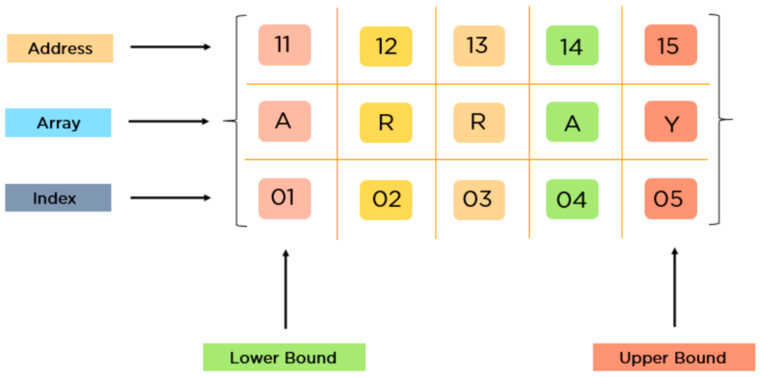


Array & String Problem Solving Techniques

Uber

Agenda

Two Pointer Techniques

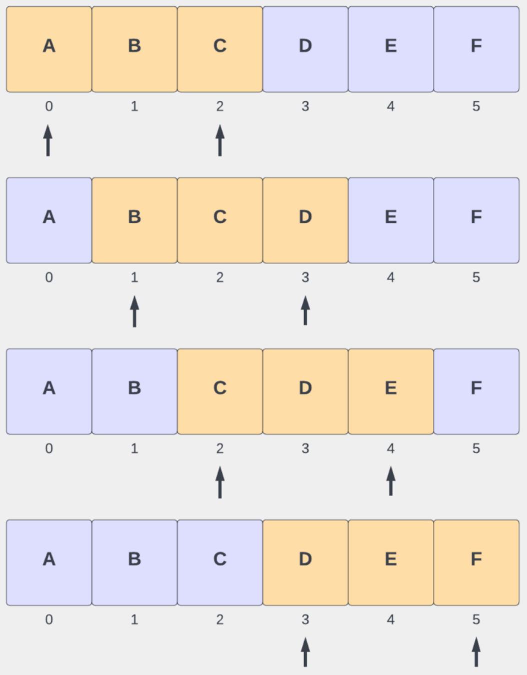
Move through the array/string keeping track of two indices ("pointers") which move until different conditions are met or at different fixed speeds. The pointers alternate moving; when both have moved, the current elements and/or indices are processed. This cycle repeats until the array/string is exhausted.

This technique is effective for eliminating a nested loop and thereby reducing the time complexity by a factor of n (most commonly from O(n^2) to O(n)).

Sliding Window

Fixed Size k

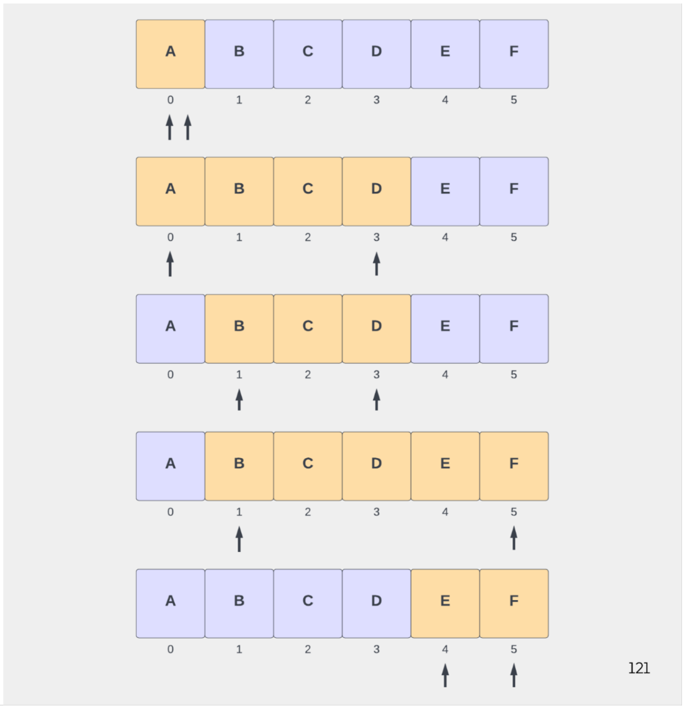
In this variation, the distance between the two pointers always stays the same. The leading pointer typically starts at index k-1 and the trailing pointer at index 0. Each pointer advances one index at a time. When a problem asks you to find a value (e.g., min or max) over a subarray/substring of size k, that is your clue to use fixed size sliding window!



Growing / Shrinking Sliding Window

Variable Size

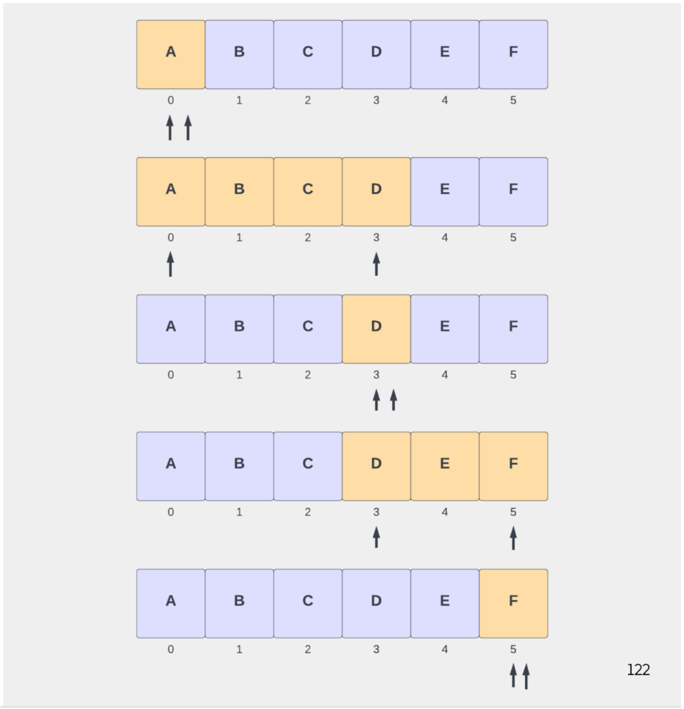
In this variation, the distance between the two pointers changes. The leading pointer advances, growing the window, until a condition is met. Then the trailing pointer advances, shrinking the window., and the process repeats. When a problem asks for the longest/shortest subarray/substring satisfying a condition use variable size sliding window!.



Two-Pointer with "Catch-Up" Condition

Reset to 0 Size

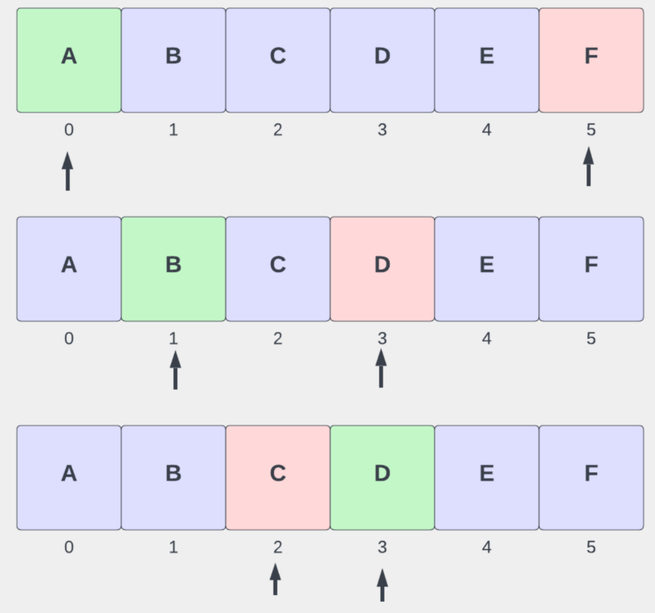
In this variation, the leading pointer advances until some condition is met. The computation (often difference) occurs *before* the trailing pointer moves; it then "jumps ahead" to the position of the leading pointer. Use this technique if the array/string can be chunked (e.g., sequences of increasing/decreasing values, split on "break" values, etc.)



Forward-Backward Two-Pointer

Meet in the “Middle”

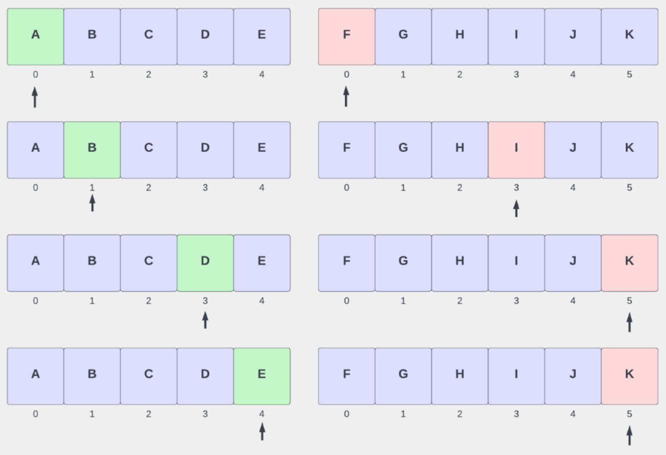
In this variation, one pointer starts at the beginning of the array/string and moves forward; the other starts at the end and moves backwards. The process terminates when the two pointers cross. Use this technique for problems requiring finding corresponding pairs at opposite ends of the array/string (for swapping, summing, etc.).



Simultaneous Iteration Two-Pointer

Two (or more) arrays/strings

In this variation, the two pointers traverse different arrays/strings in parallel. Depending on the problem, the process terminates when one *or both* arrays/string sare exhausted. Use this technique for problems require traversing two arrays/strings “simultaneously.”.



Sorting & Searching Techniques

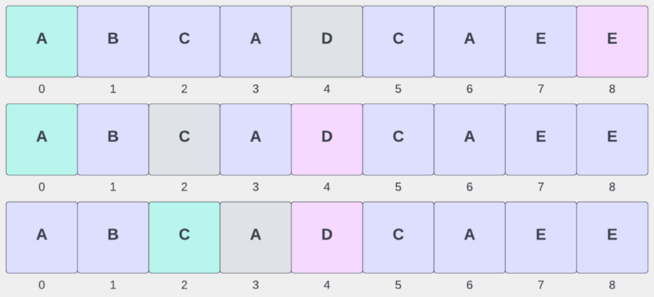
Interview questions will rarely directly ask you to implement a standard sorting or searching algorithm, but they may indirectly imply that you should implement a specific algorithm, usually with some modifications.

Other times, you will can use an out-of-box to pre-sort the input and thereby allow you to use another technique to solve it. The challenge in these problems lies in identifying that the input needs to be sorted and sometimes in determining what aspect of the input to sort by.

Binary Search Variation

Sorted or Almost Sorted

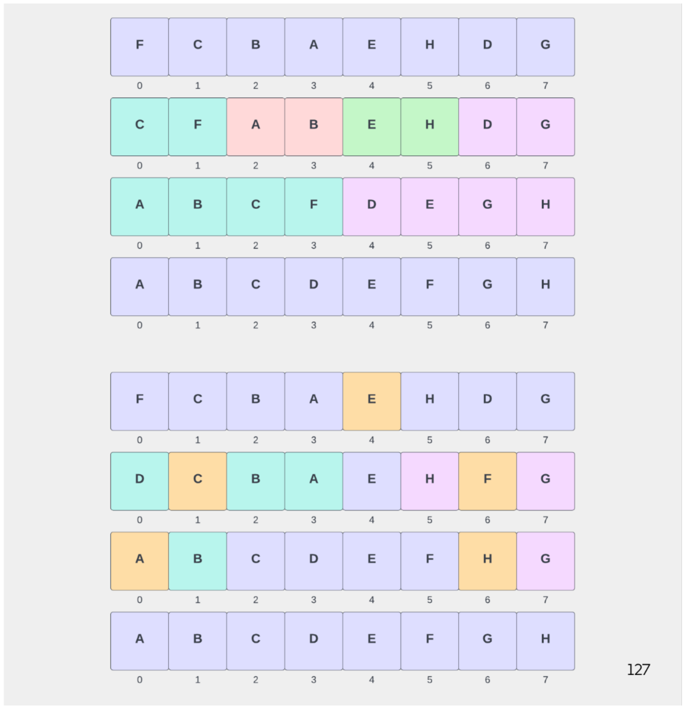
When a problem asks you to find a value or index in a sorted or almost sorted array/string, use binary search. "Almost sorted" means it can be sorted by performing a constant number of shifts or swaps. You can still use binary search by modifying the comparison condition and/or boundary updates.



Sorting Algorithm Variation

Introspect a sort

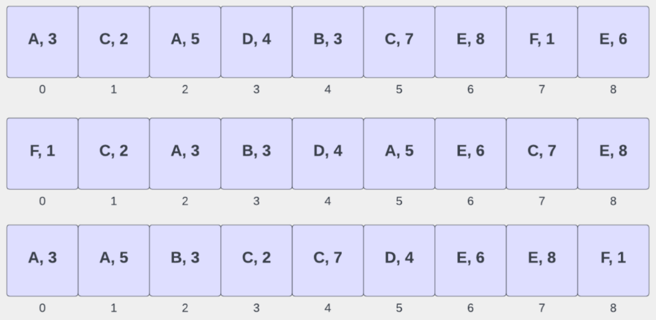
Some problems require you to sort the input with modifications to or data collected from the internal workings of the algorithm (e.g., number of swaps or positions moved). These are rare cases in which you'll implement a sorting algorithm yourself. Often the problem statement will describe the steps in a specific algorithm without naming it.



Sort, then Solve

If only it were sorted...

Sort the input (with a built-in method), then solve it using another technique (often binary search). The caveat is that sorting is O(nlogn) so confirm you can't use an O(n) approach first. This is an example of a more general technique called **transform and conquer** in which the input format is changed to enable use of a problem-solving technique.


A, 3 | C, 2 | A, 5 | D, 4 | B, 3 | C, 7 | E, 8 | F, 1 | E, 6
0 1 2 3 4 5 6 7 8
F, 1 | C, 2 | A, 3 | B, 3 | D, 4 | A, 5 | E, 6 | C, 7 | E, 8
0 1 2 3 4 5 6 7 8
A, 3 | A, 5 | B, 3 | C, 2 | C, 7 | D, 4 | E, 6 | E, 8 | F, 1
0 1 2 3 4 5 6 7 8

Hash Something Techniques

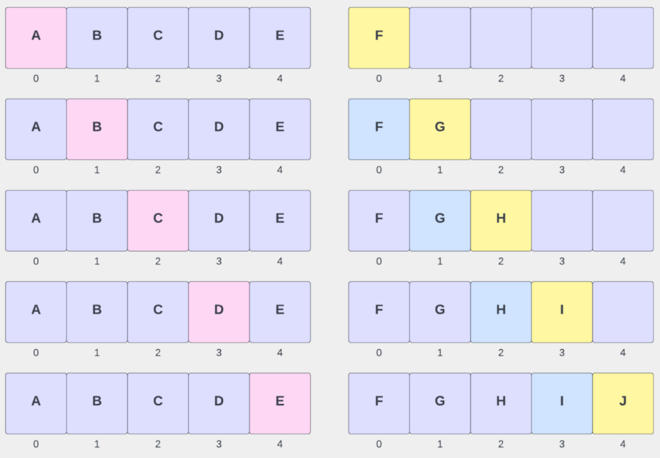
In problems that require repeatedly looking up or re-calculating the same value, using a hash data structure (set or map) can eliminate a nested loop and thereby reducing the time complexity by a factor of n (most commonly from O(n^2) to O(n)).

Hashing techniques are the most common examples of space/time tradeoffs - increasing space complexity to decrease time complexity. Usually, using a set or map will take O(n) space compared to constant space in the brute force solution.

Hash the Running Computation

One-Directional

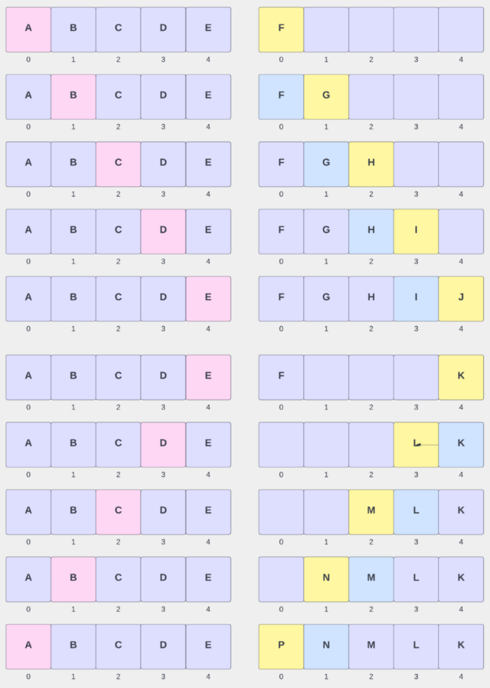
In this technique, a hashmap or an array of the same size as the input is used to store partial computations. The keys are indices; the values are computed while traversing from previous result (for index i-0) and the element at index i. This avoids re-traversing the array to compute the value for each index. This is an example of **dynamic programming**.



Forward-Backward Running Computation

Bi-Directional

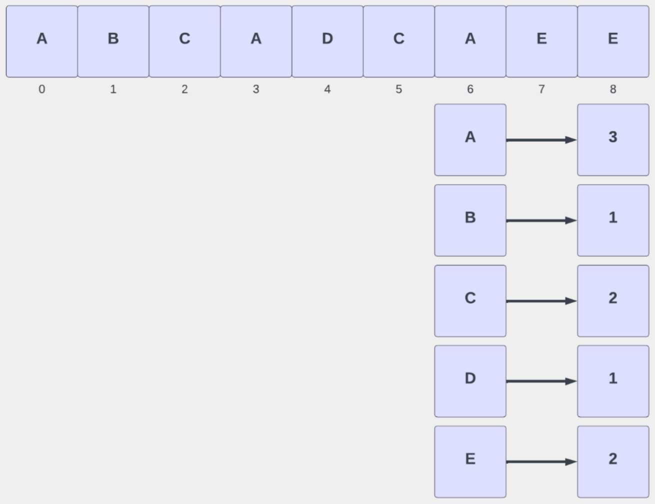
In this variation of running computation, you will traverse the array twice, once forwards and once backwards, and build two maps/arrays. You then combine the forwards and backwards partial results for the solution. Use this technique when you need information from both the left and right of the index to calculate the result.



Hash the Elements

Need them Faster?

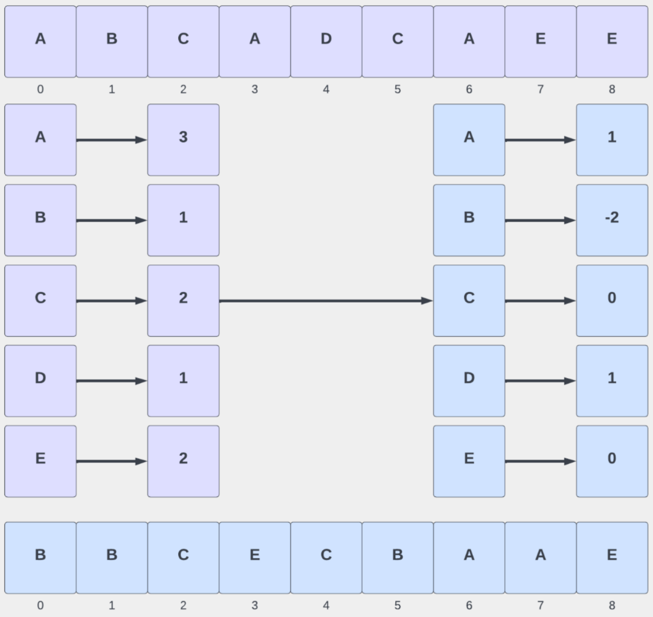
If you need to lookup elements n times, copy the elements from a linear data structure to a hash one, which reduces each lookup time from O(n) to O(1). If you only care about presence or absence of values, create a set of the elements. If you need the actual counts, create a map with the unique elements as keys and the counts as values.



Increment / Decrement Counts

Two Arrays/Strings

If the problem requires comparing elements of two arrays/strings based on presence/absence or counts (and order doesn't matter), you can use a single data structure. Add/increment the elements of one array/string to a set/map, then remove/decrement the elements from the other and evaluate the result.



A B C A D C A E E
0 1 2 3 4 5 6 7 8
A 3
B 1
C 2
D 1
E 2
A 1
B -2
C 0
D 1
E 0
B B C E C B A A E
0 1 2 3 4 5 6 7 8