Algorithm study for interviews:

Two Pointer:

The two-pointer technique is widely used to solve those problems where the sub-optimal solution is greater or equal to O(n^2). The optimal solution of the two-pointer technique is usually O(n \* log(n)) or O(log(n)) if the input is sorted or not. This technique can be divided into two different categories.

The first is called fast-slow pointers, where the fast pointer proceeds at a higher frequency than the slow one.

Example Slow-Fast pointers:

You have a linked list of numbers; find the middle element:

Please set up a slow and a fast pointer where both point to the list's beginning. The fast pointer will proceed twice as fast as the slow pointer. Once the fast reaches the end, the slow pointer will be in the middle of the list.

The second variation of this technique is called front-back pointers, where you set up the pointers at the beginning of the array and the end and move one or the other based on a condition.

Example front-back pointers:

You have an array of values. Find the sum of two values that meet the target value:

First, sort the array in ascending order. Then, set up the front array at the beginning of the array and set up the back pointer at the end of the array. If the sum of the two pointers is greater than the target value, decrease the back pointer by one since the value needs to be smaller; if the sum is more minor, increase the front pointer with the same logic. Return the target if met, or if the two pointers meet, return -1.

Prefix Sum:

The prefix sum is a technique usually used in those problems where there is an array, and you need to calculate a sub array based on the sum of the previous elements or any time there is a sum of values inside an array of elements. This technique works well also for matrices where you need to find the sum of a partition of the matrix based on coordinates.

The main idea is to keep track of the current Sum of the elements and check if the condition is met.

Another approach is to actually modify the elements of the array by adding the previous value to the current to finally obtain a prefix array where every cell of the array contains the sum of the previous elements.

There are some minor changes that you need to make in order to adapt the solution to the problems such as keeping track of the frequency of occurrence using a hash map or replacing the 0s with -1.

In particular in the case of keeping the frequency of occurrence using a hash map it is particularly useful in the problems where you need to find a sub array of sum of elements where they meet a target.

Example of prefix Sum using HashMap: 560. Subarray Sum Equals K

Define hash map like this {0:1} The base case is that if the difference between the target and the current sum is 0 ( for the first time ) we are adding the frequence value 1

Now each time you need to calculate the current Sum if the difference between the current sum and the target is already in the cache you need to add to the output the frequency value and increase it by one ( after adding the value to the output ) otherwise keep track of the current Sum by adding it to the cache or increasing the frequency by one if already existing inside the cache.

The main idea is to check from the current sum how much I have incremented, for example:

out = 0

{0: 1, 4: 1}

If target is 2 and current sum is 4: 4 + 2 = 6 which is the current sum

6 – target (2) = 4 which is already in the cache this means I moved by the target value, and this is a valid subarray

out = 1

{0: 1, 4: 1, 6: 1}

Once this is done keep going and add the frequency up to that point because you are counting all the possible subarrays that you have reached.

Sliding Window:

This technique uses a temporary array where some elements of the initial array are passed. There is a constraint also referred as windows size which defines the length of the temporary array. Depending on the exercises you may consider replacing the temporary array with a counter variable.

This particular technique is useful when you need to calculate sub-arrays and check for similar conditions and can be implemented also for other algorithms seen before and after.

This algorithm reduces the time complexity of the problem from O(N^2) to O(N) but it can also increase the space complexity of the problem if used a temporary array.

Divide and Conquer:

Divide and Conquer is not an algorithm such as the previous ones but is more a general idea behind some of the algorithms we have seen. The general idea behind divide and conquer is to divide the problem is many sub problems until we find a solution to the smallest subproblem generated, once the subproblem has been solved we are returning to the problem above and we solve it. Once all the subproblems are solved we need to merge all the subproblems and then we have the solution to the initial problem. This can be achieved by setting up a base case and finally merging the solution returning the initial problem.

An example of this approach is the merge sort algorithm which heavily uses this idea to sort an array in O(NlogN) time complexity.

Linked List:

Linked list is an important data-structure that is the basis for other more complicated data structures such as trees and graphs. The main idea behind this data structure is to connect one element with the other where each element can access only the next one without the possibility ( unless is a doubly linked list ) to return back. There are two different techniques to use when this problems are encountered.

Two pointer techniques:

Usually save the first and second element and iterate the linked list to create two or more linked lists. This is helpful to make a distinction between different elements and can be helpful when you need to separate elements in the list.

Recursion:

This technique is particular, and it is used in different circumstances, in particular when you need to traverse the list from the end to the beginning. In fact, you keep track of the previous element by returning it recursively ( Using the stack ) and then you can manage the list by interacting with the current and the previous.

This technique can be used to reverse a list.

Monotonic Stack:

The monotonic Stack is a useful technique used when we need to solve those problems that requires two for loops having O(N^2) time complexity. To reduce the time complexity, we set up a stack (which is generally just a list or an array) and we store values in monotonic increasing or decreasing way.

I found useful sometimes to start from the end of the array to initially store the values inside the stack.

The main idea is to check weather a condition is met and pop / append the values in the stack.

Sometimes it is crucial to store the indices rather than the values itself to then subtract the index with the current index to find the value of the sequence.

Priority Queue:

Top 'k' Elements

Quick Select

Overlapping Intervals

Modified Binary Search

Depth-First Search(DFS)

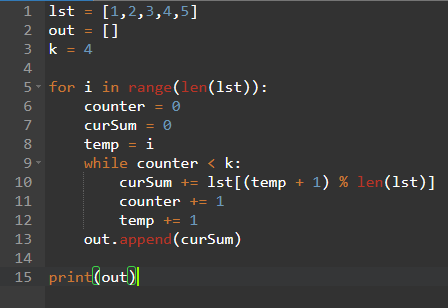
Breadth-First Search(BFS)

Matrix Traversal

Backtracking

Dynamic Programming

Important Notes:

This is a good way to traverse the array in a rotational order, keep in mind that index % length return the current index if < than length, 0 if equal to length and start over if greater. This is how to deal with rotation in arrays.