Algorithms Notes

William Yang

June 3, 2025

Contents

1 Introduction

This document serves as a collection of problem solving techniques to common interview algorithms problems centered around DSA. My studying primarily follows the problems found in Neetcode's 150 list and USACO Guide's readings, both of which I recommend. Additional study resources can be found in the references section.

2 Two Pointers

Our strategy involves iterating two monotonic pointers, usually named `left` and `right`, across the array. Here, monotonic means that both pointers are strictly increasing or decreasing. This propery is essential to ensure the run time of our pointer increments across our input is $O(n)$.

The example problem we will be looking at is *Trapping Rain Water* on Leetcode. The solution breakdown is as follows

- Initialize two pointers `left` and `right` at the beginning and end of the array.

- Initialize two variables `high_left` and `high_right` to keep track of the highest elevation strictly left of the `left` pointer and the highest elevation strictly right of the `right` pointer respectively.

- Iterate while `left` is less than or equal to `right`.

- On each iteration, we increment the pointer on the side with the lower max elevation. So if `high_left` is less than `high_right`, we increment `left`. Otherwise, we increment `right`. The key idea here is that the min elevation between the two sides will also be strictly increasing.

- Now whenever we move one of the pointers, we calculate the amount of trapped water at the index of the pointer we're incrementing from. This is just the minimum of the max elevations of the two sides minus the height of the current index.

```
    def trap(height: List[int]) -> int:
        left = 0
        right = len(height) - 1
        high_left = 0
        high_right = 0
        result = 0
        while left <= right:
            min_height = min(high_left, high_right)
            if high_left > high_right:
                result += max(0, min_height - height[right])
                right -= 1
                high_right = max(height[right + 1], high_right)
            else:
                result += max(0, min_height - height[left])
                left += 1
                high_left = max(height[left - 1], high_left)
        return result
```

Listing 1: Trapping Rain Water

3 Sliding Window

Similar to the two pointer technique, sliding window problems also involve using two pointers to define a window to search for subarrays or substrings with a specific desired property. Typically, key words such as finding minimum or maximum subarray or substring of size k or count the number of subarrays or substrings that satisfy x are signs of a sliding window problem. If you are repeatedly computing a running sum, product etc. for overlapping regions, then sliding window can help avoid doing redundant computations.

Importantly, our left and right pointers have to be monotonically increasing in order to avoid $O(n^2)$ computation time. The first example problem we will be looking at is *Sliding Window Maximum* on Leetcode. The solution breakdown is as follows:

- Our core strategy will be maintaining a monotonic strictly decreasing stack of values from within a given size k window.

- We initialize a double ended queue to represent our stack.

- We vary our right pointer across the array, and at each step whenever we add a value onto the stack, we pop all elements from the back of the stack that are less than the value being added. This is to ensure the stack is strictly increasing and that the leftmost element is the maximum value in the current k wide window.

- Whenever the lowest index in the stack (leftmost element) is more than k indices away from r, then we pop the leftmost element.

- Run time for this algorithm is $O(n)$ since each element in `nums` is only pushed and popped from the stack once. Storage is also $O(n)$ since worst case $k = n$ and we could store all the elements in the array.

```
    def maxSlidingWindow(nums: List[int], k: int) -> List[int]:
        q = deque([])
        results = []
        for r in range(len(nums)):
            if len(q) > 0 and q[0][0] <= r - k:
                q.popleft()
            while len(q) > 0 and q[-1][1] < nums[r]:
                q.pop()
            q.append((r, nums[r]))
            if r >= k - 1:
                results.append(q[0][1])
        return results
```

Listing 2: Sliding Window Maximum

The second example problem is *Minimum Window Substring* on Leetcode. The solution breakdown is as follows:

- We use a hashmap `t_freqs` to keep track of the frequency of each letter in `t`, `all_char_freqs` to keep track of the frequency of each letter in the current window, and `letters_finished` to keep track of the letters in `t` that have been finished in the current window.

- We initialize two pointers `left` and `right` to represent the current window.

- On each iteration, we first check if the length of `letters_finished` equals the length of *t_freqs*, which means we have a valid substring with all the characters in `t`. If so, we check if the current substring is the shortest valid substring we have so far, and if so record the indices of the substring. We then iterate the left pointer forward and iterate again.

- If we don't have a valid substring, we iterate the right pointer forward and iterate again.

```
    def minWindow(s: str, t: str) -> str:
        t_freqs = defaultdict(int)
        # construct freq of letters for t
        for c in t:
            t_freqs[c] += 1

        all_char_freqs = defaultdict(int)
        letters_finished = set() # letter: last index
        left = 0
        right = -1
        shortest_substr_len = float('inf')
        shortest_substr = [0, 0]
```

```python
        while left < len(s):
            # check if we have a valid substring
            if len(letters_finished) == len(t_freqs):
                if right - left + 1 < shortest_substr_len:
                    shortest_substr_len = right - left + 1
                    shortest_substr = [left, right + 1]
                left += 1
                remove_char = s[left - 1]
                all_char_freqs[remove_char] -= 1
                if all_char_freqs[remove_char] < t_freqs[remove_char] and remove_char in letters_finished:
                    letters_finished.remove(remove_char)
            elif right < len(s) - 1:
                right += 1
                c = s[right]
                # add right char
                all_char_freqs[c] += 1
                if all_char_freqs[c] >= t_freqs[c]:
                    letters_finished.add(c)
            else:
                break

        return s[shortest_substr[0]:shortest_substr[1]]
```

Listing 3: Minimum Window Substring

4 Stacks

Stacks are a LIFO data structure where you can only insert and access elements at the top of the stack.

To identify a stack/monotonic stack problems, look for the following:

- You're given a sequence of elements and need to search for each index, what is the nearest element, largest/smallest element with a specific property to the left or right of the current index.
- For instance, a problem such as for each element in an arrya, find the next element to its right that is greater than it.
- Problems where you need to perform matching or balancing of elements such as the balancing parantheses problem.

The problem we will be looking at is *Largest Rectangle in Histogram* on Leetcode. The solution breakdown is as follows:

- The key is, for each rectangle in our rectangles array, we want to find the minimum height rectangle both to its right and left. This will allow us to calculate the area of the rectangle centered at one of the current rectangles in the array.
- We use a monotonic increasing stack to store indices of bars in our heights array.
- We'll iterate through each rectangle in our heights array. For each rectangle, while its height is smaller than the height of the rectangle at the top of our stack, we pop the that rectangle.

  - Note importantly, we're processing the height of the rectangle that we're popping from the stack. The algorithm works since we eventually insert and pop every rectangle from our stack.
  - Since the indices in our stack are monotonically increasing, the current index in our for loop we're on represents the next smallest index to the right of rectangle we're processing from the stack.
  - The new rectangle at the top of the stack then represents the previous smallest index to the left of the rectangle we're processing.

- Once we iterate through all the indices in our for loop, we process the remaining rectangles in our stack assuming that the next smallest index is just at the end of the heights array.

```
def largestRectangleArea(heights) -> int:
    indices_stack = deque([])
    result = 0
    for i, h in enumerate(heights):
        while len(indices_stack) > 0 and h < heights[indices_stack[-1]]:
            bar_ind = indices_stack.pop()
            next_smallest_ind = i
            prev_smallest_ind = -1
            if len(indices_stack) > 0:
                prev_smallest_ind = indices_stack[-1]
            result = max(result, heights[bar_ind] * (next_smallest_ind -
prev_smallest_ind - 1))
        indices_stack.append(i)

    # process remaining indices in stack
    while len(indices_stack) > 0:
        bar_ind = indices_stack.pop()
        next_smallest_ind = len(heights)
        prev_smallest_ind = -1
        if len(indices_stack) > 0:
            prev_smallest_ind = indices_stack[-1]
        result = max(result, heights[bar_ind] * (next_smallest_ind -
prev_smallest_ind - 1))
    return result
```

Listing 4: Next Greater Element I

5 Heaps

6 Binary Search

7 Linked Lists

8 Tries

9 Graphs

10 Dynamic Programming