrite S[i] in the case it has a duplicate. This means checking hashTable(X – S[i]) != i) will check also duplicates that make the sum correct. Ex: S = [0, 5, 1, 5], we want to sum to 10. HashTable[0] = 0, hashTable[5] = 1, hashTable[1] = 2, but then hashTable[5] = 3. So when we check hashTable(10 – S[1]), we get hashTable(5) = 3, not 1, which is i. Apart from that, it's clear that we are checking a value exists in the table for the key necessary to sum to X. If it does, then we know the complement was in the array and we can return true.

3.

3.1. Part A

3.1.1. Assume the 'n' people range in index from 0 to n – 1. Every time I ask a question if X knows Y, I either get: yes (eliminate X since a celebrity knows no one) or no (eliminate Y since a celebrity is known by everyone). Start with X = 0 and Y = 1. In the general case: if X knows Y -> move person X to person Y and increment Y by 1. If they don't -> increment Y. Say X (0) doesn't know Y (1) -> Y = 2. Say X doesn't know Y -> Y = 3. Say X knows Y -> X = 3, Y = 4. You can repeat this process and will end up asking n – 1 questions. This leaves 2 cases. First, the final person (Y) was known by their predecessor (X), in which case the final person (Y) is the candidate for the celebrity. Second, the final person (Y) is not known by their predecessor (X) in which case, X is the final person who was known by someone who doesn't know someone else when we checked, so they are the candidate. This accounts for the first n – 1 questions to find a celebrity candidate

3.1.2. The second n – 1 is checking with all n – 1 other people at the party that they know the candidate. Worst case is that they all do, as then the candidate may still be a celebrity, so we need to check they know nobody. However, since we know already that one person knows the celebrity candidate in the first line of questioning, we can subtract a question, giving n-2

3.1.3. The final n – 1 is checking the candidate knows none of the other n – 1 people at the party. Worst case is that they don't know anyone (result = celebrity) or know the last person we check (result = no celebrities). Both are n – 1.

3.1.3.1. Note, the n – 1 and n – 2 aren't exclusive, i.e. if the celebrity was X & didn't know Y, we would save a question from 3.1.3, not 3.1.2, still reducing the questions needed by 1.

3.1.4. Therefore, 2 rounds of n – 1 and 1 round of n – 2 questions find a result in the **worst case of 3n – 4**, better than the required 3n-3.

3.2. Part B: Group all party attendees into pairs. Ask either member of the pair (X) if they know (Y). Then keep the pool of possible candidates for celebrities and repeat with the smaller group for the initial n-1 questions. In the process of doing this, we are halving the group size. Therefore, to be the final candidate, you must have passed *floor(*log(2, n)) rounds of questioning, representing a count of questions and answers that we don't have to repeat. We can then proceed as in part A, having saved *floor(*log(2, n)) questions. Therefore, we get n – 1 + n – 1 + n – 1 – *floor(*log(2, n)), giving a final result of **3n – *floor(*log(2, n)) – 3,** superior to the limit imposed on us by 1 question.

4.

4.1. Use log laws to simplify the expression

$$\frac{(\log(2,n))^2}{\log(2,\ n^{(\log(2,n))}) + \log(2,n)} = \frac{\log(2,n)^2}{\log(2,n)^2 + \log(2,n))} = \frac{1}{1+\frac{1}{\log(2,n)}}$$

Since $\lim_{n\to\infty} \frac{1}{1+\frac{1}{\log(2,n)}}$ -> 1 as n -> ∞,

$$(\log(2,n)^2) = \theta(\log\left(2, n^{(\log(2,n))}\right) + 2\log(2,n)$$

4.2. Use L'Hopital's Rule to obtain:

$\lim\limits_{n\to\infty} \frac{100n^{99}}{\frac{1}{25}n^{\frac{n}{100}-2}*ln2}$ which can be done 100 times, eliminating n from the numerator to get something of the form:

$\lim\limits_{n\to\infty} \frac{C}{A*n^{\frac{n}{100}-B}}$, making C, A and B insignificant as n → ∞, and making the limit -> 0.

Therefore: $n^{100} = O(2^{\frac{n}{100}})$

4.3. Take logs of both sides and set up an equality:

$$\log_2 \sqrt{n} = \log_2 c + \log_2 2^{\sqrt{\log_2 n}}$$
$$\frac{1}{2}\log_2 n = \log_2 c + \sqrt{\log_2 n}$$
$$\frac{1}{2}\log_2 n - \sqrt{\log_2 n} = \log_2 c$$
$$\log_2 n\,(\frac{1}{2} - \frac{1}{\sqrt{\log_2 n}}) = \log_2 c$$

Next, we can take c = 1, and find roots at n = 1, 16. We then observe that sqrt(n) is the smaller function in the domain (1, 16) and is then the bigger for n > 16.

Hence, we get $\frac{1}{2}\log_2 n > \log_2 1 + \sqrt{\log_2 n}$ for n > 16 and can say:

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

4.4. This can be re-written as:

$\frac{(\log_{10} 2)\, n^{0.001}}{(\log_{10} n)}$ which we can use L'Hopital's Rule on to get

$\frac{0.001\times(\log_{10} 2)\times\frac{1}{n^{0.999}}}{1/n} = n^{0.001}(0.001)(\log_{10} 2)$-> ∞ as n -> ∞

Therefore, $n^{1.001}$ = Ω $(n\log_2 n)$

4.5. Since we are taking n to a periodic exponent that ranges from 0 to 1, we range from 1 to 'n' over a cycle. Except 'n' is continually getting larger but still comes back to 1 every cycle. Hence there is no 'N' such that 'n' > 'N' satisfies any "Big" notation properties. So, the functions are incomparable.

5.

5.1.

Firstly, by the inequality 1 <= 2 + sin(n) <= 3, we know that sin(n) does not change the asymptotic behaviour of the function.

n^(log2(2)) = n and n <= f(n) <= 3n, we see that f(n) = θ(n), so by the 2nd case of the Master Theorem, we conclude that T(n) = θ(nlog2(n))

5.2.

First compare n^(0.5) and log(n) for asymptotic behaviour.

$$\lim_{n\to\infty} \frac{n\text{^}(0.5)}{\log(n)} = \lim_{n\to\infty} \frac{1/2\sqrt{n}}{1/n} = \lim_{n\to\infty} \frac{\log_b 10 * n}{2\sqrt{n}} \to \infty .$$

So therefore n^(0.5) is the dominating factor and can be used alone for comparison.

n^(log2(2)) = n, and f(n) = n^(0.5). Therefore, since f(n) = O($n^{1-\epsilon}$) for $\epsilon < 0.5$, by the first case of the Master Theorem, T(n) = θ(n).

5.3.

Part 1: First observe the equality n^(3) = n^(log(2, n)).

Log both sides and simplify to get 3logn – (logn)^2 = 0.

Factor out a logn to get logn(3 – logn) = 0, which has roots n = 4, 8.

Observe that n^logn is larger whenever n > 8, so this looks like the third case for master theorem.

Part 2: set up that for some 0 < c < 1, for all n > N:

8f(n/2) < c f(n).

$8(\frac{n}{2})^{\log_2 \frac{n}{2}} < c * n^{\log_2 n}$, which can be simplified as:

$8(\frac{n^{\log_2 n} * n^{-1}}{2^{\log_2 n} * 2^{-1}}) < c * n^{\log_2 n}$, which can further be simplified as:

$16(\frac{n^{\log_2 n}}{n^2}) < c * n^{\log_2 n}$, which can be even further simplified by dividing by n^(logn):

$(\frac{16}{n^2}) < c.$

Therefore, choose n > 4, such that c has property 0 < 16/n^2 < 1.

Then by the third case of the master theorem, T(n) = $\theta(n^{logn})$.

5.4.

T(n-1) = T(n-2) + (n-1) and so on in a similar fashion such that T(n) = T(n-k) + (n-(k-1)) + … + n

We can unwind this recurrence until the base case of T(1). This case is when n – k = 1, so then k = n – 1.

Substituting k = n – 1, we get: T(n) = T(1) + (n – (n – 1)– 1)) + … + n

-> T(n) = T(1) + 2 + 3 + … + n

-> T(n) = T(1) + $\sum_{k=3}^{n} k - \sum_{k=0}^{1} k$ = T(1) + n(n+1)/2 – 1 which is ((n^2 + n)/2) – 1

This shows that T(n) = $\Theta(n^2)$