Binary Search for Greedy Problems (Looking for Maximum) — Coding Interview Notes (Light Theme)

General Pattern Template

```
def fn(arr):
    def check(x):
        # this function is implemented depending on the problem
        return BOOLEAN

    left = MINIMUM_POSSIBLE_ANSWER
    right = MAXIMUM_POSSIBLE_ANSWER
    while left <= right:
        mid = (left + right) // 2
        if check(mid):
            left = mid + 1
        else:
            right = mid - 1

    return right
```

Concept:
This binary search pattern finds the **maximum feasible value** satisfying a monotonic condition. Unlike the "minimize feasible" version (which returns left), this variant returns right, representing the **largest value** that still meets the constraints.

The function check(x) determines feasibility — returning True if x can be achieved. The feasible region lies to the **left** (smaller x), and infeasible region to the right.

Time Complexity: O(log(max - min) × check_time)

Key Ideas

1. Search over the range of possible answers — **not indices**.
2. The **check(x)** function must be monotonic: if x works, all smaller x also work.
3. When looking for the maximum, move left = mid + 1 if feasible.
4. Return **right** as the largest feasible value after binary search ends.
5. Common in problems asking for maximum distance, minimum difference, or optimal largest value.

Example 1: Aggressive Cows (Maximize Minimum Distance)

Goal: Place k cows in stalls such that the minimum distance between any two is maximized.
Approach: Binary search the minimum distance; check() tries placing cows greedily.

```
def max_min_distance(stalls, k):
    stalls.sort()

    def check(dist):
        count, last = 1, stalls[0]
        for s in stalls[1:]:
            if s - last >= dist:
                count += 1
                last = s
            if count >= k:
                return True
        return False

    left, right = 0, stalls[-1] - stalls[0]
    while left <= right:
        mid = (left + right) // 2
        if check(mid):
            left = mid + 1
        else:
            right = mid - 1
    return right

# Example
print(max_min_distance([1,2,8,4,9], 3))  # Output: 3
```

Example 2: Split Array to Maximize Minimum Sum

Goal: Split the array into m subarrays to maximize the minimum subarray sum.
Approach: Binary search possible minimum sum values; check greedily how many parts can be formed.

```
def maximize_min_sum(nums, m):
    def check(x):
        parts, curr = 1, 0
        for num in nums:
            if curr + num > x:
                parts += 1
                curr = 0
            curr += num
        return parts <= m

    left, right = min(nums), sum(nums)
    while left <= right:
        mid = (left + right) // 2
        if check(mid):
            left = mid + 1
        else:
            right = mid - 1
    return right

# Example
print(maximize_min_sum([7,2,5,10,8], 2))  # Output: 18
```

Example 3: Maximize Rope Cut Length

Goal: Given ropes of varying lengths, find the maximum possible length to cut them into at least K equal pieces.
Approach: Binary search length; check how many pieces can be formed.

```
def max_rope_length(ropes, k):
    def check(length):
        return sum(r // length for r in ropes) >= k

    left, right = 1, max(ropes)
    while left <= right:
        mid = (left + right) // 2
        if check(mid):
            left = mid + 1
        else:
            right = mid - 1
    return right

# Example
print(max_rope_length([802,743,457,539], 11))  # Output: 200
```

Summary Table

ProblemRangeCheck ConditionGoalReturn Aggressive cows[0, max-min]Can place ≥ k cowsMaximize min distanceright Split array[min(nums), sum(nums)]Can form ≤ m groupsMaximize smallest sumright Rope cutting[1, max(ropes)]Pieces ≥ kMaximize cut lengthright