1. $P_A(x) = A_o + A_3x^3 + A_6x^6$ and $P_B(x) = B_o + B_3x^3 + B_6x^6 + B_9x^9$ are really just polynomials $P_A(y) = A_o + A_3y + A_6y^2$ and $P_B(y) = B_o + B_3y + B_6y^2 + B_9y^3$ where y = x^2. Hence multiplying a deg = 3 and deg = 2 polynomial will give a deg = 5 polynomial which requires 6 points to be uniquely defined. Therefore, we evaluate $P_B(y)\ and\ P_A(y)$ at 6 points and then do **exactly 6 pointwise multiplications**.

$$P_A(-2) = A_o - 2A_3 + 4A_6\ ; P_B(-2) = B_o - 2B_3 + 4B_6 - 8B_9$$
$$P_A(-1) = A_o - A_3 + A_6\ ; P_B(-1) = B_o - B_3 + B_6 - B_9$$
$$P_A(0) = A_o\ ; P_B(0) = B_o$$
$$P_A(1) = A_o + A_3 + A_6\ ; P_B(1) = B_o + B_3 + B_6 + B_9$$
$$P_A(2) = A_o + 2A_3 + 4A_6\ ; P_B(2) = B_o + 2B_3 + 4B_6 + 8B_9$$
$$P_A(3) = A_o + 3A_3 + 9A_6\ ; P_B(3) = B_o + 3B_3 + 9B_6 + 27B_9$$

Then evaluate $P_c(y) = P_A(y) * P_B(y)$ and also sub in those same point values for Pc:

$$P_c(-2) = C_o - 2C_1 + 4C_2 - 8C_3 + 16C_4 - 32C_5\ ;$$
$$P_c(-1) = C_o - C_1 + C_2 - C_3 + C_4 - C_5\ ;$$
$$P_c(0) = C_o\ ;$$
$$P_c(1) = C_o + C_1 + C_2 + C_3 + C_4 + C_5\ ;$$
$$P_c(2) = C_o + 2C_1 + 4C_2 + 8C_3 + 16C_4 + 32C_5\ ;$$
$$P_c(3) = C_o + 3C_1 + 9C_2 + 27C_3 + 81C_4 + 243C_5\ ;$$

to get a system of linear equations like that we can solve for $C_5, C_4, C_3, C_1, C_1, C_0$ by inverting it through various row operations.
Then match C_o up to y^(0), $C_1$to y, C_2 to y^2 and so forth.
Finally recall that all these y values represent x^3's so sub back in x^3 to get the final polynomial with only 6 multiplications.

2.

a. Target: (a + bi)(c + di) = ac – bd + (ad+bc)i in 3 real number multiplications.

Start by multiplying (a + b)(c + d) to get ac + bc + ad + bd. Then multiply a*c and b*d. Then subtract ac + bd from the 4-element term which is then scaled by i. The ac and bd are still available to be added as the real part. This looks like ac + ((ac+ad+bc+bd) – ac – bd))i – bd = ac – bd + (ad+bc)i as required.

b. Target: (a + bi) (a + bi) = a^2 + 2abi – b^2 in 2 real number multiplications.

Start with a difference of 2 squares (a + b)(a – b) for a^2 – b^2. Then the second multiplication is ab which is scaled by 2i (not a real number multiplication). These two components are added to give a^2 - b^2 + 2abi as required.

c. Target: (a + bi)^2(c + di)^2 in 5 real number multiplications.

Combine a) and b) together. Start by re-writing the product as (a + bi) (a + bi) (c + di) (c + di) and use commutative multiplication property to obtain ((a + bi) (c + di))^2. Then we can do the inner multiplication in 3 real multiplications as in part a, giving us ((ac – bd) + (ad + cb)i)^2.

Then 2 multiplications as in part a as: ((ac – bd) + (ad + bc)) ((ac – bd) – (ad + bc)) and (ac – bd)(ad + cb) which is scaled by 2i.

This gives (ac – bd)^2 – (ad + bc)^2 + 2(ac – bd)(ad + bc)i in 5 real number multiplications as required.

3.

a. Pad out polynomials A(x) and B(x) with 0 coefficients until they both are of degree $2^p$where $2^p \geq 2n$. Then implement an FFT (black box) on both polynomials, to get their pointwise representations. Then multiply pointwise to get C(x) = A(x)B(x) for all points x that the FFT was evaluated for. You can then do an inverse FFT to separate the powers from the coefficients to get the traditional definition of the polynomial.

b.

i. Note: to uniquely determine an S degree polynomial you need S+1 points. Therefore start by padding all the individual polynomials until they are of degree S, or the next 2^p > S is S|2 != 0, which is an O(K) operation since we do a simple operation for all K polynomials.

Now set up a loop, for(i = 0; i < K; i++){ fft(polynomials [i]); i += 1; } which is O(K*SlogS) since we do FFT on a polynomial of degree S, K times.
Then multiply K polynomials pointwise for S+1 points giving K(S+1) = O(KS).

Finally finish by doing an Inverse FFT on the ONE remaining product polynomial to get the coefficients of the product polynomial in O(SlogS).

Hence the dominating term gives a result of O(KSlogS).

ii. In the case where K is a power of 2, we can get a perfect binary tree. Then we put each polynomial at the leaf nodes of the tree (lets label them a, b, c, d, and so on). Then we pairwise multiply a*b, c*d and so on down the pairs, with the FFT. The complexity of these operations will be (a+b)log(a+b), (c+d)log(c+d), and so on. What is important is since S is the sum of the degrees of every polynomial, any smaller subset of polynomials will have a degree <= S. Therefore for each pairwise multiplication, (x+y)log(x+y) <= (x+y)log(S). Then we repeat this with the resulting polynomials, proceeding up the tree until we complete the final pairwise multiplication, which will be (a+b+c+d+….)log(a+b+c+d+…) where the sum of the variables = S. So the final multiplication is SlogS. Therefore, on each level, the multiplications are O(SlogS) and there are logk levels since we had K polynomials to start, so therefore we get **O(logk*S*logS) as required.**

For a non-perfect binary tree, form the closest perfect binary tree with less than K polynomials. Then we can see that adding the remaining elements only adds one to the height. Then we do the same thing with a height log(k+1) and this is still O(log(k)), so complexity class is maintained.

4.

a. 1. **Predicate**: for Fibonacci numbers, where Fo $= 0$, F1 $= 1$ and Fn $= F_{n-1} + F_{n-2}$:

$$\begin{pmatrix} F_{n+1} & Fn \\ Fn & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

2. **Base Case**: for n $= 1$, $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1$ is true since F2 $=$ F1 $+$ F0 $= 1 + 0 = 1$.

3. **Assume** that the predicate is true for n $=$ k, such that: $\begin{pmatrix} F_{k+1} & Fk \\ Fk & F_{k-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^k$

4. **Inductive step**: attempt to prove for n $=$ k+1, given the assumption that the predicate holds for n $=$ k.

By definition, $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{k+1} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^k \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1$, which we substitute in to obtain:

$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{k+1} = \begin{pmatrix} F_{n+1} & Fn \\ Fn & F_{n-1} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$, which we matrix multiply to obtain:

$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{k+1} = \begin{pmatrix} F_{k+1} + Fk & F_{k+1} \\ Fk + F_{k-1} & F_k \end{pmatrix}$, which we can simplify by the definition of the Fibonacci sequence as: $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{k+1} = \begin{pmatrix} F_{k+2} & F_{k+1} \\ F_{k+1} & F_k \end{pmatrix}$.

Therefore, the predicate holds for k+1 when it holds for k, and so, by induction, the predicate holds for all integers > 0.

b. Since we have the matrix holding Fn+1 in position [0][0], we can obtain Fn by recursing on a sequence of length n-1 and a 2d-array Matrix = $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. This function will recurse by dividing n by 2 until n = 1, where it returns. At the base step we have $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. As we unwind the recurrence, we keep updating Matrix to multiply by itself and so double the powers each time, requiring logn function calls. A caveat to this is if n isn't a multiple of 2. In this case we can multiply by $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ to get the odd power required. Then return the position [0][0] of n-1 recursions for nth value of Fib. If floor division wasn't a feature of the language used, take the floor of the normal division manually.

Example: for n = 8, we pass in 7, halve to get 3 and then again to get 1. For 1 we simply return. For 3 we multiply to get Matrix*Matrix, then adjust by $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ for Matrix^3. Then for 7 we get Matrix^3 * Matrix^3 for Matrix^6 and since n = 7 is not a multiple of 2, adjust by multiplying by $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ to get matrix^7 as desired. Then return Matrix[0][0] to get 8th element.

```
driver(){
    Matrix = ( 1  1 )
             ( 1  0 )
    recursive_multiply (Matrix, n-1);
    return Matrix [0][0];
}
void recursive_multiply (int Matrix [2][2], int n) {
    if (n == 1) return;
    int tmp [2][2] = ( 1  1 )
                     ( 1  0 )
    power (Matrix, n/2);
    multiply (Matrix, Matrix); // O(1) operation of multiplying & adding the correct matrix elements
    if (n%2 != 0) { multiply(Matrix, tmp); }
```

```
}
```

5.

a. *Explanation:*
Take the parameters. Then loop through the array until either is runs out of elements, or you have found an adequate number of leaders. Within the loop, loop the array until you find an element >= T, or the array runs out. If the array ran out, leave the outer loop and return the number of values found (assuming the driver will still want the max achievable leaders for each iteration, even if it is a failed iteration, which doesn't hurt the speed or performance in any way). However, if you found a value, put it in the resulting array at its respective position, and then increment the count of found variables. For the next iteration of the loop, we want to ignore elements within K distance of the found element, so increment by K + 1.

This is a greedy solution that ultimately finds the 3 values appearing earliest in the array that meet the condition. It gives an optimal solution since if there were acceptable values spaced with a distance greater than required, but also an acceptable value at an acceptable closer distance, we could just choose the closer option that is > K away, which won't affect whichever solution was at the next viable index after the 2nd of the pair, since the distance between that second pair is increased by shifting to an earlier left value. Apply this to all pairs to get a greedy, earliest occurring viable solution.

You can see by the fact that we are only looping a single viable ("i") through <= the n elements of the array that this is an O(n) solution to the *decision problem.*

Pseudo:

```
int decisionProblem (int N, int L, int K, int T, int H[], int res[]){
    found = 0;
    i = 0;
    while (found < L and i < N){
            while(H[i] < T and i < N){ i += 1; }
            if(i >= N) break out of loop
            res[found] = H[i];
            found += 1;
            i += K+1;
    }
    return found;
}
```

b. Discovery: if we did the *Decision Problem* for all 'n' T values, and then chose the largest minimum of those, we would run in O(n^2) time. So there needs to be some kind of div and conquer or benefit to sort then search. I note that the largest minimum is the largest value of T from the decision problem that yields a valid result, since any T smaller could just have that same result as the largest if we wanted, this is useful.

Steps:
Copy all N values into another array O(n).
Sort the copied array from smallest to largest with merge sort O(nlogn).
Small optimisation: if we have N elements and need to find L nominations for leaders, there is no point dealing with potential T's in an index position of the sorted array larger than N – L. I.e. a 10 value array, we need 3 values, the smallest can never be in the last 2 positions 8 or 9, the best possible would be values in indices 7, 8, 9 so only consider the decision problem from T = sortedArray[N – L].
Therefore, with 0 as lower bound and N – L as upper bound, do a binary search-like operation **on the originally ordered (not newly sorted) array** but instead of looking for a number, every time the decision problem returns a result, take the upper half again. Note: update a final result array with a tmp result array instead of observing the tmp result since you only want the values for an iteration if a result is actually found, not just for any iteration where upper >= lower.
This way we narrow in on the largest value of T that gives a valid answer to the decision problem, in logn time, and do an O(n) decision problem to see if we get a result each time. So logn*n will give O(nlogn).

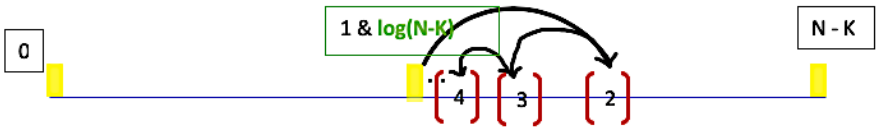
Pseudo:

driver:

```
    sortedArray = copy(H) //O(n)
    sortedArray = mergesort(sortedArray) //O(nlogn)
    int upper = N – L
    int lower = 0
    int finalRes[L]
    while(upper >= lower){
            int mid = (upper + lower)/2
            if(decisionProblem(N, L, K, mid, H[],res[]) < L){
                    upper = mid - 1
            } else {
                    lower = mid + 1
                    finalRes = copy(Res)
            }
    }
    Return/observe finalRes
```

end driver

Importantly, this search will always be O(nlogn), since the optimum solution could correspond to the middle index in the array, but the search will operate until the condition is met, and so looks something like:



Where the first guess is correct, but we still traverse nlogn to realise it is the best solution.