Linear Datastructure - Techniques

Two Pointer Technique

Concept

- Uses two pointers moving at different speeds or in opposite directions.
- Helps solve **searching, sorting, and subarray problems** efficiently.

Problems Where It's Used

1. Find a Pair with a Given Sum in a Sorted Array
2. Remove Duplicates from a Sorted Array
3. Merge Two Sorted Arrays Without Extra Space
4. Checking if a String is a Palindrome

```
function hasPairWithSum(arr: number[], target: number): boolean {
  let left = 0;
  let right = arr.length - 1;

  while (left < right) {
    const currSum = arr[left] + arr[right];

    if (currSum === target) {
      return true;
    } else if (currSum < target) {
      left++;
    } else {
      right--;
    }
  }

  return false;
}

// Example usage
console.log(hasPairWithSum([1, 2, 3, 4, 6], 6)); // true
```

Sliding Window Technique

Concept

- Optimizes problems involving **subarrays** or **substrings** by maintaining a “window” of elements.
- Instead of recomputing for each subarray, it **expands and shrinks the window**.

Problems Where It’s Used

1. Find the Maximum Sum of a Subarray of Size K
2. Find the Longest Substring Without Repeating Characters
3. Find the Smallest Subarray with Sum ≥ X

Longest Subarray with Sum at Most K (Sliding Window)

```
function longestSubarray(arr: number[], k: number): number {
  let left = 0;
  let currSum = 0;
  let maxLen = 0;

  for (let right = 0; right < arr.length; right++) {
    currSum += arr[right];

    while (currSum > k) {
      currSum -= arr[left];
      left++;
    }

    maxLen = Math.max(maxLen, right - left + 1);
  }

  return maxLen;
}

// Example usage
console.log(longestSubarray([3, 1, 2, 1, 4, 5], 5)); // 3
```

Prefix Sum & Difference Arrays

Concept

- **Prefix Sum: Stores cumulative sums to quickly compute sum of subarrays.**
- **Difference Array: Efficiently applies range updates.**

Problems Where It’s Used

1. Find Sum of Any Subarray in Constant Time
2. Range Update Queries in Constant Time
3. Number of Subarrays with a Given Sum

Find Equilibrium Index

```
function findEquilibrium(arr: number[]): number {
  const totalSum = arr.reduce((acc, num) => acc + num, 0);
  let leftSum = 0;

  for (let i = 0; i < arr.length; i++) {
    if (leftSum === totalSum - leftSum - arr[i]) {
      return i;
    }
    leftSum += arr[i];
  }

  return -1;
}

// Example usage
console.log(findEquilibrium([-7, 1, 5, 2, -4, 3, 0])); // 3
```

Kadane’s Algorithm (Maximum Subarray Sum)

Concept

- Finds the **maximum sum subarray** in **O(N)**.
- Apply it when the window is not given and it’s an **unsorted array**.
- Uses **dynamic programming** to maintain a running sum.

```
function maxSubarraySum(arr: number[]): number {
  let maxSum = -Infinity;
  let currSum = 0;

  for (const num of arr) {
    currSum = Math.max(num, currSum + num);
    maxSum = Math.max(maxSum, currSum);
  }

  return maxSum;
}
```

Practice Questions

---

1. Find Pair with Target Sum (Two Pointers)

Problem:
Given a **sorted array** and a target sum, check if there exists a pair of numbers that add up to the target.

Example:

- **Input:** `arr = [1, 2, 3, 4, 6]`, `target = 6`
- **Output:** `True (Pair: (2, 4))`

---

2. Remove Duplicates from Sorted Array (Two Pointers)

Problem:
Given a sorted array, remove duplicates **in-place** such that each element appears only once.

Example:

- **Input:** `[1, 1, 2, 2, 3]`
- **Output:** `[1, 2, 3]`

---

3. Max Consecutive Ones (Sliding Window)

Problem:
Given a binary array, find the **maximum number of consecutive 1s** you can get by flipping **at most one `0`**.

Example:

- **Input:** `[1, 1, 0, 1, 1, 1]`
- **Output:** `5`

---

4. Longest Subarray with Sum at Most K (Sliding Window)

Problem:
Find the **longest subarray** whose **sum does not exceed K**.

Example:

- Input: `[3, 1, 2, 1, 4, 5]`, `K = 5`
- Output: `3` (Subarray `[3, 1, 2]`)

5. Prefix Sum Range Query

Problem:
Given an array, **precompute** prefix sums to answer multiple range sum queries efficiently.

Example:

- Input: `arr = [1, 2, 3, 4, 5]`, `queries = [(1, 3), (2, 4)]`
- Output: `[9, 12]`

---

6. Find Equilibrium Index (Prefix Sum)

Problem:
An **equilibrium index** in an array is an index where the sum of elements to the **left** equals the sum to the **right**.

Example:

- Input: `[-7, 1, 5, 2, -4, 3, 0]`
- Output: `3`

---

7. Maximum Subarray Sum (Kadane's Algorithm)

Problem:
Find the contiguous subarray with the **maximum sum**.

Example:

- Input: `[-2, 1, -3, 4, -1, 2, 1, -5, 4]`
- Output: `6` (Subarray `[4, -1, 2, 1]`)

---

8. Maximum Product Subarray (Kadane’s Variant)

Problem:
Find the contiguous subarray that has the **maximum product**.

Example:

- Input: `[2, 3, -2, 4]`
- Output: `6` (Subarray `[2, 3]`)

---

9. Minimum Size Subarray Sum (Sliding Window)

Problem:
Find the **minimum length** of a contiguous subarray whose sum is **at least** target.

Example:

- Input: `[2, 3, 1, 2, 4, 3]`, `target = 7`
- Output: `2` (Subarray `[4, 3]`)

---

10. Find Missing Number (Prefix Sum)

Problem:
Find the **missing number** in an array of size N containing numbers from 1 to N.

Example:

- Input: `[3, 7, 1, 2, 8, 4, 5]`, `N = 8`
- Output: `6`

---

11. Count Subarrays with Given Sum (Prefix Sum)

Problem:
Given an array and a sum `S`, find the number of **subarrays** that sum up to `S`.

Example:

- Input: `[1, 2, 3, 4]`, `S = 6`
- Output: `2` (Subarrays: `[1, 2, 3]`, `[2, 4]`)

---

12. Maximum Sum Circular Subarray (Kadane's Algorithm)

Problem:
Find the **maximum subarray sum** considering the array to be **circular**.

Example:

- Input: `[5, -3, 5]`
- Output: `10`

---

13. Find Subarray with Zero Sum (Prefix Sum)

Problem:
Find if there exists a **subarray** with sum **zero**.

Example:

- Input: `[4, 2, -3, 1, 6]`
- Output: `True` (Subarray `[2, -3, 1]`)

---

14. Find Maximum Average Subarray (Sliding Window)

Problem:
Find the contiguous subarray of size `K` with the **maximum average**.

Example:

- Input: `[1, 12, -5, -6, 50, 3]`, `K = 4`
- Output: `12.75`

---

15. Merge Two Sorted Arrays (Two Pointers)

Problem:
Given two sorted arrays, merge them **in-place** without using extra space.

Example:

- Input: `arr1 = [1, 3, 5]`, `arr2 = [2, 4, 6]`
- Output: `[1, 2, 3, 4, 5, 6]`

---

16. Find Longest Substring Without Repeating Characters (Sliding Window)

Problem:
Given a string, find the **length** of the longest substring without repeating characters.

Example:

- Input: `"abcabcbb"`
- Output: `3` (Substring `"abc"`)

---

17. Smallest Window Containing All Characters of Another String

Problem:
Find the **smallest substring** in `s1` that contains all characters of `s2`.

Example:

- Input: `s1 = "ADOBECODEBANC"`, `s2 = "ABC"`
- Output: `"BANC"`

---

18. Shortest Unsorted Continuous Subarray

Problem:
Find the **shortest contiguous subarray** that needs to be sorted to make the entire array sorted.

Example:

- Input: `[2, 6, 4, 8, 10, 9, 15]`
- Output: `5` (Subarray `[6, 4, 8, 10, 9]`)

---

19. Find K-th Smallest Element in Sorted Arrays (Two Pointers)

Problem:
Given two sorted arrays, find the **K-th smallest** element in the merged array.

Example:

- Input: `arr1 = [2, 3, 6, 7]`, `arr2 = [1, 4, 5, 8]`, `K = 5`
- Output: `5`

---

20. Count Number of Substrings with At Most K Distinct Characters (Sliding Window)

Problem:
Find the **number of substrings** with at most **K distinct characters**.

Example:

- Input: `"aabacbebebe"`, `K = 3`
- Output: `23`

Solutions:

1. Find Pair with Target Sum (Two Pointers)

```
function hasPairWithSum(arr: number[], target: number): boolean {
  let left = 0, right = arr.length - 1;
  while (left < right) {
    const sum = arr[left] + arr[right];
    if (sum === target) return true;
    sum < target ? left++ : right--;
  }
  return false;
}
```

2. Remove Duplicates from Sorted Array (In-place)

```
function removeDuplicates(arr: number[]): number[] {
  let i = 0;
  for (let j = 1; j < arr.length; j++) {
    if (arr[i] !== arr[j]) arr[++i] = arr[j];
  }
  return arr.slice(0, i + 1);
}
```

3. Max Consecutive Ones (Sliding Window with Flip)

```
function maxConsecutiveOnes(nums: number[]): number {
  let left = 0, zeroCount = 0, maxLen = 0;
  for (let right = 0; right < nums.length; right++) {
    if (nums[right] === 0) zeroCount++;
    while (zeroCount > 1) {
      if (nums[left++] === 0) zeroCount--;
    }
    maxLen = Math.max(maxLen, right - left + 1);
  }
  return maxLen;
}
```

4. Longest Subarray with Sum ≤ K (Sliding Window)

```
function longestSubarrayAtMostK(arr: number[], k: number): number {
  let left = 0, sum = 0, maxLen = 0;
  for (let right = 0; right < arr.length; right++) {
    sum += arr[right];
    while (sum > k) sum -= arr[left++];
    maxLen = Math.max(maxLen, right - left + 1);
  }
  return maxLen;
}
```

5. Prefix Sum Range Query

```
function prefixSumQueries(arr: number[], queries: [number, number][]):
number[] {
  const prefix = [0];
  for (let num of arr) prefix.push(prefix[prefix.length - 1] + num);
  return queries.map(([l, r]) => prefix[r + 1] - prefix[l]);
}
```

---

6. Find Equilibrium Index

```
function findEquilibrium(arr: number[]): number {
  const total = arr.reduce((a, b) => a + b, 0);
  let left = 0;
  for (let i = 0; i < arr.length; i++) {
    if (left === total - left - arr[i]) return i;
    left += arr[i];
  }
  return -1;
}
```

7. Maximum Subarray Sum (Kadane's)

```
function maxSubarraySum(arr: number[]): number {
  let max = -Infinity, curr = 0;
  for (let n of arr) {
    curr = Math.max(n, curr + n);
    max = Math.max(max, curr);
  }
  return max;
}
```

---

8. Maximum Product Subarray

```
function maxProductSubarray(nums: number[]): number {
  let maxProd = nums[0], minProd = nums[0], result = nums[0];
  for (let i = 1; i < nums.length; i++) {
    const curr = nums[i];
    [maxProd, minProd] = [
      Math.max(curr, curr * maxProd, curr * minProd),
      Math.min(curr, curr * maxProd, curr * minProd),
    ];
    result = Math.max(result, maxProd);
  }
  return result;
}
```

9. Minimum Size Subarray Sum ≥ Target

```
function minSubarrayLen(target: number, nums: number[]): number {
  let sum = 0, left = 0, minLen = Infinity;
  for (let right = 0; right < nums.length; right++) {
    sum += nums[right];
    while (sum >= target) {
      minLen = Math.min(minLen, right - left + 1);
      sum -= nums[left++];
    }
  }
  return minLen === Infinity ? 0 : minLen;
}
```

10. Find Missing Number (1 to N)

```
function findMissingNumber(arr: number[], n: number): number {
  const expectedSum = (n * (n + 1)) / 2;
  const actualSum = arr.reduce((a, b) => a + b, 0);
  return expectedSum - actualSum;
}
```

11. Count Subarrays with Given Sum

```
function countSubarraysWithSum(nums: number[], k: number): number {
  const map = new Map<number, number>();
  map.set(0, 1);
  let count = 0, sum = 0;
  for (let num of nums) {
    sum += num;
    count += map.get(sum - k) || 0;
    map.set(sum, (map.get(sum) || 0) + 1);
  }
  return count;
}
```

12. Maximum Sum Circular Subarray

```
function maxCircularSubarraySum(nums: number[]): number {
  const kadane = (arr: number[]) => {
    let max = arr[0], curr = arr[0];
    for (let i = 1; i < arr.length; i++) {
      curr = Math.max(arr[i], curr + arr[i]);
      max = Math.max(max, curr);
    }
    return max;
  };
  const total = nums.reduce((a, b) => a + b);
  const maxKadane = kadane(nums);
  const minKadane = kadane(nums.map(n => -n));
  const wrap = total + minKadane; // total - (-min)
  return maxKadane < 0 ? maxKadane : Math.max(maxKadane, wrap);
}
```

13. Subarray with Zero Sum

```
function hasZeroSumSubarray(nums: number[]): boolean {
  const seen = new Set<number>();
  let sum = 0;
  for (let num of nums) {
    sum += num;
    if (sum === 0 || seen.has(sum)) return true;
    seen.add(sum);
  }
  return false;
}
```

14. Max Average Subarray of Size K

```
function findMaxAverage(nums: number[], k: number): number {
  let sum = 0;
  for (let i = 0; i < k; i++) sum += nums[i];
  let maxSum = sum;
  for (let i = k; i < nums.length; i++) {
    sum = sum + nums[i] - nums[i - k];
    maxSum = Math.max(maxSum, sum);
  }
  return maxSum / k;
}
```

15. Merge Two Sorted Arrays In-Place

```
function mergeSortedArrays(arr1: number[], arr2: number[]): number[] {
  let i = 0, j = 0, merged: number[] = [];
  while (i < arr1.length && j < arr2.length) {
    if (arr1[i] < arr2[j]) merged.push(arr1[i++]);
    else merged.push(arr2[j++]);
  }
  return [...merged, ...arr1.slice(i), ...arr2.slice(j)];
}
```

16. Longest Substring Without Repeating Characters

```
function lengthOfLongestSubstring(s: string): number {
  const map = new Map<string, number>();
  let start = 0, maxLen = 0;
  for (let end = 0; end < s.length; end++) {
    if (map.has(s[end]) && map.get(s[end])! >= start) {
      start = map.get(s[end])! + 1;
    }
    map.set(s[end], end);
    maxLen = Math.max(maxLen, end - start + 1);
  }
  return maxLen;
}
```

17. Minimum Window Substring

```
function minWindow(s: string, t: string): string {
  const need = new Map<string, number>();
  for (let c of t) need.set(c, (need.get(c) || 0) + 1);
  let have = 0, needCount = need.size;
  const window = new Map<string, number>();
  let res = "", minLen = Infinity;
  let left = 0;
  for (let right = 0; right < s.length; right++) {
    const c = s[right];
    window.set(c, (window.get(c) || 0) + 1);
    if (need.has(c) && window.get(c) === need.get(c)) have++;

    while (have === needCount) {
      if (right - left + 1 < minLen) {
        minLen = right - left + 1;
        res = s.substring(left, right + 1);
      }
      const lChar = s[left++];
      if (need.has(lChar)) {
        if (window.get(lChar) === need.get(lChar)) have--;
        window.set(lChar, window.get(lChar)! - 1);
      }
    }
  }
  return res;
}
```

18. Shortest Unsorted Subarray

```
function findUnsortedSubarray(nums: number[]): number {
  let left = 0, right = nums.length - 1;
  while (left < nums.length - 1 && nums[left] <= nums[left + 1]) left++;
  if (left === nums.length - 1) return 0;
  while (right > 0 && nums[right] >= nums[right - 1]) right--;

  let min = Infinity, max = -Infinity;
  for (let i = left; i <= right; i++) {
    min = Math.min(min, nums[i]);
    max = Math.max(max, nums[i]);
  }

  while (left > 0 && nums[left - 1] > min) left--;
  while (right < nums.length - 1 && nums[right + 1] < max) right++;

  return right - left + 1;
}
```

19. Find K-th Smallest in Sorted Arrays

(Use min-heap or binary search approach; here's a simple merge-based solution)

```
function findKthSmallest(arr1: number[], arr2: number[], k: number):
number {
  let i = 0, j = 0;
  while (true) {
    if (i === arr1.length) return arr2[j + k - 1];
    if (j === arr2.length) return arr1[i + k - 1];
    if (k === 1) return Math.min(arr1[i], arr2[j]);

    const half = Math.floor(k / 2);
    const newI = Math.min(i + half, arr1.length) - 1;
    const newJ = Math.min(j + half, arr2.length) - 1;
    if (arr1[newI] < arr2[newJ]) {
      k -= (newI - i + 1);
      i = newI + 1;
    } else {
      k -= (newJ - j + 1);
      j = newJ + 1;
    }
  }
}
```

20. Count Substrings with At Most K Distinct Characters

```
function countSubstringsAtMostK(s: string, k: number): number {
  const map = new Map<string, number>();
  let left = 0, count = 0;
  for (let right = 0; right < s.length; right++) {
    map.set(s[right], (map.get(s[right]) || 0) + 1);
    while (map.size > k) {
      const c = s[left];
      map.set(c, map.get(c)! - 1);
      if (map.get(c) === 0) map.delete(c);
      left++;
    }
    count += right - left + 1;
  }
  return count;
}
```