Subarray Largest Sum

https://www.youtube.com/watch?v=5WZl3MMT0Eg

Here are the key concepts you'd encounter (and learn) while tackling the Maximum Subarray problem:

1. **Brute-Force Enumeration**
   - Enumerating every possible contiguous subarray using three nested loops.
   - Understanding index ranges (start = *i*, end = *j*, and summing from *i* to *j*).
   - Learning how summing from scratch each time costs $O(n^3)$.
   - Realizing why such a naïve approach doesn't scale.
2. **Incremental (Prefix) Summation Trick**
   - Instead of recomputing the entire sum for each new end index, keep a "running sum" as you extend the subarray.
   - For a fixed start *i*, you compute
   - `curr = 0`
   - `for j in i…n-1:`
   - `    curr += nums[j]`
   - `    // now curr is the sum of nums[i..j]`
   - This removes the innermost loop and drops the time complexity to $O(n^2)$.
   - Teaches you how storing partial results can save repeated work.
3. **Time-Complexity Analysis**
   - Summing up: three loops → $O(n^3)$; two loops with running sum → $O(n^2)$.
   - Learning how to count iterations (e.g. Σ_{i=0…n–1} Σ_{j=i…n–1} 1 = $O(n^2)$), and how removing one loop changes the big-O.
4. **Handling Negative Prefixes (Greedy/Pruning Idea)**
   - Observing that if your running sum ever goes below zero, adding it to a future positive stretch can only hurt.
   - Learning the "if current_sum < 0, reset to 0" heuristic.
   - Understanding why "throwing away" a negative prefix is safe when looking only for the maximum contiguous sum.
5. **Kadane's Algorithm (Linear DP/GREEDY)**
   - Recognizing that at each index *i*, the best subarray ending exactly at *i* is either:
     - Just `nums[i]` (if the previous running sum was negative), or
     - Previous running sum + `nums[i]` (if that prefix was ≥ 0).
   - Translating that into a one-pass loop that keeps two variables:
     - `currSum = max(nums[i], currSum + nums[i])` (or the equivalent "reset if negative, then add"),
     - `maxSoFar = max(maxSoFar, currSum).`
   - Internalizing the "maximum-ending-here" vs. "maximum-so-far" dynamic-programming idea.
6. **Edge-Case Handling**
   - Why you initialize `maxSoFar = nums[0]` instead of 0 (so you handle all-negative arrays correctly).
   - Ensuring you never accidentally return 0 when every element is negative.
   - Confirming that the array is nonempty before you begin.
7. **Divide-and-Conquer Approach (O(n log n))**
   - Splitting the array into two halves, recursively finding:

        - The maximum subarray entirely in the left half.
        - The maximum subarray entirely in the right half.
        - The maximum subarray that crosses the midpoint (find best suffix on left + best prefix on right).
    - Learning how to combine those three cases to get the overall maximum.
    - Analyzing why this yields $T(n) = 2\,T(n/2) + O(n) \Rightarrow O(n \log n)$.
    - Gaining comfort with "divide, conquer, and merge" recursion patterns.
8. **Space-Complexity Trade-Offs**
    - Brute-force and $O(n^2)$ methods use $O(1)$ extra space (just a few counters).
    - A divide-and-conquer recursion uses $O(\log n)$ stack space.
    - Kadane's uses strictly $O(1)$ extra space, which is optimal.
9. **Understanding "Local" vs. "Global" Maximum**
    - Local maximum: best sum ending at the current index.
    - Global maximum: best sum seen anywhere so far.
    - Seeing how local decisions (resetting when negative) affect the global answer.
10. **Practical Implementation Details**
    - Iterating through arrays in Python (or any language).
    - Using `max()` to update the global best.
    - Handling off-by-one errors when summing subarrays.
    - Translating pseudocode into clean, production-ready code.

By working through these steps—starting from brute-force, then prefix-sum optimization, then Kadane's linear solution, and optionally the divide-and-conquer—you'll cover a spectrum of algorithmic thinking:

- **Brute-force enumeration** (time complexity reasoning),
- **Incremental summation** (prefix-sum optimization),
- **Greedy/DP pruning** (Kadane's algorithm),
- **Divide & Conquer recursion** (splitting and merging subproblems),
- **Edge-case robustness** (handling all-negative arrays),
- **Space/time trade-offs** (understanding $O(1)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$ behaviors).

All of these concepts show up in many other array/DP problems too, so you build a toolkit you can reuse on other coding-interview questions.