

Lesson Plan

Java Binary Search

What & Why?

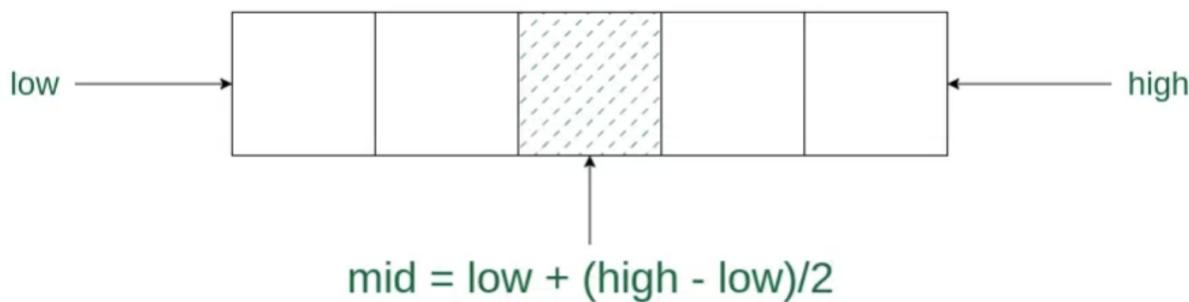
Binary search is an efficient algorithm used to search for a specific element within a sorted array or list. It works by repeatedly dividing the search interval in half until the target element is found or the interval becomes empty. Here's how it works:

Requirement: The array must be sorted in ascending or descending order for binary search to work effectively.

Algorithm:

In this algorithm,

1. Divide the search space into two halves by finding the middle index "mid".
2. Compare the middle element of the search space with the key.
3. If the key is found at middle element, the process is terminated.
4. If the key is not found at middle element, choose which half will be used as the next search space.
5. If the key is smaller than the middle element, then the left side is used for next search.
6. If the key is larger than the middle element, then the right side is used for next search.
7. This process is continued until the key is found or the total search space is exhausted.



Code:

```
int binarySearch(int arr[], int x)
{
    int l = 0, r = arr.length - 1;
    while (l <= r) {
        int m = l + (r - l) / 2;

        // Check if x is present at mid
        if (arr[m] == x)
            return m;

        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;
    }
}
```

```

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // If we reach here, then element was
    // not present
    return -1;
}

```

Q: Binary Search [Leetcode 704]

Given an array of integers `nums` which is sorted in ascending order, and an integer `target`, write a function to search `target` in `nums`. If `target` exists, then return its index. Otherwise, return `-1`.

You must write an algorithm with $O(\log n)$ runtime complexity.

Input: `nums = [-1,0,3,5,9,12]`, `target = 9`

Output: 4

Explanation: 9 exists in `nums` and its index is 4

Code:

```

class Solution {
    public int search(int[] nums, int target) {
        int left = 0; // initialize left pointer to 0
        int right = nums.length - 1; // initialize right
pointer to the last index of the array

        while (left <= right) { // continue the loop till
left pointer is less than or equal to right pointer
            int mid = left + (right - left) / 2; // calculate the middle index of the array

            if (nums[mid] == target) { // check if the
middle element is equal to target
                return mid; // return the middle index
            } else if (nums[mid] < target) { // check if the
middle element is less than target
                left = mid + 1; // move the left pointer to
the right of middle element
            } else { // if the middle element is greater
than target
                right = mid - 1; // move the right pointer
to the left of middle element
            }
        }

        return -1; // target not found in the array
    }
}

```

Explanation: Simply use binary search as discussed above to find the target element

Time Complexity: $O(\log n)$, because of binary search(each time we divide the array in half)

Space: $O(1)$

Q: Given a sorted integer array and an integer 'x', find the lower bound of x.

Input : 4 6 10 12 18 18 20 20 30 45

Output: lower_bound for element 18 at index 4

```

static int lower_bound(int array[], int key)
{
    // Initialize starting index and
    // ending index
    int low = 0, high = array.length;
    int mid;

    // Till high does not crosses low
    while (low < high) {

        // Find the index of the middle element
        mid = low + (high - low) / 2;

        // If key is less than or equal
        // to array[mid], then find in
        // left subarray
        if (key ≤ array[mid]) {
            high = mid;
        }

        // If key is greater than array[mid],
        // then find in right subarray
        else {

            low = mid + 1;
        }
    }

    // If key is greater than last element which is
    // array[n-1] then lower bound
    // does not exists in the array
    if (low < array.length && array[low] < key) {
        low++;
    }

    // Returning the lower_bound index
    return low;
}

```

Explanation: Initialize the low as 0 and high as N.

Compare key with the middle element($\text{arr}[\text{mid}]$)

If the middle element is greater than or equal to the key then update the high as a middle index(mid).

Else update low as mid + 1.

Repeat step 2 to step 4 until low is less than high.

After all the above steps the low is the lower_bound of a key in the given array.

Time Complexity: $O(\log N)$, binary search

Auxiliary Space: $O(1)$, no extra space being used

If you use C++, it has a build in builtin function to find lower bound using binary search

`lower_bound(arr.begin(), arr.end(), target)`

Q: Given a sorted integer array and an integer 'x', find the upper bound of x.

Input : 10 20 30 30 40 50

Output : upper_bound for element 30 is 40 at index 4

Code:

```
static void upper_bound(int arr[], int key)
{
    int mid, N = arr.length;

    // Initialise starting index and
    // ending index
    int low = 0;
    int high = N;

    // Till low is less than high
    while (low < high && low != N) {
        // Find the index of the middle element
        mid = low + (high - low) / 2;

        // If key is greater than or equal
        // to arr[mid], then find in
        // right subarray
        if (key ≥ arr[mid]) {
            low = mid + 1;
        }

        // If key is less than arr[mid]
        // then find in left subarray
        else {
            high = mid;
        }
    }
}
```

```

    // If key is greater than last element which is
    // array[n-1] then upper bound
    // does not exists in the array
    if (low == N) {
        System.out.print("The upper bound of " + key + "
does not exist.");
        return;
    }

    // Print the upper_bound index
    System.out.print("The upper bound of " + key + "
is " + arr[low] + " at index " + low);
}

```

Explanation: Sort the array before applying binary search.

Initialize low as 0 and high as N.

Find the index of the middle element (mid)

Compare key with the middle element(arr[mid])

If the middle element is less than or equals to key then update the low as mid+1, Else update high as mid.

Repeat step 2 to step 4 until low is less than high.

After all the above steps the low is the upper_bound of the key

Time Complexity: $O(\log N)$,binary search

Auxiliary Space: $O(1)$, no extra space being used

If you use C++, it has a build in builtin funciton to find upper bound using binary search
`upper_bound(arr.begin(), arr.end(), target)`

Q: Given a sorted array of n elements and a target 'x'. Find the first occurrence of 'x' in the array. If 'x' does not exist return -1.

```

static int firstOccurrence(int array[], int key)
{
    int x=lower_bound(array[],key)
    if(array[x]==key) return x;
    else return -1;
}

```

Explanation: Find the lower bound then check if the element at index lower bound equals the key the return the index, else it means the element doesn't exits

Time Complexity: $O(\log N)$,binary search,in lower_bound

Auxiliary Space: $O(1)$, no extra space being used

Q: Peak index in mountain array (leetcode 852)

An array arr is a mountain if the following properties hold:

```

arr.length ≥ 3
There exists some i with 0 < i < arr.length - 1 such that:
arr[0] < arr[1] < ... < arr[i - 1] < arr[i]
arr[i] > arr[i + 1] > ... > arr[arr.length - 1]
Given a mountain array arr, return the index i such that
arr[0] < arr[1] < ... < arr[i - 1] < arr[i] > arr[i + 1] >
... > arr[arr.length - 1].

```

You must solve it in $O(\log(\text{arr.length}))$ time complexity.

Code:

```

class Solution {
    public int peakIndexInMountainArray(int[] arr) {
        int start = 0;
        int end = arr.length - 1;

        while(start < end){
            int mid = start + (end - start) / 2;
            if (arr[mid] > arr[mid+1]) {
                end = mid;
            } else {
                // you are in ascending part of array
                start = mid + 1;
                // because we know that mid+1 element > mid
            }
        }
        above

        return start;
    }
}

```

Explanation:

Binary Search Approach: Helps find the maximum element in a rotated sorted array.

Comparing Midpoints: Check if the middle element is larger than its next element.

Deciding the Search Direction:

- If it's larger, explore the left side for a potentially larger element.
- If not, explore the right side.

Refinement of Pointers:

- `start` and `end` pointers keep the best possible answer found so far.
- When they converge to one element, it's likely the maximum.

Determining the Maximum: Return either `start` or `end` as they point to the maximum due to previous comparisons.

Time Complexity: $O(\log N)$, binary search

Auxiliary Space: $O(1)$, no extra space being used

Ques: Search in Rotated Sorted Array (leetcode 33)

There is an integer array nums sorted in ascending order (with distinct values).

Prior to being passed to your function, nums is possibly rotated at an unknown pivot index k ($1 \leq k < \text{nums.length}$) such that the resulting array is $[\text{nums}[k], \text{nums}[k+1], \dots, \text{nums}[\text{n}-1], \text{nums}[0], \text{nums}[1], \dots, \text{nums}[k-1]]$ (0-indexed). For example, $[0,1,2,4,5,6,7]$ might be rotated at pivot index 3 and become $[4,5,6,7,0,1,2]$.

Given the array nums after the possible rotation and an integer target, return the index of target if it is in nums, or -1 if it is not in nums.

You must write an algorithm with $O(\log n)$ runtime complexity.

Input : arr[] = {5, 6, 7, 8, 9, 10, 1, 2, 3}, key = 3

Output : Found at index 8

Code:

```
static int search(int arr[], int l, int h, int key)
{
    if (l > h)
        return -1;

    int mid = (l + h) / 2;
    if (arr[mid] == key)
        return mid;

    /* If arr[l...mid] first subarray is sorted */
    if (arr[l] <= arr[mid]) {
        /* As this subarray is sorted, we
           can quickly check if key lies in
           half or other half */
        if (key >= arr[l] && key <= arr[mid])
            return search(arr, l, mid - 1, key);
        /*If key not lies in first half subarray,
        Divide other half into two subarrays,
        such that we can quickly check if key lies
        in other half */
        return search(arr, mid + 1, h, key);
    }
}
```

```

/* If arr[l..mid] first subarray is not sorted,
   then arr[mid... h] must be sorted subarray*/
if (key ≥ arr[mid] && key ≤ arr[h])
    return search(arr, mid + 1, h, key);

return search(arr, l, mid - 1, key);
}

```

Explanation: The idea is to create a recursive function to implement the binary search where the search region is $[l, r]$. For each recursive call:

We calculate the mid value as $mid = (l + h) / 2$

Then try to figure out if l to mid is sorted, or $(mid+1)$ to h is sorted

Based on that decide the next search region and keep on doing this till the element is found or l overcomes h .

Time Complexity: $O(\log N)$. Binary Search requires $\log n$ comparisons to find the element. So time complexity is $O(\log n)$.

Auxiliary Space: $O(1)$. As no extra space is required.

Q: Find K Closest Elements (leetcode 658)

Given a sorted integer array arr, two integers k and x, return the k closest integers to x in the array. The result should also be sorted in ascending order.

An integer a is closer to x than an integer b if:

$|a - x| < |b - x|$, or

$|a - x| == |b - x|$ and $a < b$

Input: arr = [1,2,3,4,5], k = 4, x = 3

Output: [1,2,3,4]

Code:

```

import java.util.ArrayList;
import java.util.List;

public class ClosestElements {
    public List<Integer> findClosestElements(int[] arr, int
k, int x) {
        int n = arr.length;
        int R = lowerBound(arr, x); // Find the lower bound
        using a similar method as in C++
        int L = R - 1;

```

```

        // Expand the [L, R] window till its size becomes
        equal to k
        while (k > 0) {
            if (R >= n || (L >= 0 && x - arr[L] <= arr[R] -
x)) {
                L--; // Expand from left
            } else {
                R++; // Expand from right
            }
            k--;
        }

        List<Integer> result = new ArrayList<>();
        for (int i = L + 1; i < R; i++) {
            result.add(arr[i]);
        }
        return result;
    }

    // Implementing lower bound function
    private int lowerBound(int[] arr, int target) {
        int low = 0;
        int high = arr.length;

        while (low < high) {
            int mid = low + (high - low) / 2;

            if (arr[mid] < target) {
                low = mid + 1;
            } else {
                high = mid;
            }
        }

        return low;
    }

    public static void main(String[] args) {
        ClosestElements solution = new ClosestElements();
        int[] arr = {1, 2, 3, 4, 5};
        int k = 4;
        int x = 3;

        List<Integer> closest =
solution.findClosestElements(arr, k, x);
        System.out.println("Closest elements to " + x + "
are: " + closest);
    }
}

```

Explanation: We can take advantage of the fact that input array given to us is already sorted. We can use binary search to find the smallest element in arr which is greater or equal to x. Let's mark this index as R. Let's mark the index preceding to R as L and element at this index will be smaller than x.

Now, [L, R] forms our window of elements closest to x. We have to expand this window to fit k elements. We will keep picking either arr[L] or arr[R] based on whichever's closer to x and expand our window till it contains k elements.

Time Complexity: $O(\log n + k)$, We need $O(\log n)$ time complexity to find r at the start. Then we need another $O(k)$ time to expand our window to fit k elements

Space Complexity: $O(1)$

Q: Sum of Square Numbers (Leetcode 633)

Given a non-negative integer c, decide whether there're two integers a and b such that $a^2 + b^2 = c$.

Input: c = 5

Output: true

Explanation: $1 * 1 + 2 * 2 = 5$

Code:

```
class Solution {
    public boolean judgeSquareSum(int c)
    {
        long l = 0, h = (long) Math.sqrt(c);
        while (l <= h)
        {
            long cur = l * l + h * h;
            if (cur < c)
            {
                l++;
            }
            else if (cur > c)
            {
                h--;
            }
            else
            {
                return true;
            }
        }
        return false;
    }
}
```

Explanation: Binary Search the given element in which up to the given number if the square number is greater than the given number decrement the high and otherwise increment the low and in the else condition be true if there are equal, otherwise return false.

Time complexity: $O(\sqrt{c} \log(c))$, outer loop \sqrt{c} , $\log(c)$ for binary search

Space: $O(1)$

Q: Median of Two Sorted Arrays (leetcode 4)

Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays.

The overall run time complexity should be $O(\log(m+n))$.

Input: nums1 = [1,3], nums2 = [2]

Output: 2.00000

Explanation: merged array = [1,2,3] and median is 2.

Code:

```
static double Median(int[] A, int[] B)
{
    int n = A.length;
    int m = B.length;
    if (n > m)
        return Median(B,
                      A); // Swapping to make A smaller

    int start = 0;
    int end = n;
    int realmidinmergedarray = (n + m + 1) / 2;

    while (start ≤ end) {
        int mid = (start + end) / 2;
        int leftAsize = mid;
        int leftBsize = realmidinmergedarray - mid;
        int leftA
            = (leftAsize > 0)
                ? A[leftAsize - 1]
                : Integer
                    .MIN_VALUE; // checking overflow
                    // of indices
        int leftB = (leftBsize > 0) ? B[leftBsize - 1]
                                    : Integer.MIN_VALUE;
        int rightA = (leftAsize < n)
            ? A[leftAsize]
            : Integer.MAX_VALUE;
        int rightB = (leftBsize < m)
            ? B[leftBsize]
            : Integer.MAX_VALUE;
```

```

// if correct partition is done
if (leftA <= rightB && leftB <= rightA) {
    if ((m + n) % 2 == 0)
        return (Math.max(leftA, leftB)
            + Math.min(rightA, rightB))
            / 2.0;
    return Math.max(leftA, leftB);
}
else if (leftA > rightB) {
    end = mid - 1;
}
else
    start = mid + 1;
}
return 0.0;
}

```

Explanation: The given two arrays are sorted, so we can utilize the ability of Binary Search to divide the array and find the median.

Median means the point at which the whole array is divided into two parts. Hence since the two arrays are not merged so to get the median we require merging which is costly.

Hence instead of merging, we will use a modified binary search algorithm to efficiently find the median.

Time Complexity: $O(\min(\log M, \log N))$: Since binary search is being applied on the smaller of the 2 arrays

Auxiliary Space: $O(1)$, no extra space use

Q: Given a sorted array of non-negative distinct integers, find the smallest missing non-negative element in it.

Input: {0, 1, 2, 6, 9}, n = 5, m = 10

Output: 3

Input: {4, 5, 10, 11}, n = 4, m = 12

Output: 0

Code:

```

public class SmallestMissingNumber {
    public int findSmallestMissing(int[] arr) {
        int left = 0;
        int right = arr.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

```

```

        // If the element at mid doesn't match its index
        and the previous element is not missing
        if (arr[mid] ≠ mid && (mid = 0 || arr[mid - 1]
= mid - 1)) {
            return mid;
        }

        // If the element at mid matches its index, the
        missing element is on the right side
        if (arr[mid] = mid) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    // If no missing element is found, the smallest
    missing element is at the end
    return arr.length;
}

public static void main(String[] args) {
    SmallestMissingNumber solution = new
SmallestMissingNumber();
    int[] arr = {0, 1, 2, 3, 4, 7, 10};

    int smallestMissing =
solution.findSmallestMissing(arr);
    System.out.println("Smallest missing number is: " +
smallestMissing);
}
}

```

Explanation: This solution utilizes a modified binary search algorithm:

It checks if the middle element matches its index and if the previous element is not missing (index-wise). If the condition fails, the missing element is in the left half of the array; otherwise, it's in the right half. The search continues in the respective half until the smallest missing number is found.

Time Complexity: $O(\log n)$, for binary search

Auxiliary Space : $O(1)$, no extra space used

Q: $\text{Sqrt}(x)$ (leetcode 69)

Given a non-negative integer x , return the square root of x rounded down to the nearest integer. The returned integer should be non-negative as well.

You must not use any built-in exponent function or operator.

Input: x = 4

Output: 2

Explanation: The square root of 4 is 2, so we return 2.

Code:

```
public static int floorSqrt(int x)
{
    // Base Cases
    if (x == 0 || x == 1)
        return x;

    // Do Binary Search for floor(sqrt(x))
    long start = 1, end = x / 2, ans = 0;
    while (start <= end) {
        long mid = (start + end) / 2;

        // If x is a perfect square
        if (mid * mid == x)
            return (int)mid;

        // Since we need floor, we update answer when
        // mid*mid is smaller than x, and move closer to
        // sqrt(x)
        if (mid * mid < x) {
            start = mid + 1;
            ans = mid;
        }
        else // If mid*mid is greater than x
            end = mid - 1;
    }
    return (int)ans;
}
```

Explanation: The idea is to find the largest integer i whose square is less than or equal to the given number. The values of $i * i$ is monotonically increasing, so the problem can be solved using binary search.

Base cases for the given problem are when the given number is 0 or 1, then return X;

Create some variables, for storing the lower bound say l = 0, and for upper bound r = x / 2 (i.e, The floor of the square root of x cannot be more than x/2 when x > 1).

Run a loop until l <= r, the search space vanishes

Check if the square of mid ($mid = (l + r)/2$) is less than or equal to X, If yes then search for a larger value in the second half of the search space, i.e l = mid + 1, update ans = mid

Else if the square of mid is more than X then search for a smaller value in the first half of the search space, i.e r = mid - 1

Finally, Return the ans

Time Complexity: $O(\log(X))$, Binary search

Auxiliary Space: $O(1)$, no extra space being used

Q: Capacity to ship packages within D days (leetcode 1011)

A conveyor belt has packages that must be shipped from one port to another within days days.

The i th package on the conveyor belt has a weight of $\text{weights}[i]$. Each day, we load the ship with packages on the conveyor belt (in the order given by weights). We may not load more weight than the maximum weight capacity of the ship.

Return the least weight capacity of the ship that will result in all the packages on the conveyor belt being shipped within days days.

Input: $\text{weights} = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$, $\text{days} = 5$

Output: 15

Explanation: A ship capacity of 15 is the minimum to ship all the packages in 5 days like this:

1st day: 1, 2, 3, 4, 5

2nd day: 6, 7

3rd day: 8

4th day: 9

5th day: 10

Note: That the cargo must be shipped in the order given, so using a ship of capacity 14 and splitting the packages into parts like (2, 3, 4, 5), (1, 6, 7), (8), (9), (10) is not allowed.

Code:

```
public int shipWithinDays(int[] weights, int D) {
    int left = 0, right = 0;
    for (int w: weights) {
        left = Math.max(left, w);
        right += w;
    }
    while (left < right) {
        int mid = (left + right) / 2, need = 1, cur = 0;
        for (int w: weights) {
            if (cur + w > mid) {
                need += 1;
                cur = 0;
            }
            cur += w;
        }
        if (need > D) left = mid + 1;
        else right = mid;
    }
    return left;
}
```

Explanation:

Given the number of bags,
return the minimum capacity of each bag,
so that we can put items one by one into all bags.

We binary search the final result.

The left bound is $\max(A)$,
The right bound is $\sum(A)$.

Time Complexity: $O(n * \log n)$. We perform $\log n$ analyses, and for each we process n packages.

Auxiliary Space: $O(1)$, no extra space being used

Q: Koko eating bananas (leetcode 875)

Koko loves to eat bananas. There are n piles of bananas, the i th pile has $piles[i]$ bananas. The guards have gone and will come back in h hours.

Koko can decide her bananas-per-hour eating speed of k . Each hour, she chooses some pile of bananas and eats k bananas from that pile. If the pile has less than k bananas, she eats all of them instead and will not eat any more bananas during this hour.

Koko likes to eat slowly but still wants to finish eating all the bananas before the guards return.

Return the minimum integer k such that she can eat all the bananas within h hours.

Example 1:

Input: $piles = [3,6,7,11]$, $h = 8$

Output: 4

Example 2:

Input: $piles = [30,11,23,4,20]$, $h = 5$

Output: 30

Code:

```
class Solution {
    public int minEatingSpeed(int[] piles, int h) {
        int left = 1;
        int right = Arrays.stream(piles).max().getAsInt();

        while (left < right) {
            int mid = (left + right) / 2;
            if (canEatAll(piles, mid, h)) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
    }
}
```

```

        return left;
    }

    private boolean canEatAll(int[] piles, int speed, int h)
    {
        int time = 0;
        for (int pile : piles) {
            time += (pile - 1) / speed + 1; // calculate the
time required to eat this pile
            if (time > h) {
                return false; // if the total time is
greater than h, return false
            }
        }
        return true; // if all piles can be eaten within h
hours, return true
    }
}

```

Example 1:

Intuition

We need to find the minimum integer k such that Koko can eat all the bananas within h hours. This means that we need to find the smallest value of k such that she can eat all the bananas within h hours.

Approach

Initialize left and right pointers as $\text{left} = 1$ and $\text{right} = \text{maximum number of bananas in any pile}$.

While the left pointer is less than the right pointer, repeat the following:

- Calculate the middle pointer using $\text{mid} = (\text{left} + \text{right}) / 2$.
- Check if Koko can eat all the bananas at the current speed (middle pointer) within h hours using the `canEatAll` method.
- If Koko can eat all the bananas at the current speed, update the right pointer to the middle pointer using $\text{right} = \text{mid}$.
- If Koko cannot eat all the bananas at the current speed, update the left pointer to $\text{mid} + 1$.

Once the left pointer is equal to the right pointer, return the left pointer as the minimum speed at which Koko can eat all the bananas within h hours.

The `canEatAll` method calculates the total time required to eat all the piles using the given speed. If the total time is greater than h , the method returns false, otherwise, it returns true.

Time Complexity: $O(n * \log n)$. The binary search algorithm has a time complexity of $O(\log n)$, where n is the maximum number of bananas in a pile. The `canEatAll` function has a time complexity of $O(n)$, where n is the number of piles. We perform $\log n$ analyses, and for each we process n packages.

Auxiliary Space: $O(1)$, no extra space being used

Q: Minimum time to complete trips (leetcode 2187)

You are given an array `time` where `time[i]` denotes the time taken by the i th bus to complete one trip.

Each bus can make multiple trips successively; that is, the next trip can start immediately after completing the current trip. Also, each bus operates independently; that is, the trips of one bus do not influence the trips of any other bus.

You are also given an integer `totalTrips`, which denotes the number of trips all buses should make in total. Return the minimum time required for all buses to complete at least `totalTrips` trips.

Input: `time = [1,2,3], totalTrips = 5`

Output: 3

Explanation:

- At time $t = 1$, the number of trips completed by each bus are $[1,0,0]$.

The total number of trips completed is $1 + 0 + 0 = 1$.

- At time $t = 2$, the number of trips completed by each bus are $[2,1,0]$.

The total number of trips completed is $2 + 1 + 0 = 3$.

- At time $t = 3$, the number of trips completed by each bus are $[3,1,1]$.

The total number of trips completed is $3 + 1 + 1 = 5$.

So the minimum time needed for all buses to complete at least 5 trips is 3.

Code:

```
public long minimumTime(int[] time, int totalTrips) {
    long lo = 1, hi = 1000000000000000L;
    while (lo < hi) {
        long mid = lo + (hi - lo) / 2;
        if (!f(mid, totalTrips, time)) lo = mid + 1;
        else hi = mid;
    }
    return lo;
}
private boolean f(long x, int totalTrips, int[] time) {
    long sum = 0;
    for(int t: time) sum += x / t;
    return sum ≥ totalTrips;
}
```

Explanation:

For any time x, we can have total trips = $\sum(x / \text{time}[i])$ where i in [0, time.size()])

We need to minimize the above mentioned function total trips such that it is greater than or equal to the given variable totalTrips.

We can use binary search.

During the contest I got away with keeping lo = 1 and hi = 10 ^ 15

On further inspection of the problem we can deduce that max value of x can be min(times) * totalTrips . So that can be used as hi

Time Complexity: $O(n \log(\min(\text{time}) * \text{totalTrips}))$ //n for the boolean f funciton // $\log(\min(\text{time}) * \text{totaltrips})$ for binary search since $\max_x = \min(\text{time}) * \text{totaltrips}$

Auxiliary Space: $O(1)$, no extra space being used



**THANK
YOU!**