DAY 33/180

Q1- Allocate Minimum Number of Pages

```
bool check(int arr[],int N,int M,int mid){
    int sum =0,stu=1;
    for(int i=0;i<N;i++){
        sum += arr[i];
        // If the sum exceeds the current minimum pages 'mid', assign a new student and reset the sum.
        if(sum>mid){
            stu++;
            sum=arr[i];
        }
        // If the number of students required exceeds 'M', it's not possible, so return false.
        if(stu>M) return 0;
    }
    return 1;
}
// Function to find the minimum number of pages for 'M' students to allocate 'N' books.
int findPages(int A[], int N, int M){
    if(N<M) return -1;
    int s=0,e=0,ans=-1;
    for(int i=0;i<N;i++){
        s=max(s,A[i]);
        e+=A[i];
    }
    // Perform binary search to find the minimum pages required for allocation.
    while(s<=e){
        int mid=(s+e)/2;
        // Check if it's possible to allocate books with 'mid' pages per student.
        if(check(A,N,M,mid)==1){
            ans=mid;
            e=mid-1;// Adjust the end value to search for smaller 'mid' values.
        }
        else{
            s=mid+1;// Adjust the start value to search for larger 'mid' values.
        }
    }
    return ans;
}
```

Q2- The Painter’s Partition Problem

```cpp
bool isPossible(int boards[], long long mid, int k, int n) {
    long long sum = 0;
    int man = 1;
    for (int i = 0; i < n; i++) {
        if ((long long)boards[i] + sum <= mid) {
            sum += boards[i];  // Add the length of the current board to the sum.
        } else {
            man++;             // If adding the current board exceeds 'mid', assign a new painter.
            sum = boards[i];  // Reset the sum to the length of the current board.
        }
    }
    // If the number of painters required is less than or equal to 'k', it's possible, so return false.
    if (man <= k) return false;
    // If the number of painters required exceeds 'k', it's not possible, so return true.
    return true;
}
long long minTime(int boards[], int n, int k) {
    long long s = 0;  // Initialize the start time.
    long long e = 0;  // Initialize the end time.

    for (int i = 0; i < n; i++) {
        s = max(s, (long long)boards[i]);  // Set the start time to the maximum board length.
        e += boards[i];                    // Set the end time to the sum of all board lengths.
    }

    // Perform binary search to find the minimum time required.
    while (s <= e) {
        long long mid = s + (e - s) / 2;  // Calculate the middle time.

        // Check if it's possible to distribute boards within 'mid' time.
        if (isPossible(boards, mid, k, n)) {
            s = mid + 1;  // Adjust the start time to search for longer time.
        } else {
            e = mid - 1;  // Adjust the end time to search for shorter time.
        }
    }
    return s;  // Return the minimum time required to paint all the boards.
}
```

Q3- Capacity to ship Packages within D Days

```cpp
bool check(int mid, int days, vector<int>& arr) {
    int n = arr.size();
    int sum = 0;
    int cnt = 1;
    for (int i = 0; i < n; i++) {
        sum += arr[i];
        // If the sum exceeds the 'mid' capacity, assign a new day (cnt) and reset the sum.
        if (sum > mid) {
            cnt++;
            sum = arr[i];
        }
    }
    // If the number of days required is less than or equal to 'days', it's possible, so return true.
    return cnt <= days;
}
// Function to find the minimum weight capacity required to ship all weights within 'days'.
int shipWithinDays(vector<int>& weights, int days) {
    int n = weights.size();
    int s = *max_element(weights.begin(), weights.end());  // Initialize the start capacity.
    int ans = 1e9;  // Initialize the answer.
    int e = accumulate(weights.begin(), weights.end(), 0);  // Initialize the end capacity.

    // Perform binary search to find the minimum weight capacity required.
    while (s <= e) {
        int mid = (s + e) / 2;  // Calculate the middle capacity.

        // Check if it's possible to ship all weights within 'days' using the given 'mid' capacity.
        if (check(mid, days, weights)) {
            ans = mid;  // Update the answer to the current 'mid'.
            e = mid - 1;  // Adjust the end capacity to search for smaller 'mid' values.
        } else {
            s = mid + 1;  // Adjust the start capacity to search for larger 'mid' values.
        }
    }
    return ans;  // Return the minimum weight capacity required to ship all weights within 'days'.
}
```

Q4- Koko Eating Bananas.

```cpp
#define ll long long
    bool check(ll mid, vector<int>& piles, ll h) {
        int n = piles.size();
        ll time = 0;
        for (int i = 0; i < n; i++) {
            if (piles[i] < mid) {
                time++;
            } else {
                ll t = piles[i] / mid;
                if (piles[i] % mid) t++;
                time += t;
            }
        }
        // If the time required is less than or equal to 'h', it's possible, so return true.
        return time <= h;
    }
    // Function to find the minimum eating speed required to eat all piles within 'h' hours.
    int minEatingSpeed(vector<int>& piles, int h) {
        int n = piles.size();
        ll s = 1;          // Initialize the start speed.
        ll e = 1e9 + 1;    // Initialize the end speed.
        int ans = -1;       // Initialize the answer.
        // Perform binary search to find the minimum eating speed required.
        while (s <= e) {
            int mid = (e + s) >> 1;  // Calculate the middle speed.
            // Check if it's possible to eat all piles within 'h' hours using the given 'mid' speed.
            if (check(mid, piles, h)) {
                ans = mid;        // Update the answer to the current 'mid'.
                e = mid - 1;      // Adjust the end speed to search for smaller 'mid' values.
            } else {
                s = mid + 1;      // Adjust the start speed to search for larger 'mid' values.
            }
        }
        return ans;
    }
```

Q5- Split Array Largest Sum

```
int splitArray(int a[], int n, int k) {
    int l = 0;            // Initialize the left boundary for binary search.
    int r = 1e9;          // Initialize the right boundary for binary search.
    int ans = 1;          // Initialize the answer.
    int sum = 0;          // Initialize the sum of elements.

    // Perform binary search as long as the left boundary is less than or equal to the right boundary.
    while (l <= r) {
        int m = (l + r) / 2;  // Calculate the middle value.
        int cnt = 0;          // Initialize the count of subarrays.
        sum = 0;              // Reset the sum.

        for (int i = 0; i < n; i++) {
            if (sum + a[i] > m) {
                cnt++;  // Increment the count of subarrays.
                sum = a[i];  // Reset the sum to the current element.

                // If the current element is larger than 'm', it's not possible, set cnt to a large value.
                if (a[i] > m) {
                    cnt = INT_MAX;
                    break;
                }
            } else {
                sum += a[i];  // Add the current element to the sum.
            }
        }
        cnt++;  // Increment the count for the last subarray.

        if (cnt <= k) {
            ans = m;  // Update the answer to the current 'm'.
            r = m - 1;  // Adjust the right boundary to search for smaller 'm' values.
        } else {
            l = m + 1;  // Adjust the left boundary to search for larger 'm' values.
        }
    }
    return ans;  // Return the minimum value 'm' that satisfies the condition.
}
```