

## Topic Introduction

Hello, prepper! Today, we're diving into a powerful and sneaky algorithm pattern: **Binary Search on the Answer** (sometimes called "Parametric Search"). You might already know classic binary search for finding a value in a sorted array. But what if you could use binary search to find the *best* answer in a range of possibilities, even when the array isn't sorted by the thing you're searching for?

### Definition:

Binary Search on the Answer is a technique where, instead of searching a sorted array, you search within a range of possible answers, repeatedly narrowing the range based on whether a "candidate answer" can solve the problem.

### How does it work?

- You define a range of possible answers (say, [min, max]).
- For each midpoint in this range, you check — using a helper function — "Is it possible to solve the problem with this value?"
- Depending on the answer, you adjust your search range up or down, just like classic binary search.

### When is it useful?

Interviewers love these problems, especially when:

- You need to minimize or maximize something (speed, distance, time, etc.).
- The answer falls within a numeric range.
- For a given candidate answer, you can efficiently check feasibility (often with greedy or simulation).

### Quick Example (not from our problem list):

Suppose you want to fill **k** containers with **n** items, minimizing the largest load. If you can quickly check "Can I do this with a max load of X?", you can binary search X.

Let's see this concept in action with our handpicked trio:

- [Koko Eating Bananas](#)
- [Minimum Time to Complete Trips](#)
- [Magnetic Force Between Two Balls](#)

### Why are these grouped together?

Each one asks for the minimum or maximum value satisfying a certain condition, and brute-force is way too slow. They all become efficient when you use binary search not on the array, but on the *possible answer*!

## Problem 1: Koko Eating Bananas

[LeetCode Link](#)

### Problem Statement (in my words):

Koko loves bananas. She has **n** piles of bananas, and each pile has some bananas. She can eat at a fixed speed of **k** bananas per hour. Each hour, she can choose any pile and eat up to **k** bananas from it (if the pile has less, she eats what's left). Koko wants to eat all the bananas in **h** hours or less. What is the *minimum integer speed* **k** so that she can finish all the bananas on time?

### Example Input/Output:

## PrepLetter: Koko Eating Bananas and similar

---

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

Output: **4**

Explanation:

- If she eats 4/hour, it takes 1 hour for 3, 2 hours for 6, 2 for 7, and 3 for 11, totaling 8 hours.

### Let's walk through it:

Suppose Koko tries eating 3 bananas/hour:

- [3]: 1 hour
- [6]: 2 hours
- [7]: 3 hours
- [11]: 4 hours

Total:  $1+2+3+4 = 10$  hours  $> 8$ . So too slow.

Try 4 bananas/hour:

- [3]: 1 hour
- [6]: 2 hours
- [7]: 2 hours
- [11]: 3 hours

Total: 8 hours. Fits!

### Try this test case yourself:

`piles = [30, 11, 23, 4, 20], h = 5`

### Brute-force approach:

Try every speed `k` from 1 up to the largest pile, and for each, simulate eating.

Time Complexity:  $O(\max(\text{piles}) * n)$

Too slow if piles are large.

### Optimal approach: Binary Search on Answer

- The *slowest* possible speed is 1 banana/hour.
- The *fastest* needed is the size of the largest pile (because she can't eat more than that from any pile in an hour).
- For any guess `k`, we can compute how many hours she needs (by summing up  $\text{ceil}(\text{pile}/k)$  for each pile).
- If she can finish in  $\leq h$  hours, maybe we can do with even less speed.
- Otherwise, we need to eat faster.

### Step-by-Step:

- Set `left = 1, right = max(piles)`.
- While `left < right`:
  - Try `mid = (left + right) // 2` bananas/hour.
  - Compute total hours needed at this speed.
  - If within `h`, try slower (`right = mid`).
  - Else, try faster (`left = mid + 1`).
- At the end, `left` is the minimal speed.

Take a moment to solve this on your own before jumping into the code!

### Python Solution:

```
import math

def minEatingSpeed(piles, h):
    left, right = 1, max(piles)
    while left < right:
        mid = (left + right) // 2
        total_hours = 0
        for pile in piles:
            # math.ceil(pile / mid) gives hours needed for this pile
            total_hours += math.ceil(pile / mid)
        if total_hours <= h:
            right = mid # Try slower
        else:
            left = mid + 1 # Need more speed
    return left
```

**Time Complexity:** O( $n \log M$ ), where  $n$  = number of piles,  $M$  = max pile size.

**Space Complexity:** O(1)

### Code Explanation:

- We use binary search over possible speeds.
- For each candidate speed, we calculate the total hours needed using `ceil(pile / speed)`.
- Adjust the search range based on whether Koko can finish in time.

### Trace Example:

piles = [3, 6, 7, 11], h = 8

- left=1, right=11
- mid=6: hours=1+1+2+2=6 (within h), so right=6
- mid=3: hours=1+2+3+4=10 (too slow), left=4
- mid=5: hours=1+2+2+3=8, right=5
- mid=4: hours=1+2+2+3=8, right=4

Now left==right==4; that's the answer!

### Try this test case yourself:

piles = [312884470], h = 968709

### Reflect:

Did you know this could also be solved with a priority queue by simulating eating the largest pile first? But binary search is far more efficient here!

## Problem 2: Minimum Time to Complete Trips

[LeetCode Link](#)

### Problem Statement (in my words):

## PrepLetter: Koko Eating Bananas and similar

You have `n` buses, each with a trip time (`time[i]`). Each bus can make as many round-trips as possible. Given a goal `totalTrips`, find the *minimum time* needed for all buses combined to complete at least `totalTrips` trips.

### Example Input/Output:

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

Output: `3`

Explanation:

- In 3 units of time: bus 1 does 3 trips, bus 2 does 1, bus 3 does 1. Total =  $3+1+1=5$ .

### Similarities to previous problem:

- We're searching for a *minimum* value (time).
- For a candidate time, we can check if enough trips can be made.

### Brute-force approach:

Try every possible time from 1 upwards, counting total trips for each.

Way too slow.

### Optimal approach: Binary Search on Answer

- The earliest possible time is 1.
- The latest we need to try is `min(time) * totalTrips` (if the fastest bus did all trips).
- For a candidate time, sum up `candidate_time // time[i]` for all buses (trips each bus can complete).
- If total  $\geq$  totalTrips, try less time. Else, try more.

### Step-by-Step Pseudocode:

```
left = 1
right = min(time) * totalTrips
while left < right:
    mid = (left + right) // 2
    trips = sum(mid // t for t in time)
    if trips >= totalTrips:
        right = mid
    else:
        left = mid + 1
return left
```

### Example Walkthrough:

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

- `left=1, right=5`
- `mid=3: 3//1=3, 3//2=1, 3//3=1, total=5`
- `trips >= totalTrips, so right=3`
  - `mid=2: 2//1=2, 2//2=1, 2//3=0, total=3`
  - `trips < totalTrips, left=3`

Now `left==right==3`, so answer is 3.

### Try this test case yourself:

`time = [2], totalTrips = 1`

**Time Complexity:**  $O(n \log(\min(\text{time}) * \text{totalTrips}))$

**Space Complexity:**  $O(1)$

Give this a try before peeking at the code!

## Problem 3: Magnetic Force Between Two Balls

[LeetCode Link](#)

### Problem Statement (in my words):

You have  $n$  positions (sorted or unsorted) along a line. You want to place  $m$  balls in these positions, so that the *minimum distance* between any two balls is as large as possible. Find this maximum possible minimum distance.

### What's different here?

- This time, we're maximizing the *minimum* of something (distance).
- For a candidate distance, we check if it's possible to place all balls with at least that much separation.
- The feasibility check uses a greedy algorithm.

### Brute-force approach:

Try every possible minimal distance, for each, check all placement possibilities.

Way too slow.

### Optimal approach: Binary Search on Answer

- Positions must be sorted.
- The *smallest* possible min distance is 1.
- The *largest* possible is  $\max(\text{positions}) - \min(\text{positions})$ .
- For candidate distance, greedily place balls as far apart as possible.
- If you can place all balls, try a bigger distance. Else, try smaller.

### Step-by-Step Pseudocode:

```
sort positions
left = 1
right = (max(positions) - min(positions))
while left < right:
    mid = (left + right + 1) // 2 # upper mid to avoid infinite loop
    count = 1
    last_pos = positions[0]
    for pos in positions[1:]:
        if pos - last_pos >= mid:
            count += 1
            last_pos = pos
        if count >= m:
            left = mid # Possible to place, try for more
        else:
            right = mid - 1 # Not enough, try less
```

```
return left
```

### Example Input/Output:

positions = [1, 2, 3, 4, 7], m = 3

Output: 3

Explanation: Place balls at 1, 4, 7. Min distance = 3.

### Try this test case yourself:

positions = [5, 4, 3, 2, 1, 1000000000], m = 2

**Time Complexity:**  $O(n \log(\text{max\_distance}))$

**Space Complexity:**  $O(1)$

Can you spot the similarities and subtle differences with the previous problems? Could you imagine a different approach here, like using backtracking (though less optimal)? Take a moment to reflect and try your own implementation!

## Summary and Next Steps

Today, you tackled three challenging yet classic "binary search on the answer" problems:

- **Koko Eating Bananas:** Minimize Koko's eating speed.
- **Minimum Time to Complete Trips:** Minimize total time so all trips can finish.
- **Magnetic Force Between Two Balls:** Maximize the minimum distance between balls.

### Key Patterns:

- When the solution is a number (min/max something) and can be checked efficiently, use binary search on that number.
- For each "guess", simulate or greedily check if it works.
- The search space is often based on problem constraints, not just the input array.

### Common Mistakes:

- Off-by-one errors: Be careful with binary search boundaries and mid calculation, especially when maximizing.
- Forgetting to sort when needed (like in the last problem).
- Not checking if the feasibility function is monotonic (should always be for these problems).

### Action List:

- Solve all 3 problems on your own — even the one with code provided.
- Try solving Problem 2 and 3 using a different technique, like greedy or even brute force for small cases.
- Explore other problems that use this same "binary search on the answer" pattern (e.g., Split Array Largest Sum, Capacity to Ship Packages within D Days).
- Compare your solution with others — especially for edge-case handling and style.
- If you get stuck, that's okay! The key is to keep practicing and reflecting on your thought process.

Keep at it! Every time you spot this pattern, you'll get a little faster and a lot more confident. Happy coding!