Day 36/180

Binary Search Hard Interview Problem 📑

Aggressive Cows

```
class Solution {
public:

    int solve(int n, int k, vector<int> &stalls)
    {
        // Initialize variables for binary search
        int start = 1, end, mid, ans;

        // Sort the stalls in increasing order
        sort(stalls.begin(),stalls.end());

        // Set the maximum possible distance between stalls
        end = stalls[n-1] - stalls[0];

        // Binary search loop
        while(start <= end)
        {
            // Calculate the middle point
            mid = start + (end - start) / 2;

            // Initialize variables for counting and tracking position
            int count = 1, pos = stalls[0];

            // Iterate through the stalls
            for(int i = 1; i < n; i++)
            {
                  // Check if the current stall can accommodate a cow with
distance 'mid'
                if(pos + mid <= stalls[i])
                {
                        // If yes, increment the count and update the
position
                    count++;
                    pos = stalls[i];
                }
```

```
            }

            // Adjust the search space based on the count of cows placed
            if (count < k)
            {
                  // If fewer cows are placed than required, reduce the
search space
                end = mid - 1;
            }
            else
            {
                // If enough or more cows are placed, update the answer
and expand the search space
                ans = mid;
                start = mid + 1;
            }
        }

        // Return the largest possible minimum distance between cows
        return ans;
    }
};
```

Code Explanation and Complexity of Aggressive Cows

1. Binary Search Approach:
    - The code uses binary search to efficiently find the largest possible minimum distance between cows.
    - Binary search is performed on the possible distances between stalls.
2. Sorting Stalls:
    - The input array stalls are sorted in increasing order.
    - Sorting is essential for the binary search to work correctly.
3. Setting Initial Search Space:
    - Initialise start and end for binary search.
    - start is set to 1, and end is set to the maximum possible distance between the last and first stall.
4. Binary Search Loop:
    - The binary search loop runs until the start becomes greater than the end.
    - In each iteration, the middle point mid is calculated.
5. Counting Cows:
    - For each mid, a loop iterates through the sorted stalls to determine how many cows can be accommodated with a minimum distance of mid.

- The variable count keeps track of the number of cows placed.
- The variable pos keeps track of the position of the last placed cow.

6. Adjusting Search Space:
   - If the count of placed cows (count) is less than the required number of cows (k), it means the distance mid is too large.
   - Therefore, adjust the search space by setting end = mid - 1.
   - If enough or more cows can be placed, update the answer (ans = mid) and expand the search space by setting start = mid + 1.
7. Returning Result:
   - Once the binary search loop completes, the function returns the largest possible minimum distance between cows (ans).
8. Time and Space Complexity:
   - The time complexity is O(n*log(n)) due to the binary search and sorting.
   - The space complexity is O(1) since the algorithm uses a constant amount of additional space.

Koko Eating Bananas

```
class Solution
{
     public: int minEatingSpeed(vector<int> &piles, int h)
     {
          int start = 0, end = 0, mid, ans, n = piles.size();
          long long sum = 0;
          // Calculate sum of all piles to find average eating speed, and
find largest pile
          for (int i = 0; i < n; i++)
          {
               sum += piles[i];
               end = max(end, piles[i]);// Keep track of the largest pile
          }

          // Initiate the eating speed to average bananas per hour, but
at least 1
          start = sum / h;
          if (!start)
               start = 1;

          // Implement binary search to find optimal bananas-per-hour
eating speed
          while (start <= end)
          {
```

```
            // Find the mid point which will be used as a potential
eating speed
            mid = start + (end - start) / 2;
            int total_time = 0;
            // Calculate the total time needed to consume all piles
with the mid eating speed
            for (int i = 0; i < n; i++)
            {
                total_time += piles[i] / mid; // Regular hours
                if (piles[i] % mid)
                    total_time++;  // Adding one more hour if there
are remaining bananas in current pile
            }

            // If the total_time exceeds allowed time, we need to
increase eating speed
            if (total_time > h)
            {
                start = mid + 1;
            }
            else
            {
                // If time is within limit, this eating speed is a
potential answer
                // But we will continue to find the minimum possible
speed
                ans = mid;
                end = mid - 1;
            }
        }

        // Return the minimum eating speed found that allow to consume
all bananas within the limit
        return ans;
    }
};
```

Code Explanation and Complexity of Koko Eating Bananas

1. Binary Search Approach:
    - The code uses binary search to find the minimum eating speed (k) required for Koko to finish eating all bananas within h hours.

2. Initial Guess:
    - Calculate the sum of all bananas and find the maximum pile size to set the initial search space.
    - Set the initial guess for eating speed (start) based on the average bananas per hour, ensuring it's not zero.
3. Binary Search Loop:
    - While the search space (start to end) is valid, perform binary search iterations.
4. Total Time Calculation:
    - Calculate the total time required to eat all bananas at the current speed (mid).
    - Iterate through each pile, determining how many hours are needed to consume the bananas.
5. Adjusting Search Space:
    - If the total time is greater than h, it means the eating speed is too slow. Adjust the search space by increasing the speed (start = mid + 1).
    - If the total time is within the allowed hours, update the answer (ans = mid) and decrease the speed (end = mid - 1).
6. Returning Result:
    - Once the binary search loop completes, return the minimum eating speed (ans).
7. Time Complexity:
    - The time complexity is O(n log m), where n is the number of piles and m is the maximum size of a pile.
8. Space Complexity:
    - The space complexity is O(1).

Magnetic Force Between Two Balls

```
class Solution {
public:

     // function to check whether it is possible to place m balls such that
    // the minimum distance between any two is greater or equal to limit
    bool solve(vector<int>& position, long long m,long long limit)
    {
        long long k=1; // first ball is placed in the first basket
        long long prev=position[0]; // position of the first ball
        for(long long i=1;i<position.size();i++)
        {
               // if the distance from the current position to the last placed ball is greater or equal to limit
            // then we can place a new ball in the current position
```

```
            if(position[i]-prev>=limit)
            {
                k++;
                prev=position[i];
            }
            // if all balls have been placed successfully return true
            if(k>=m) return 1;
        }
        // return false in case some balls couldn't be placed properly
        return 0;
    }

    // function to find the maximum possible minimum distance between any
two balls
    long long maxDistance(vector<int>& position, long long m) {
        long long i=0;
        long long j=0;
        for(auto it:position) j+=it;

            // sorting to ensure the balls are placed in the positions
incrementally
        sort(position.begin(),position.end());
        long long ans=0;

        // binary search between 0 to maximum possible value of j
        while(i<=j)
        {
            long long mid=(i+j)/2;
            // if it is possible to place the balls such that
            // the minimum distance between any two is equal to mid
             // search for possibly greater valid kinds by moving towards
right in the search space
            // and update the answer
            if(solve(position,m,mid))
            {
                i=mid+1;
                ans=mid;
            }
             // if it is not possible to place the balls as required for
current mid
             // search for possibly valid mids by moving towards left in
the search space
            else j=mid-1;
        }
```

```
        return ans;
    }
};
```

Code Explanation and Complexity of Koko Eating Bananas

1. Binary Search Approach:
    - The code uses binary search to find the maximum minimum magnetic force between two balls.
    - Binary search is performed on the possible values of the minimum magnetic force.
2. Helper Function (solve):
    - The solve function checks if it's possible to distribute balls with a minimum magnetic force of at least 'limit'.
    - It iterates through the positions and checks if the distance between consecutive positions is at least 'limit'.
    - If enough balls can be placed with the specified minimum force, it returns true; otherwise, it returns false.
3. Main Function:
    - Calculate the sum of all positions (j).
    - Sort the positions in increasing order to facilitate binary search.
    - Initialize ans to store the maximum minimum magnetic force.
4. Binary Search Loop:
    - While the search space (i to j) is valid, perform binary search iterations.
5. Checking Possibility:
    - Use the solve function to check if it's possible to distribute balls with a minimum magnetic force of at least the current mid.
    - If possible, update the answer (ans = mid) and increase the search space (i = mid + 1).
    - If not possible, reduce the search space (j = mid - 1).
6. Returning Result:
    - Once the binary search loop completes, return the maximum minimum magnetic force (ans).
7. Time Complexity:
    - The time complexity is O(n log M), where n is the number of positions and M is the maximum possible distance between positions.
    - The binary search has a time complexity of O(log M), and for each binary search iteration, the solve function iterates through the positions, resulting in O(n) per iteration.
8. Space Complexity:
    - The space complexity is O(1).

____ END ____