Binary Search and its Variants

1. We first need to find a range before performing binary search. For this we can drop the egg at floors 1,2,4,8,... until it breaks. If $2^n \leq h < 2^{n+1}$, the egg breaks at the $(n+1)^{th}$ drop and $n+1 = \lfloor \log_2 h \rfloor + 1$ which is $O(\log h)$. Now we can binary search in the range from 2^n to $2^{n+1} - 1$. The range is $O(2^n)$ so the binary search will take $O(\log 2^n) = O(n) = O(\log h)$ egg droppings, so overall it's still $O(\log h)$.

2. Whenever we want to find subarray sums, it is useful to precompute a prefix sum array, which is has the sums of the first i elements i.e. $[0, a_1, a_1 + a_2, \ldots, a_1 + a_2 + \cdots + a_n]$. This takes $O(n)$ time to calculate as the i^{th} prefix sum is obtained by adding an array element to the $(i-1)^{th}$ prefix sum. With these prefix sums, any subarray sum from a_i to a_j is obtained by subtracting the prefix sum to a_{i-1} from the prefix sum to a_j. Now for every i, we can binary search the subarray sums starting from a_i, to get the least j such that sum from a_i to a_j is at least S. This is possible as sum from a_i to a_j is an increasing function w.r.t j. So for every i, we will take $O(\log n)$ time and hence the final time complexity is $O(n \log n)$.

Another method is to binary search for the minimum length directly. For fixed subarray length, finding the maximum subarray sum takes $O(n)$ time. Just find the sum for the first subarray, and for the next subarray, add the new element and subtract the first element. So this is just a 1 time traversal of the array. Now we need $O(\log n)$ queries to narrow down the fixed length by binary searching for first length where the subarray sum is at least S. Each query takes $O(n)$ time so our time complexity is $O(n \log n)$.

Turns out this question can be solved in $O(n)$ time itself. The idea is to have 2 pointers, one for the start of the subarray and one for the end of the subarray. For every start index, keep moving the end value until the sum of the subarray is at least S. Once that's reached, move the start pointer 1 step forward and repeat. Note that this would give the smallest subarray for every start index, as the end index will never need to decrease whenever the start index moves forward. Updating subarray sum is also easy whenever a pointer moves, just correct for the new element added/deleted. This algorithm is $O(n)$ as each pointer moves at most n times, they are initialized as the beginning of the array.

3. We just need to find the n^{th} and $(n+1)^{th}$ element of the merged array. For now, let's just assume that the n^{th} element is in the first array and try to find it. How to check if the i^{th} element of the first array is the n^{th} element of the merged array? We know it's greater than exactly $i-1$ elements in the first array, so it has to be greater than $(n-1) - (i-1)$ elements in the second array, as it overall has to be greater than $n-1$ elements. So we just check that, it should be greater than the $(n-i)^{th}$ index and smaller than $(n-i+1)^{th}$ index of the second array, if so we have found it. If it's too small i.e lesser than the $(n-i)^{th}$ element of the second array. We need to check from $(i+1)$ to n in the first array. If it's too big, we need to search from 1 to $i-1$ in the first array. So we can binary search, dividing the search space by half in each time. Suppose we don't find the n^{th} element in the first array, we check the second array for it. Similarly we can find the $(n+1)^{th}$ element by checking both arrays.

4. We first sort the array taking $O(n \log n)$ time. Now for each element x in the array, binary search for $S - x$. This also takes $O(n \log n)$ as we binary search n times.

After sorting, there is also an $O(n)$ approach. Keep 2 pointers at start and end. If the first and n^{th} element sum greater than S, we know that the n^{th} element can never be part of a pair, it's too big even when paired with the first element. So we discard it from our search by decrementing our end pointer. If the sum was smaller than S, similar logic follows and we increment our start pointer. We keep doing this until we either get a sum S or our pointers clash. Since the distance between them is always decreasing by 1 step this is $O(n)$

5. Since the derivative is an increasing function, we can find where it has a root by doing something like bisection method. We first find an interval where the root is, suppose we check a random point a and $f'(a)$ is negative. We then check $f'(a+1), f'(a+2), f'(a+4), \ldots$ until we get a point where the derivative is positive, to locate our interval. Once we have a start and endpoint, we check the derivative at the midpoint and based on its sign, we half the interval search each iteration, increasing the number of bits of precision each time by 1.

6. The smallest base is 2 so the highest exponent is $\log_2 a$. For each exponent k, we binary search for b to get b^k to be n. So we need $O(\log a)$ queries to get b for each k, if it exists. But how much time does each query take. Assume the naive human multiplication method, multiplication take $O(d^2)$ time where d is the number of digits. Since we only multiply till a size of a, we have $O(\log a)$ digits. We multiply almost $O(\log a)$ times to do exponentiation so each query is $O(\log^3 a)$. This makes binary searching for a fixed exponent as $O(\log^4 a)$. So our final time complexity if $O(\log^5 a)$. This algorithm can be make a lot faster by using more efficient methods for multiplication and exponentiation, but eh.

7. The highest value of f is just $\left\lfloor \frac{a_1+\cdots+a_n}{n} \right\rfloor$. This is clearly achievable by making everyone having anything above this value give off their land one by one. This process will terminate as everyone can't have land below the average. Anything above this isn't achievable as if everyone has above the average, the total area of land will not be sufficient.

For the second part, we first sort the array, and calculate prefix and suffix sums (sums of the first i elements on either sides). This overall takes $O(n \log n)$ time. Now for a fixed value of f, we first binary search for f in our array to find how many elements are less than f. The extra land we need is summation of $f-$ every element lower than f. This is just $f\times$ (no of elements lower than f) $-$ the corresponding prefix sum. Now for any value of c, we can also get the extra land we get. We first binary search for c and with the same method, we get corresponding suffix sum - $c\times$ (no of elements higher than c). If the land we get is more than the land we need, the optimal value of c is higher. If we don't get enough extra land, the optimal value of c is lower. So we can binary search for c, and the range for c is from f to the largest element of array. So our final time complexity is $O(n \log n + \log n \log(\text{largest element} - f))$ as we have $O(\log(\text{largest element} - f))$ queries and each query takes $O(\log n)$ time.

8. Walk to 2^k and back to 0, -2^k and back to 0. Do this for every k from 0 until you find the treasure. If the treasure is at N and $2^{n-1} < N \leq 2^n$, we walk a distance of

$4(1+2+4+\cdots+2^n) = 4(2^{n+1}-1) < 16(2^{n-1}) < 16N$, so we walk $O(N)$ distance.