Arrays Cheatsheet

Topic Overview

Arrays in Java are contiguous memory blocks that store elements of the same type, providing efficient random access and serving as the foundation for many algorithmic solutions. This cheatsheet explores various techniques with step-by-step approaches.

Prerequisites

None

List of Subtopics

- 1D Arrays
- 2D Arrays
- Dynamic Arrays
- Prefix Sum (Approach Only)
- Sliding Window
- Two Pointer Technique
- Kadane's Algorithm
- Array Rotation
- Selection Sort
- Merge Intervals

Key Concepts Explained

- **1D Arrays**: A linear structure where elements are accessed via indices (0 to n-1), ideal for ordered data manipulation.
- **Sliding Window**: A technique to maintain a flexible subarray, optimizing problems by adjusting boundaries.
- **Kadane's Algorithm**: A dynamic approach to identify the maximum subarray by evaluating and updating local maxima.

Approaches to Solve Problems with Step-by-Step Algorithms

- **1D Arrays**:
  - **Algorithm**:
    1. Start with a loop variable set to index 0.
    2. Continue the loop until the index reaches the array length minus one.
    3. For each index, perform the desired operation (e.g., print or modify the element).
    4. If the operation involves aggregation, maintain a running total and update it at each step.
    5. Before processing, check for edge cases such as null or empty arrays.
  - **Context**: This method is effective for traversing or transforming data, achieving O(n) time complexity with a single pass.
- **2D Arrays**:
  - **Algorithm**:
    1. Identify the number of rows and columns based on the array's dimensions.
    2. Implement a nested loop: the outer loop iterates over rows, and the inner loop over columns.
    3. Access each element using [row][column] indices.
    4. Apply the required operation (e.g., search or transpose) considering row or column context.
    5. Include boundary checks to prevent index out-of-bounds errors.
  - **Context**: Suitable for matrix operations like searching or rotation, with O(rows * columns) complexity.
- **Dynamic Arrays**:
  - **Algorithm**:
    1. Begin with an ArrayList initialized with a default or specified capacity.
    2. When adding an element exceeds the current capacity, create a new ArrayList with double the size.
    3. Copy all existing elements from the old ArrayList to the new one.
    4. Add the new element and update the reference to the new ArrayList.
    5. Repeat the resizing process as additional elements are inserted.
  - **Context**: Addresses the limitations of fixed-size arrays, ideal for dynamic data growth.
- **Prefix Sum**:
  - **Algorithm**:
    1. Create an array to hold prefix sums, with a size of n+1.
    2. Set the first element to 0 to simplify range query calculations.

    3. Iterate from index 1 to n, adding the current element to the previous prefix sum.
    4. Store each cumulative sum at its corresponding index.
    5. For a range query from left to right, compute the difference prefix[right+1] - prefix[left].
  - **Context**: Enables O(1) range queries after O(n) preprocessing, useful for repeated sum queries.

- **Sliding Window**:
  - **Algorithm**:
    1. Define the window size or a condition (e.g., fixed length k or target sum).
    2. Set two pointers, left and right, both starting at the beginning.
    3. Move the right pointer to expand the window until the condition is satisfied.
    4. If the condition is exceeded, move the left pointer to shrink the window.
    5. Record the optimal solution (e.g., minimum length) for each valid window.
    6. Continue until the right pointer reaches the array's end.
  - **Context**: Optimizes subarray problems like maximum sum, reducing time to O(n).

- **Two Pointer Technique**:
  - **Algorithm**:
    1. Initialize two pointers, typically at the start and end or adjacent positions.
    2. Compare elements at both pointers based on the problem's condition (e.g., sum equals target).
    3. Adjust pointers (e.g., increment if too small, decrement if too large) based on the comparison.
    4. Continue until the pointers meet or cross, tracking valid pairs or merges.
    5. Ensure the input is sorted if required for the technique to work correctly.
  - **Context**: Highly efficient for sorted arrays, often achieving O(n) or O(n log n) with sorting.

- **Kadane's Algorithm**:
  - **Algorithm**:
    1. Initialize two variables: maxEndingHere and maxSoFar, both set to the first element.
    2. Begin iteration through the array starting from the second element.
    3. For each element, set maxEndingHere to the maximum of the current element or (maxEndingHere + current).
    4. Update maxSoFar if maxEndingHere is greater than the current maxSoFar.
    5. Continue until the end of the array, then return maxSoFar.

  - **Context**: Solves maximum subarray problems in O(n) time, effectively handling negative values.

- **Array Rotation**:
  - **Algorithm**:
    1. Determine the number of positions to rotate (k), reducing it modulo the array length.
    2. Use the reversal method:
       (a) Reverse the first k elements.
       (b) Reverse the remaining n-k elements.
       (c) Reverse the entire array.
    3. Alternatively, shift elements one by one, using a temp variable for the first element.
    4. Update indices circularly during shifting.
  - **Context**: Useful for cyclic data shifts, with O(n) time for the reversal method.

- **Selection Sort**:
  - **Algorithm**:
    1. Iterate through the array from the first unsorted index.
    2. Find the minimum element in the unsorted portion by comparing with subsequent elements.
    3. Swap the minimum element with the current index position.
    4. Move to the next index and repeat until the entire array is sorted.
  - **Context**: Simple sorting method with O(nš) time, suitable for small datasets.

- **Merge Intervals**:
  - **Algorithm**:
    1. Sort the intervals based on start times.
    2. Initialize a result list with the first interval.
    3. Iterate through remaining intervals, comparing the current start with the last result end.
    4. If overlapping (start <= last end), update the last end to the maximum of both.
    5. If non-overlapping, add the current interval to the result.
    6. Continue until all intervals are processed.
  - **Context**: Optimizes interval consolidation, with O(n log n) due to sorting.

Common LeetCode Problems with Approaches

- **Best Time to Buy and Sell Stock (121)**: Use two pointers to track the minimum price seen so far as the buy point and calculate the maximum profit if selling at each subsequent price. Iterate once, updating both pointers.

- **Maximum Subarray (53)**: Apply Kadane's algorithm, maintaining a running maximum ending at each position and updating the global maximum with each step.
- **Two Sum (1)**: Iterate through the array, for each element, compute the complement needed for the target, and check a separate structure for its presence, storing elements as you go.
- **Minimum Size Subarray Sum (209)**: Employ a sliding window, expanding the right pointer until the subarray sum meets or exceeds the target, then shrinking from the left while possible, tracking the minimum length.
- **Merge Intervals (56)**: Sort intervals by start time, then traverse, merging overlapping intervals by updating the end time of the last interval in the result when applicable.
- **Product of Array Except Self (238)**: Perform two passes: one to compute the product of all elements to the left of each index, and another for the right, then combine these products at each position.

Time & Space Complexities

- Access: O(1)
- Search: O(n)
- Insert/Delete: O(n) due to shifting
- Sliding Window: O(n)
- Space: O(n) for auxiliary arrays

Important Tips & Tricks

- Always validate array bounds to prevent exceptions.
- Leverage ArrayList for dynamic array needs in Java.
- Use two-pointer techniques on sorted data for efficiency.
- Preprocess with prefix sums for repeated range queries.
- Handle edge cases like single-element or empty arrays early.