[875. Koko Eating Bananas]

this is an interesting problem where I discovered a very important technique of combining binary search to eliminate some iterative process, convert n => log(n).

This problem states that, Koko the monkey, eats bananas from piles. There is a guard who remains absent for a number of hours when Koko can eat bananas. Koko cannot eat banana from more than one pile in an hour. A pile contains x number of bananas. We have to find the minimum number of banana-eating-per-hour which can ensure that, koko eats all the bananas before the guard comes in after h hours.

So let us look at this problem, there are n number of piles containing n_x number of bananas (pile x contains n_x number of bananas). Koko can eat less or equal to the total number of bananas present in a pile in an hour. The minimal number of hour needed to eat all the bananas should be n.

So h must be equal or greater than n. If k (banana eating speed per hour) is equal to max value of n_x in all the piles then, it takes h_min or n hours to finish all bananas.

So let's say, piles = {2,4,3,8} h = 4. If k = 8 (equal to the max number of bananas in a pile) Koko will eat one pile banana in an hour equaling to 4 hours. So k_max = max(piles_number). k_min = 1, because say h = 100, therefore koko can eat 1 banana an hour and finish all the bananas before time. Therefore, we got a range of k from 1 to piles_max_number. Now we can run a brute force to compute hours needed for all values of k from 1 to max_piles_number, but that will be a linear search, will take O(max_value_of_pile*n), but this can be eliminated with a binary search putting low = 1, high = max_value_of_piles and computing hrs_needed for the values of k.
If (k<=hrs) we set result is equal to k and go even left to find if a smaller result exists, else we go to the right side, as if we increase k, we are decreasing h, therefore this approach works because of the fact that, increasing k results in decreasing hrs h needed from the function. Therefore binary search works in this case instead of a regular linear search because with this technique as we already know that, h is decreasing if we increase k, we find some sort of sorting here due to the inner function inside the binary search for computing the hrs needed. The code is as follows:

```
class Solution {
public:
    unsigned long long int hrsneeded(vector<int>& piles, int k){
        unsigned long long int sum = 0;
        for(int i=0;i<piles.size();i++){
            sum+=ceil((double)piles[i]/k);
        }
        return sum;
    }
    int minEatingSpeed(vector<int>& piles, int h) {
        int mini = 1;
        int maxi = *max_element(piles.begin(),piles.end());
        int res;
        while(mini <= maxi) {
            int mid = mini + (maxi-mini)/2;
            unsigned long long int hrs = hrsneeded(piles,mid);
            if(hrs <= h){
                res = mid;
                maxi = mid - 1;
            }
            else if(hrs>h){
                mini = mid+1;
            }
        }
        return res;
    }
};
```