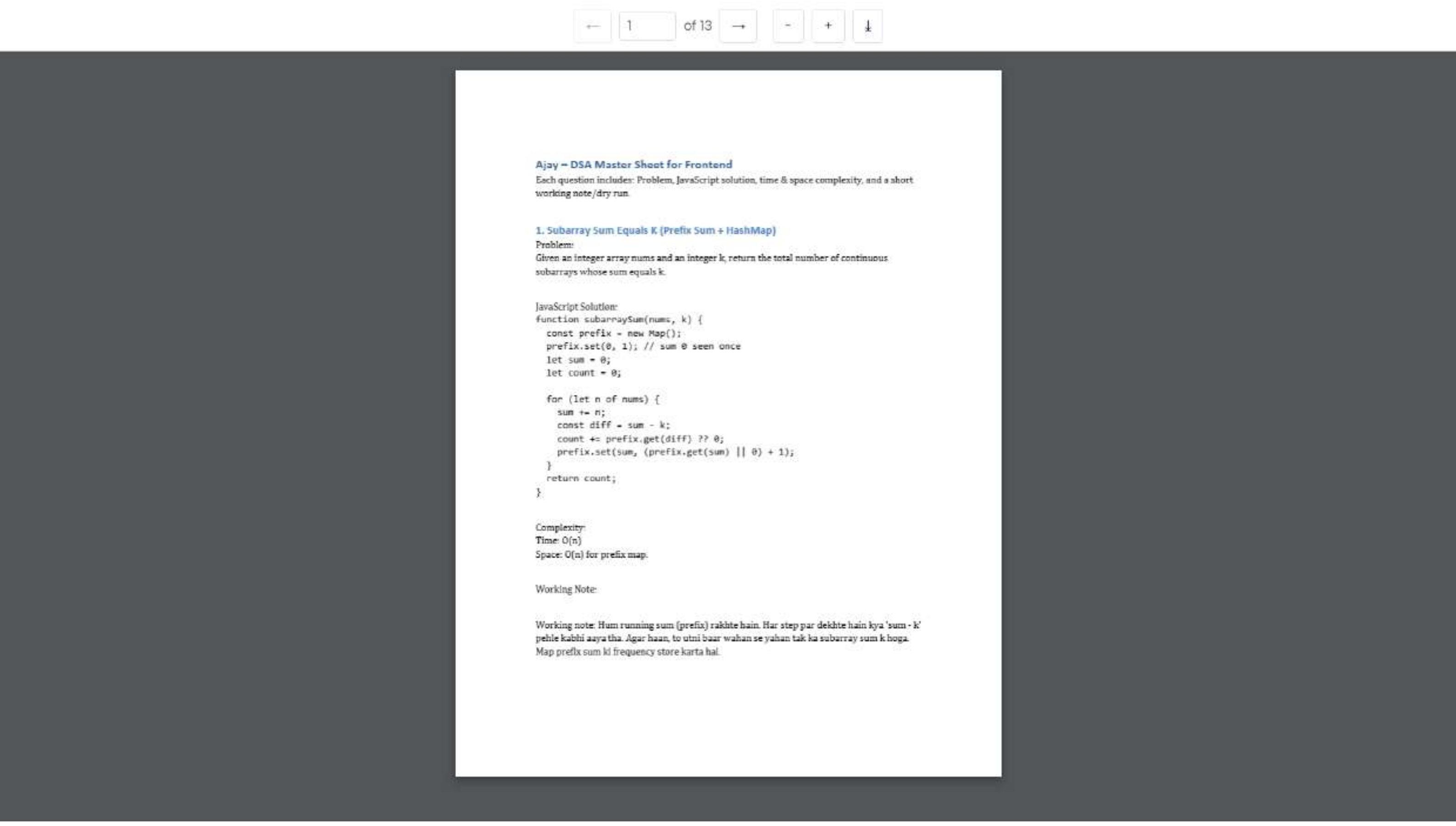


Ajay – DSA Master Sheet for Frontend

Each question includes: Problem, JavaScript solution, time & space complexity, and a short working note/dry run.

1. Subarray Sum Equals K (Prefix Sum + HashMap)

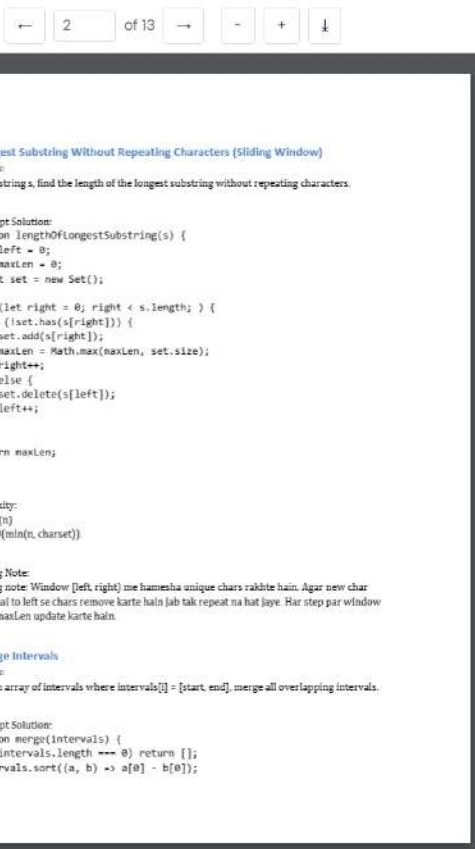
Problem:
Given an integer array nums and an integer k, return the total number of continuous subarrays whose sum equals k.

JavaScript Solution:
```
function subarraySum(nums, k) {
  const prefix = new Map();
  prefix.set(0, 1); // sum 0 seen once
  let sum = 0;
  let count = 0;

  for (let n of nums) {
    sum += n;
    const diff = sum - k;
    count += prefix.get(diff) ?? 0;
    prefix.set(sum, (prefix.get(sum) || 0) + 1);
  }
  return count;
}
```

Complexity:
Time: O(n)
Space: O(n) for prefix map.

Working Note:

Working note: Hum running sum (prefix) rakhte hain. Har step par dekhte hain kya 'sum - k' pehle kabhi aaya tha. Agar haan, to utni baar wahan se yahan tak ka subarray sum k hoga. Map prefix sum ki frequency store karta hai.



2. Longest Substring Without Repeating Characters (Sliding Window)

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

JavaScript Solution:
```
function lengthOfLongestSubstring(s) {
  let left = 0;
  let maxLen = 0;
  const set = new Set();

  for (let right = 0; right < s.length; ) {
    if (!set.has(s[right])) {
      set.add(s[right]);
      maxLen = Math.max(maxLen, set.size);
      right++;
    } else {
      set.delete(s[left]);
      left++;
    }
  }
  return maxLen;
}
```

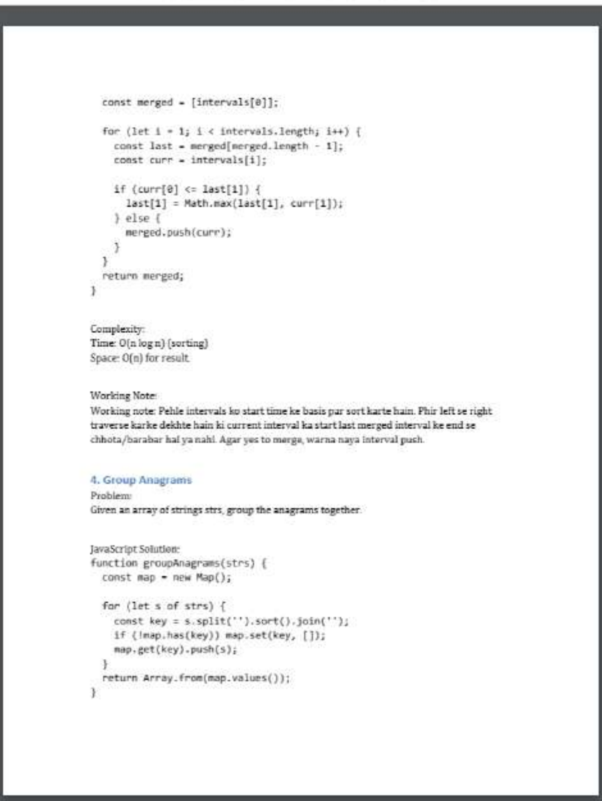
Complexity:
Time: O(n)
Space: O(min(n, charset))

Working Note:
Working note: Window [left, right] me hamesha unique chars rakhte hain. Agar new char repeat hai to left se chars remove karte hain jab tak repeat na hat jaye. Har step par window size se maxLen update karte hain.

3. Merge Intervals

Problem:
Given an array of intervals where intervals[i] = [start, end], merge all overlapping intervals.

JavaScript Solution:
```
function merge(intervals) {
  if (intervals.length === 0) return [];
  intervals.sort((a, b) => a[0] - b[0]);
```



```
  const merged = [intervals[0]];

  for (let i = 1; i < intervals.length; i++) {
    const last = merged[merged.length - 1];
    const curr = intervals[i];

    if (curr[0] <= last[1]) {
      last[1] = Math.max(last[1], curr[1]);
    } else {
      merged.push(curr);
    }
  }
  return merged;
}
```

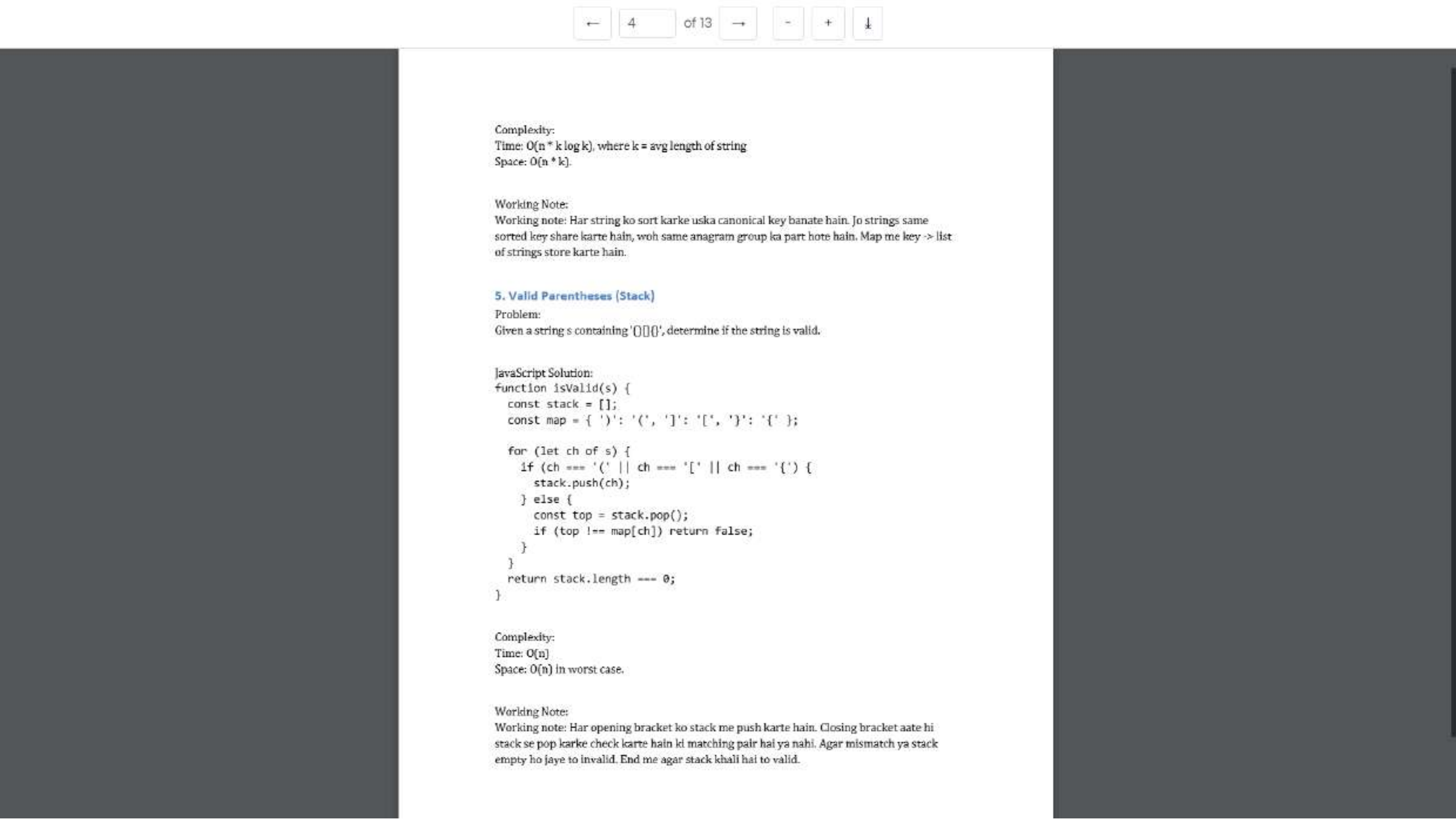
Complexity:
Time: O(n log n) (sorting)
Space: O(n) for result.

Working Note:
Working note: Pehle intervals ko start time ke basis par sort karte hain. Phir left se right traverse karke dekhte hain ki current interval ka start last merged interval ke end se chhota/barabar hai ya nahi. Agar yes to merge, warna naya interval push.

4. Group Anagrams

Problem:
Given an array of strings strs, group the anagrams together.

JavaScript Solution:
```
function groupAnagrams(strs) {
  const map = new Map();

  for (let s of strs) {
    const key = s.split('').sort().join('');
    if (!map.has(key)) map.set(key, []);
    map.get(key).push(s);
  }
  return Array.from(map.values());
}
```



Complexity:
Time: O(n * k log k), where k = avg length of string
Space: O(n * k).

Working Note:
Working note: Har string ko sort karke uska canonical key banate hain. Jo strings same sorted key share karte hain, woh same anagram group ka part hote hain. Map me key -> list of strings store karte hain.

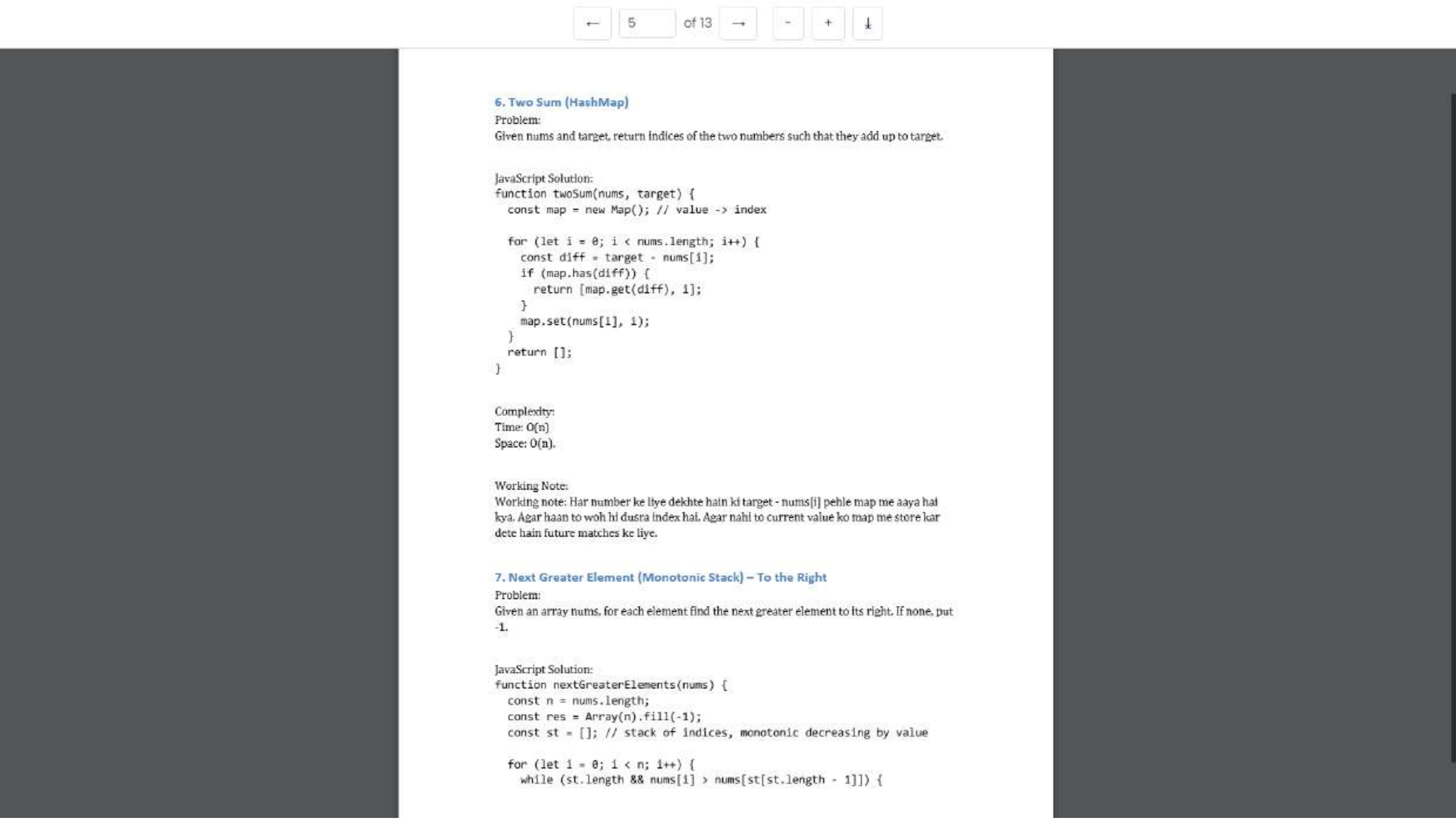
5. Valid Parentheses (Stack)

Problem:
Given a string s containing '()[]{}', determine if the string is valid.

JavaScript Solution:
```
function isValid(s) {
  const stack = [];
  const map = { ')': '(', ']': '[', '}': '{' };

  for (let ch of s) {
    if (ch === '(' || ch === '[' || ch === '{') {
      stack.push(ch);
    } else {
      const top = stack.pop();
      if (top !== map[ch]) return false;
    }
  }
  return stack.length === 0;
}
```

Complexity:
Time: O(n)
Space: O(n) in worst case.

Working Note:
Working note: Har opening bracket ko stack me push karte hain. Closing bracket aate hi stack se pop karke check karte hain ki matching pair hai ya nahi. Agar mismatch ya stack empty ho jaye to invalid. End me agar stack khali hai to valid.



6. Two Sum (HashMap)

Problem:
Given nums and target, return indices of the two numbers such that they add up to target.

JavaScript Solution:
```
function twoSum(nums, target) {
  const map = new Map(); // value -> index

  for (let i = 0; i < nums.length; i++) {
    const diff = target - nums[i];
    if (map.has(diff)) {
      return [map.get(diff), i];
    }
    map.set(nums[i], i);
  }
  return [];
}
```

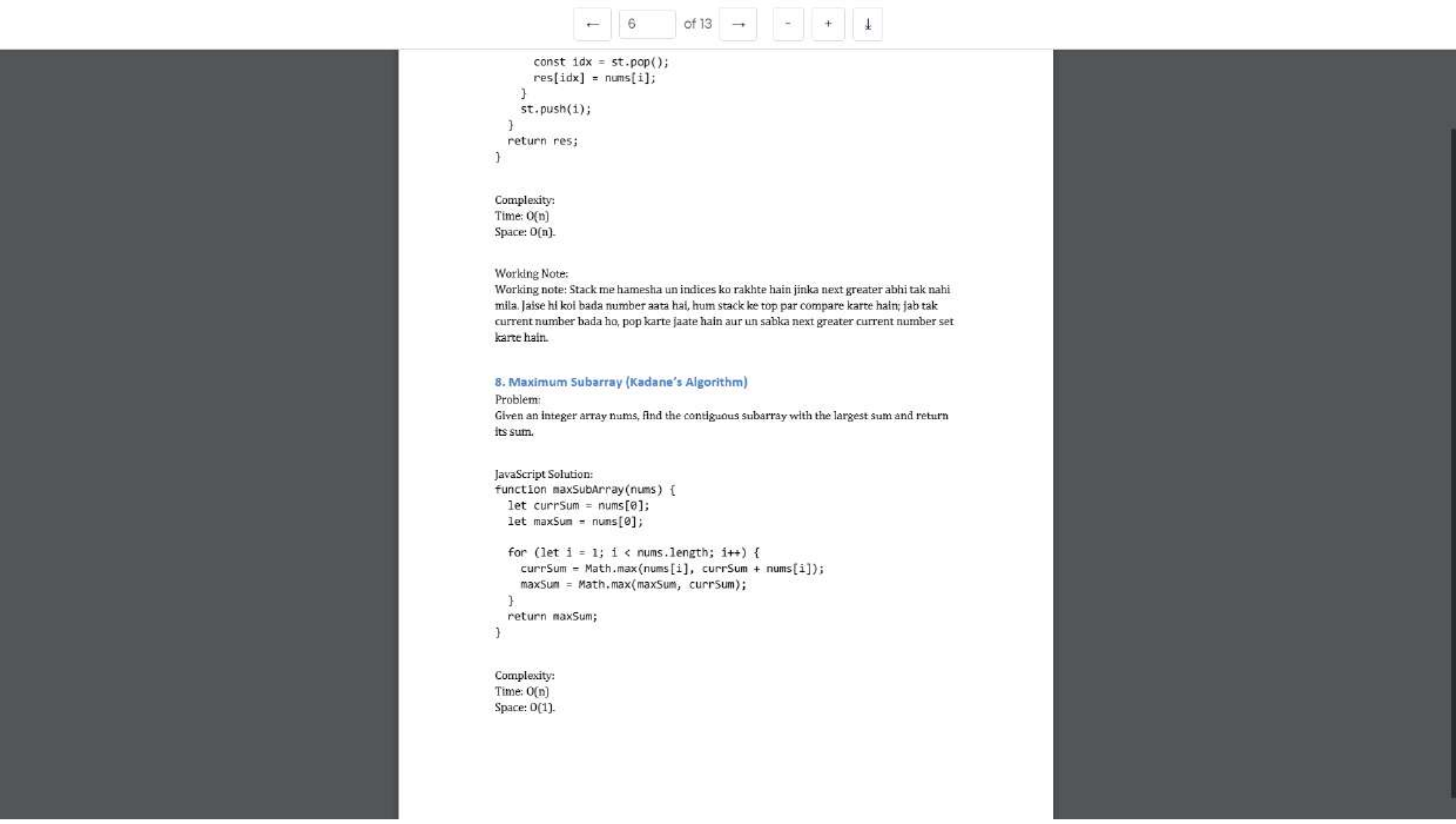
Complexity:
Time: O(n)
Space: O(n).

Working Note:
Working note: Har number ke liye dekhte hain ki target - nums[i] pehle map me aaya hai kya. Agar haan to woh hi dusra index hai. Agar nahi to current value ko map me store kar dete hain future matches ke liye.

7. Next Greater Element (Monotonic Stack) – To the Right

Problem:
Given an array nums, for each element find the next greater element to its right. If none, put -1.

JavaScript Solution:
```
function nextGreaterElements(nums) {
  const n = nums.length;
  const res = Array(n).fill(-1);
  const st = []; // stack of indices, monotonic decreasing by value

  for (let i = 0; i < n; i++) {
    while (st.length && nums[i] > nums[st[st.length - 1]]) {
```



```
      const idx = st.pop();
      res[idx] = nums[i];
    }
    st.push(i);
  }
  return res;
}
```

Complexity:
Time: O(n)
Space: O(n).

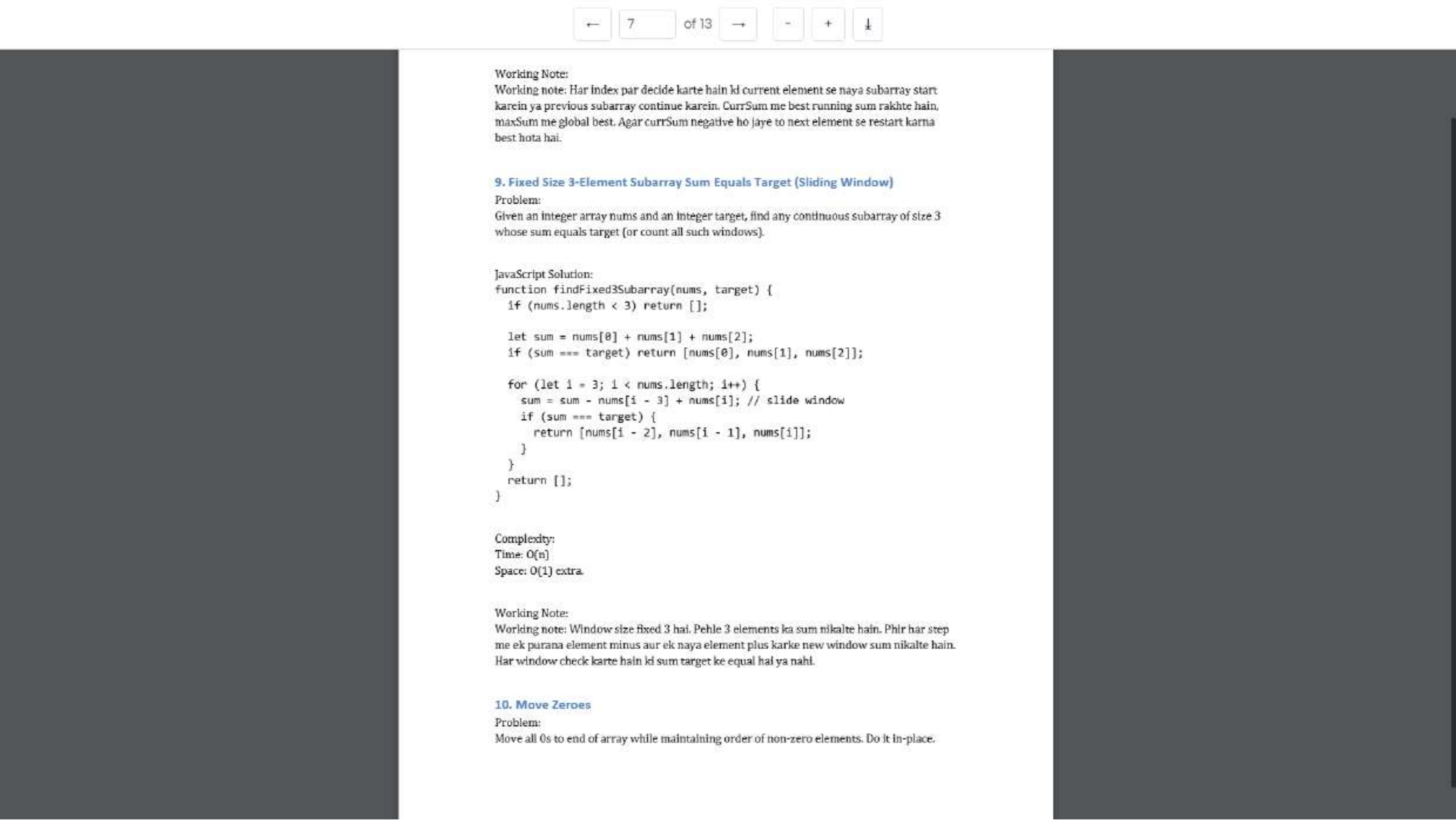
Working Note:
Working note: Stack me hamesha un indices ko rakhte hain jinka next greater abhi tak nahi mila. Jaise hi koi bada number aata hai, hum stack ke top par compare karte hain; jab tak current number bada ho, pop karte jaate hain aur un sabka next greater current number set karte hain.

8. Maximum Subarray (Kadane's Algorithm)

Problem:
Given an integer array nums, find the contiguous subarray with the largest sum and return its sum.

JavaScript Solution:
```
function maxSubArray(nums) {
  let currSum = nums[0];
  let maxSum = nums[0];

  for (let i = 1; i < nums.length; i++) {
    currSum = Math.max(nums[i], currSum + nums[i]);
    maxSum = Math.max(maxSum, currSum);
  }
  return maxSum;
}
```

Complexity:
Time: O(n)
Space: O(1).



Working Note:
Working note: Har index par decide karte hain ki current element se naya subarray start karein ya previous subarray continue karein. CurrSum me best running sum rakhte hain, maxSum me global best. Agar currSum negative ho jaye to next element se restart karna best hota hai.

9. Fixed Size 3-Element Subarray Sum Equals Target (Sliding Window)

Problem:
Given an integer array nums and an integer target, find any continuous subarray of size 3 whose sum equals target (or count all such windows).

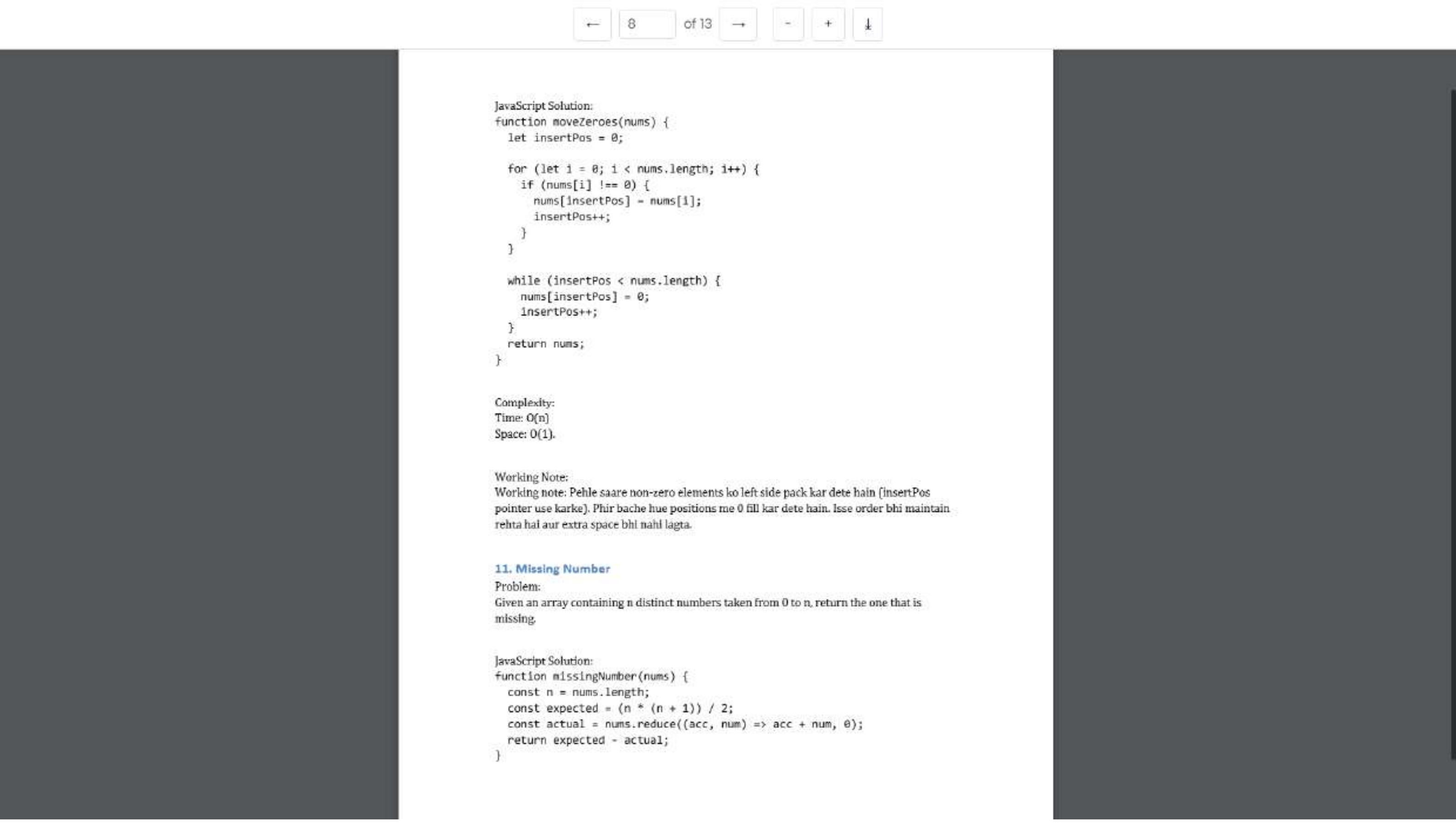
JavaScript Solution:
```
function findFixed3Subarray(nums, target) {
  if (nums.length < 3) return [];

  let sum = nums[0] + nums[1] + nums[2];
  if (sum === target) return [nums[0], nums[1], nums[2]];

  for (let i = 3; i < nums.length; i++) {
    sum = sum - nums[i - 3] + nums[i]; // slide window
    if (sum === target) {
      return [nums[i - 2], nums[i - 1], nums[i]];
    }
  }
  return [];
}
```

Complexity:
Time: O(n)
Space: O(1) extra.

Working Note:
Working note: Window size fixed 3 hai. Pehle 3 elements ka sum nikalte hain. Phir har step me ek purana element minus aur ek naya element plus karke new window sum nikalte hain. Har window check karte hain ki sum target ke equal hai ya nahi.

10. Move Zeroes

Problem:
Move all 0s to end of array while maintaining order of non-zero elements. Do it in-place.



← 8 of 13 → - + ↓

JavaScript Solution:

```
function moveZeroes(nums) {
  let insertPos = 0;

  for (let i = 0; i < nums.length; i++) {
    if (nums[i] !== 0) {
      nums[insertPos] = nums[i];
      insertPos++;
    }
  }

  while (insertPos < nums.length) {
    nums[insertPos] = 0;
    insertPos++;
  }
  return nums;
}
```

Complexity:
Time: O(n)
Space: O(1).

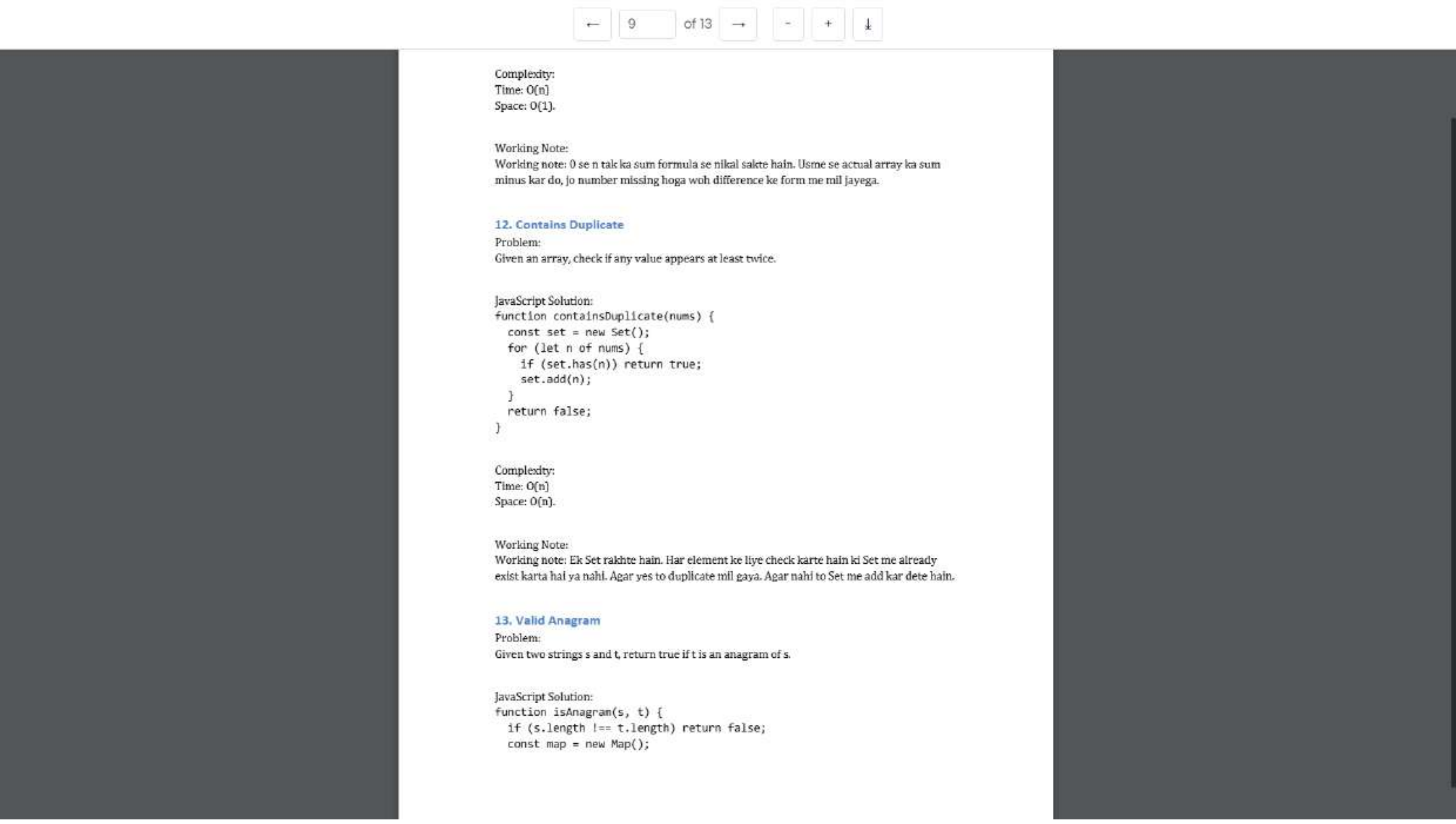
Working Note:
Working note: Pehle saare non-zero elements ko left side pack kar dete hain (insertPos pointer use karke). Phir bache hue positions me 0 fill kar dete hain. Isse order bhi maintain rehta hai aur extra space bhi nahi lagta.

11. Missing Number

Problem:
Given an array containing n distinct numbers taken from 0 to n, return the one that is missing.

JavaScript Solution:

```
function missingNumber(nums) {
  const n = nums.length;
  const expected = (n * (n + 1)) / 2;
  const actual = nums.reduce((acc, num) => acc + num, 0);
  return expected - actual;
}
```



Complexity:
Time: O(n)
Space: O(1).

Working Note:
Working note: 0 se n tak ka sum formula se nikal sakte hain. Usme se actual array ka sum minus kar do, jo number missing hoga woh difference ke form me mil jayega.

12. Contains Duplicate

Problem:
Given an array, check if any value appears at least twice.

JavaScript Solution:

```
function containsDuplicate(nums) {
  const set = new Set();
  for (let n of nums) {
    if (set.has(n)) return true;
    set.add(n);
  }
  return false;
}
```

Complexity:
Time: O(n)
Space: O(n).

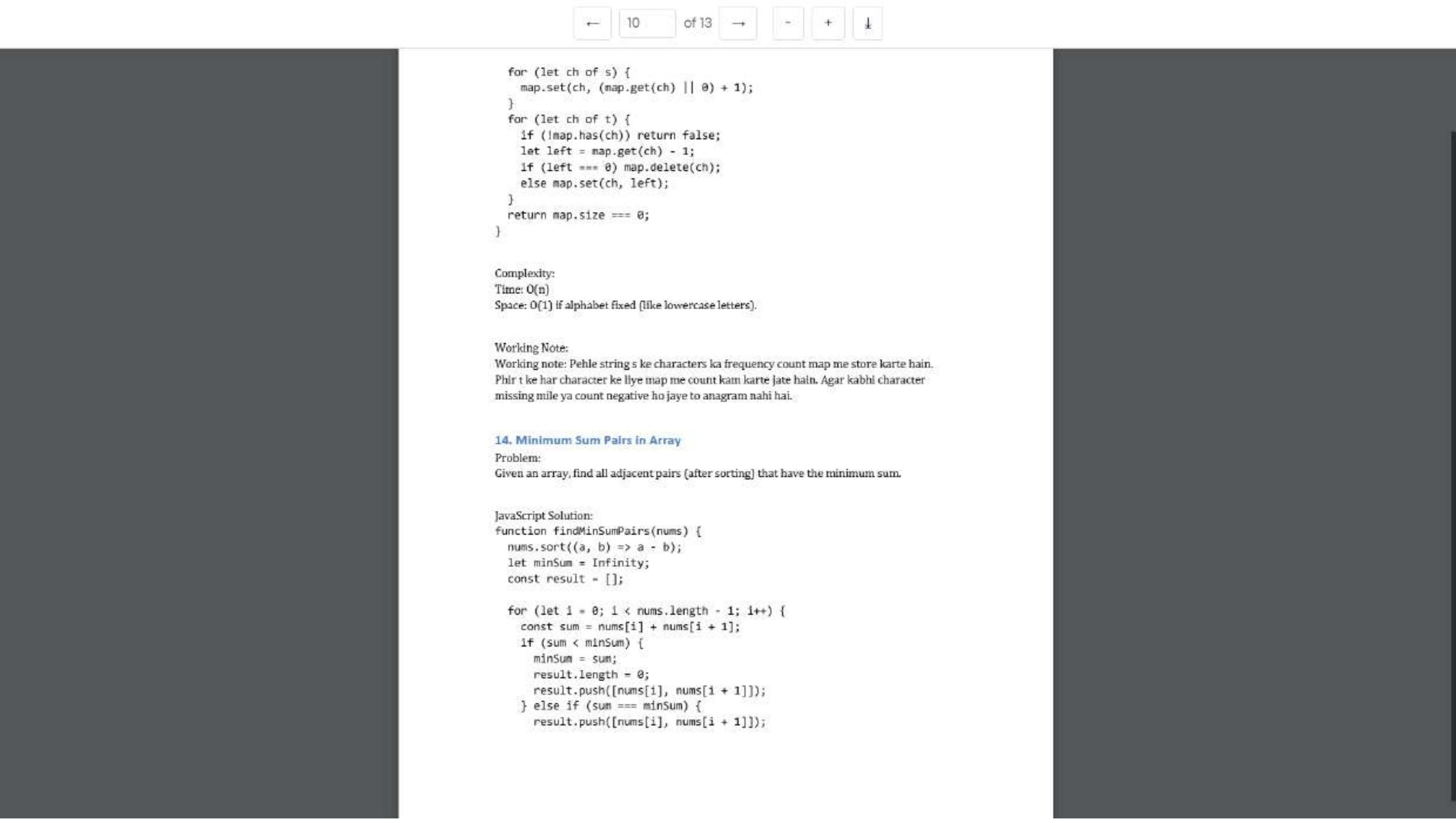
Working Note:
Working note: Ek Set rakhte hain. Har element ke liye check karte hain ki Set me already exist karta hai ya nahi. Agar yes to duplicate mil gaya. Agar nahi to Set me add kar dete hain.

13. Valid Anagram

Problem:
Given two strings s and t, return true if t is an anagram of s.

JavaScript Solution:

```
function isAnagram(s, t) {
  if (s.length !== t.length) return false;
  const map = new Map();
```



```
  for (let ch of s) {
    map.set(ch, (map.get(ch) || 0) + 1);
  }
  for (let ch of t) {
    if (!map.has(ch)) return false;
    let left = map.get(ch) - 1;
    if (left === 0) map.delete(ch);
    else map.set(ch, left);
  }
  return map.size === 0;
}
```

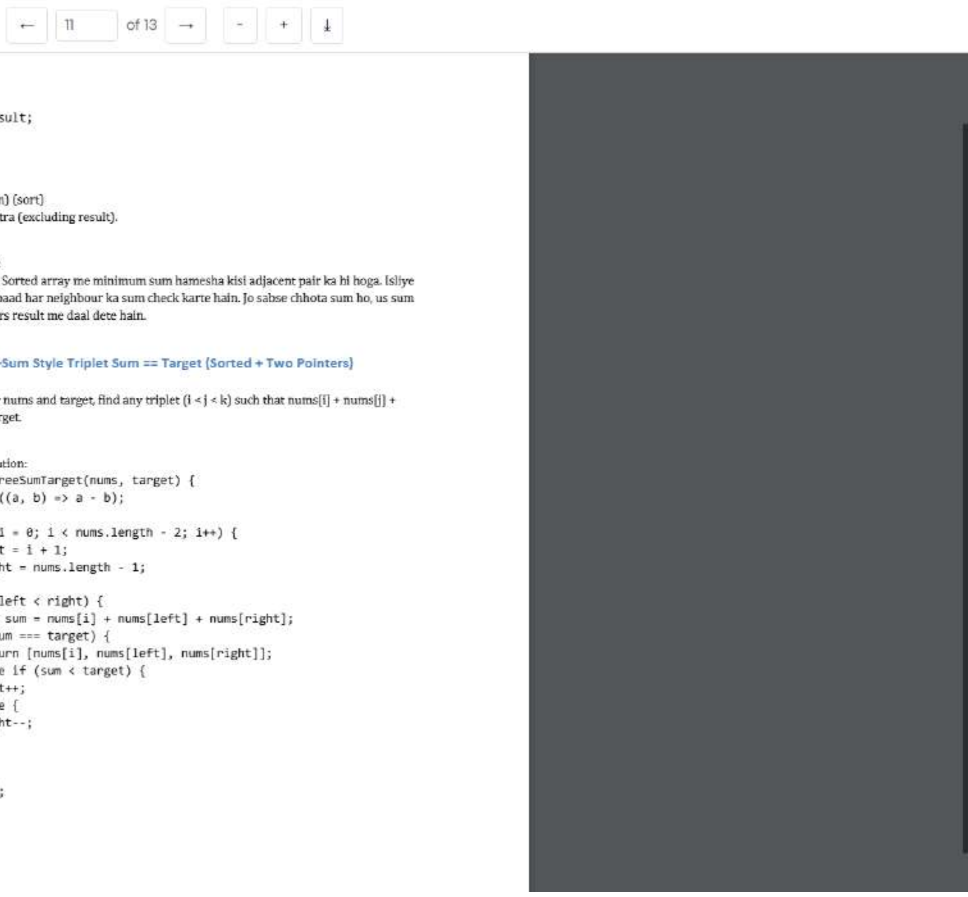
Complexity:
Time: O(n)
Space: O(1) if alphabet fixed (like lowercase letters).

Working Note:
Working note: Pehle string s ke characters ka frequency count map me store karte hain. Phir t ke har character ke liye map me count kam karte jate hain. Agar kabhi character missing mile ya count negative ho jaye to anagram nahi hai.

14. Minimum Sum Pairs in Array

Problem:
Given an array, find all adjacent pairs (after sorting) that have the minimum sum.

JavaScript Solution:
```
function findMinSumPairs(nums) {
  nums.sort((a, b) => a - b);
  let minSum = Infinity;
  const result = [];

  for (let i = 0; i < nums.length - 1; i++) {
    const sum = nums[i] + nums[i + 1];
    if (sum < minSum) {
      minSum = sum;
      result.length = 0;
      result.push([nums[i], nums[i + 1]]);
    } else if (sum === minSum) {
      result.push([nums[i], nums[i + 1]]);
```



```
    }
  }
  return result;
}
```

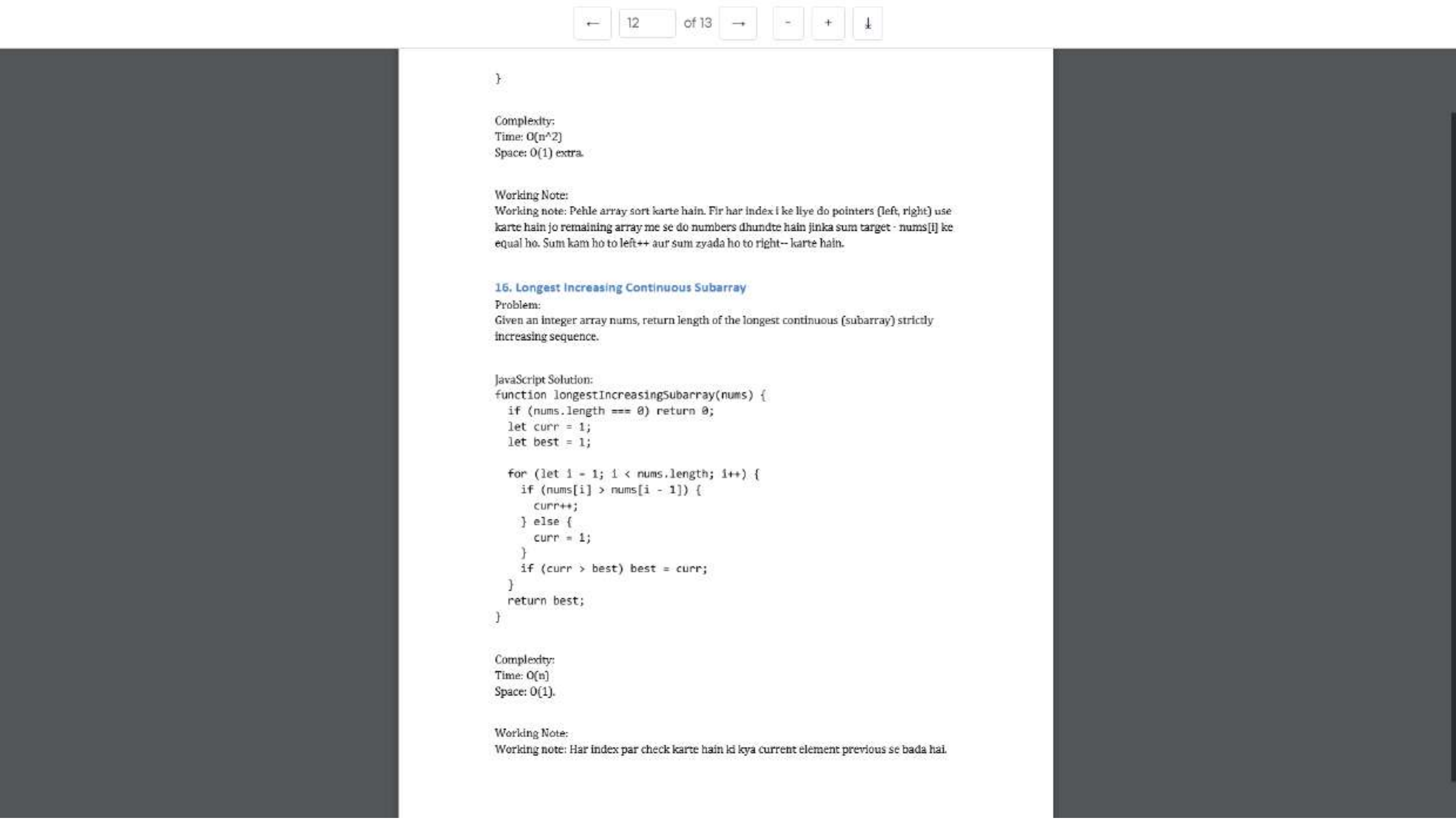
Complexity:
Time: O(n log n) (sort)
Space: O(1) extra (excluding result).

Working Note:
Working note: Sorted array me minimum sum hamesha kisi adjacent pair ka hi hoga. Isliye sort karne ke baad har neighbour ka sum check karte hain. Jo sabse chhota sum ho, us sum wale saare pairs result me daal dete hain.

15. Simple 3-Sum Style Triplet Sum == Target (Sorted + Two Pointers)

Problem:
Given an array nums and target, find any triplet (i < j < k) such that nums[i] + nums[j] + nums[k] == target.

JavaScript Solution:
```
function threeSumTarget(nums, target) {
  nums.sort((a, b) => a - b);

  for (let i = 0; i < nums.length - 2; i++) {
    let left = i + 1;
    let right = nums.length - 1;

    while (left < right) {
      const sum = nums[i] + nums[left] + nums[right];
      if (sum === target) {
        return [nums[i], nums[left], nums[right]];
      } else if (sum < target) {
        left++;
      } else {
        right--;
      }
    }
  }
  return [];
```



```
}
```

Complexity:
Time: O(n^2)
Space: O(1) extra.

Working Note:
Working note: Pehle array sort karte hain. Fir har index i ke liye do pointers (left, right) use karte hain jo remaining array me se do numbers dhundte hain jinka sum target - nums[i] ke equal ho. Sum kam ho to left++ aur sum zyada ho to right-- karte hain.

16. Longest Increasing Continuous Subarray

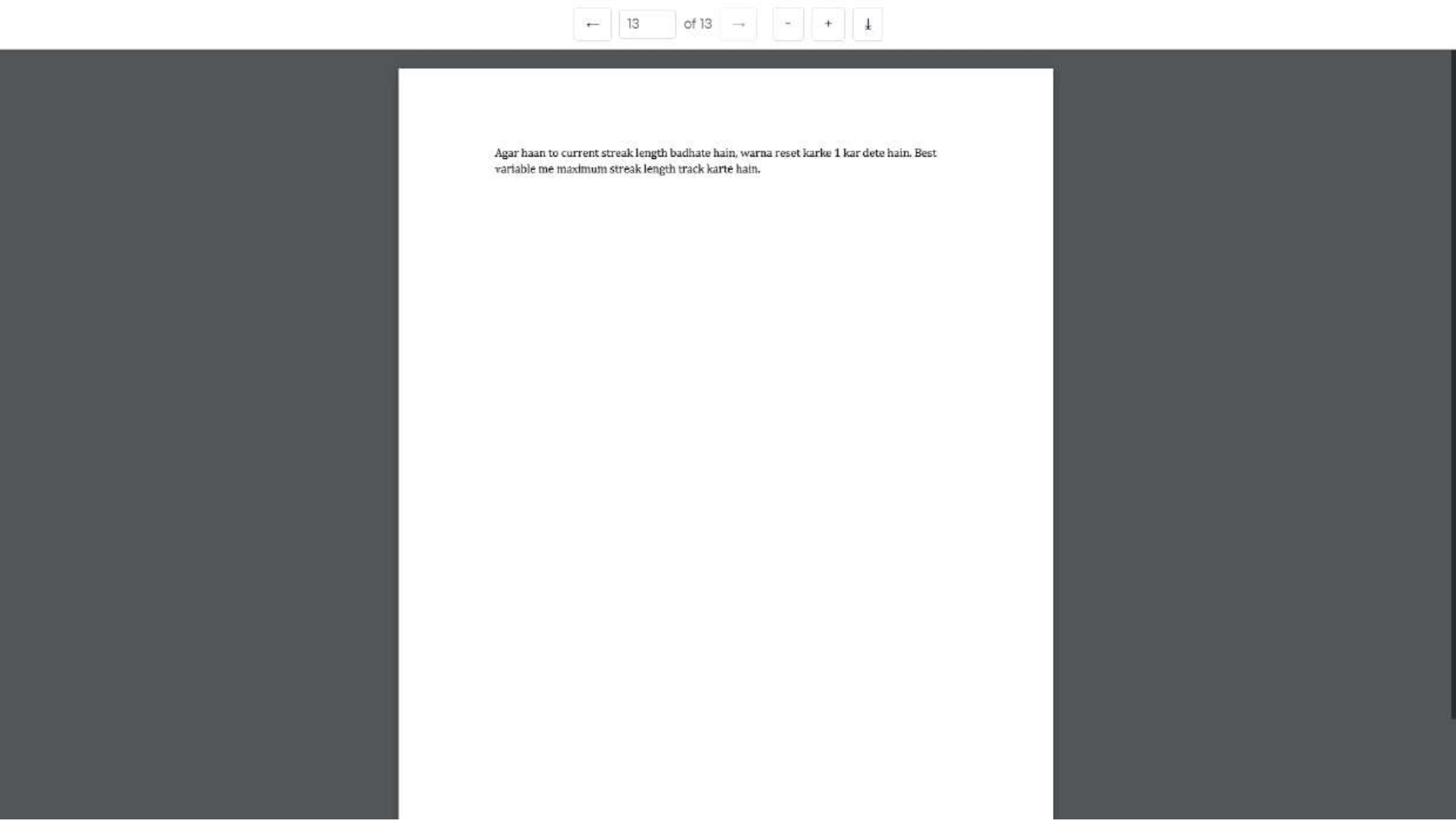
Problem:
Given an integer array nums, return length of the longest continuous (subarray) strictly increasing sequence.

JavaScript Solution:

```
function longestIncreasingSubarray(nums) {
  if (nums.length === 0) return 0;
  let curr = 1;
  let best = 1;

  for (let i = 1; i < nums.length; i++) {
    if (nums[i] > nums[i - 1]) {
      curr++;
    } else {
      curr = 1;
    }
    if (curr > best) best = curr;
  }
  return best;
}
```

Complexity:
Time: O(n)
Space: O(1).

Working Note:
Working note: Har index par check karte hain ki kya current element previous se bada hai.



Agar haan to current streak length badhate hain, warna reset karke 1 kar dete hain. Best variable me maximum streak length track karte hain.