

■ Koko Eating Bananas

Binary search on the answer — not the array!

● Binary Search

■ MEDIUM

■ What You Will Learn

- The 'binary search on answer' technique — a game changer
- How to identify monotonic functions and apply binary search on them
- Building a feasibility check function
- Why min/max optimization problems can be binary searched

■ Intuition First

Koko has banana piles. She eats at speed k (bananas/hour). She needs to finish all piles within h hours. Question: what is the minimum k ? Key insight: if Koko CAN finish at speed k , she can definitely finish at speed $k+1$. This is monotonic! Binary search on k between 1 and $\max(\text{piles})$.

■ Binary Search on Answer

Instead of searching an array, we search the answer space $[1, \max_pile]$. For each candidate speed k , we CHECK if it's feasible. The check is $O(n)$. Since the feasibility flips from False to True at some threshold, we binary search for that threshold.

■ Visual — The Monotonic Property

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

```
Speed 1: hours = 3+6+7+11 = 27 > 8 → FAIL
Speed 2: hours = 2+3+4+6 = 15 > 8 → FAIL
Speed 3: hours = 1+2+3+4 = 10 > 8 → FAIL
Speed 4: hours = 1+2+2+3 = 8 = 8 → PASS ✓
Speed 5: hours = 1+2+2+3 = 8 = 8 → PASS
...all higher speeds also PASS
```

```
Monotonic: FAIL...FAIL...FAIL...PASS...PASS...PASS
```

^

We want the FIRST PASS = 4

```
Binary search the speed range [1, 11]
```

■ Feasibility Check

For a given speed k , hours needed = sum of $\text{ceil}(\text{pile} / k)$ for each pile. In Java, $\text{ceil}(a/b) = (a + b - 1) / b$. If total hours $\leq h$, return true.

■ Java Code

```
1  class Solution {
2  public int minEatingSpeed(int[] piles, int h) {
3      int left = 1;
4      int right = 0;
5      for (int p : piles) right = Math.max(right, p);
6      // Binary search on speed [1, max_pile]
7      while (left < right) {
8          int mid = left + (right - left) / 2;
9          if (canFinish(piles, mid, h)) {
10             right = mid; // mid works, try smaller
11         } else {
12             left = mid + 1; // mid fails, need faster
13         }
14     }
15     return left; // minimum feasible speed
16 }
17
18 private boolean canFinish(int[] piles, int k, int h) {
19     int hours = 0;
20     for (int pile : piles) {
21         hours += (pile + k - 1) / k; // ceiling division
22     }
23     return hours <= h;
24 }
25 }
```

■■ Complexity

Complexity	Value	What it means
Time	$O(n \log m)$	n = piles count, m = max pile size. Binary search is $O(\log m)$, check is $O(n)$.
Space	$O(1)$	Only constant space for variables

■■ Common Mistakes

■ Mistake

Using `right=mid-1` instead of `right=mid` in the `canFinish` branch. Since `mid` itself might be the answer, you must preserve it.

■ Mistake 2

Using integer division for ceiling: `pile/k` truncates. Always use `(pile+k-1)/k` or `Math.ceil((double)pile/k)`.

■ Pro Tips

■ Pattern Recognition

The template 'binary search on answer + feasibility check' solves a huge class of problems: ship packages in D days, capacity to ship packages, minimum time to complete tasks. Master this pattern!

■ Practice Problems — Same Pattern

#	Problem	Difficulty
1	Koko Eating Bananas (LeetCode 875)	Medium
2	Capacity To Ship Packages Within D Days (LeetCode 1011)	Medium
3	Minimum Number of Days to Make m Bouquets (LeetCode 1482)	Medium
4	Split Array Largest Sum (LeetCode 410)	Hard

■ Quick Revision

✓	Binary search on answer space <code>[1, max(piles)]</code>
✓	Build <code>canFinish(k)</code> check: <code>total hours = sum(ceil(pile/k)) <= h</code>
✓	Monotonic: speeds below threshold FAIL, above PASS
✓	When <code>canFinish</code> : <code>right=mid</code> (preserve mid). When fail: <code>left=mid+1</code>
✓	Time $O(n \log m)$, Space $O(1)$

■ Visual Thinking

Imagine a dial that controls eating speed. As you turn it up, the time needed drops. At some point it crosses the threshold `h`. Binary search is like intelligently turning the dial — jumping to the middle, checking, then adjusting left or right. Much faster than turning one notch at a time.