Concept	Meaning
prefix common
Prefix Sum	Sum of elements before i
Prefix XOR	XOR of elements before i
Prefix Max/Min	Max/Min till i
Prefix Frequency	Count of elements till i
Prefix Hash	Hash of prefix (strings)
Prefix GCD	GCD of A[0..i]
Prefix AND/OR	Bitwise accumulation
Prefix Common Count	Shared items till i

Concept	Meaning
suffix common
Suffix Sum	Sum from i to end
Suffix Max/Min	Max/Min from i to end
Suffix Frequency	Count from i onward
Suffix GCD	GCD of A[i..n-1]
Suffix AND/OR	Bitwise from i to end
Suffix Hash	Hash of suffix
suffix common count

Remove one element problems
Technique	Used in
Prefix + suffix arrays	Split problems
Difference array	Range updates
Imos technique	Interval additions
Sliding window	Variable prefix
Two pointers	Prefix + suffix movement
Meet in middle	Left-right decomposition

explain all these concepts in details with proper theory proper exmaples for each and proper codes for each concepts in very very

details
give codes in cpp and python
give atleast good examples for each

Comprehensive Guide to Prefix, Suffix, and Related Techniques

1. Prefix Sum

Theory

Prefix Sum is a precomputation technique where we store cumulative sums of elements. For array `arr`, prefix sum `pref[i] = arr[0] + arr[1] + ... + arr[i]`.

Key Properties

- **Sum Query**: `sum(arr[l..r]) = pref[r] - pref[l-1]` (with pref[-1] = 0)
- **Space**: O(n)
- **Build Time**: O(n)

Applications

- Range sum queries
- Subarray sum problems
- 2D/3D prefix sums for matrix queries

Examples

Example 1: Basic Prefix Sum

```
text

Array: [2, 4, 1, 5, 3]
Prefix: [2, 6, 7, 12, 15]

Sum(1, 3) = pref[3] - pref[0] = 12 - 2 = 10
Sum(0, 2) = pref[2] - pref[-1] = 7 - 0 = 7
```

Example 2: Counting Positive Subarrays

Find number of subarrays with positive sum:

```
text

Array: [1, -2, 3, -1, 2]
Prefix: [1, -1, 2, 1, 3]
```

Code Implementation

```cpp
// C++ Implementation
#include <iostream>
#include <vector>
using namespace std;

class PrefixSum {
public:
    vector<int> prefix;

    void build(const vector<int>& arr) {
        int n = arr.size();
        prefix.resize(n);
        prefix[0] = arr[0];
        for (int i = 1; i < n; i++) {
            prefix[i] = prefix[i-1] + arr[i];
        }
    }

    int rangeSum(int l, int r) {
        if (l == 0) return prefix[r];
        return prefix[r] - prefix[l-1];
    }

    // Example: Count subarrays with sum = k
    int countSubarraysWithSumK(const vector<int>& arr, int k) {
        int n = arr.size();
        build(arr);
        unordered_map<int, int> freq;
        freq[0] = 1; // Empty subarray
        int count = 0;

        for (int i = 0; i < n; i++) {
            int current = prefix[i];
```

```
            if (freq.find(current - k) != freq.end()) {
                count += freq[current - k];
            }
            freq[current]++;
        }
        return count;
    }
};
```

python

```python
# Python Implementation
class PrefixSum:
    def __init__(self, arr):
        self.n = len(arr)
        self.prefix = [0] * self.n
        self.build(arr)

    def build(self, arr):
        self.prefix[0] = arr[0]
        for i in range(1, self.n):
            self.prefix[i] = self.prefix[i-1] + arr[i]

    def range_sum(self, l, r):
        if l == 0:
            return self.prefix[r]
        return self.prefix[r] - self.prefix[l-1]

    def count_subarrays_with_sum_k(self, arr, k):
        from collections import defaultdict
        prefix_sum = 0
        freq = defaultdict(int)
        freq[0] = 1  # Empty subarray
        count = 0

        for num in arr:
            prefix_sum += num
            count += freq.get(prefix_sum - k, 0)
            freq[prefix_sum] += 1

        return count

# Example usage
arr = [1, 2, 3, 4, 5]
ps = PrefixSum(arr)
```

```
print(f"Sum from index 1 to 3: {ps.range_sum(1, 3)}")  # 9
print(f"Total subarrays with sum 7: {ps.count_subarrays_with_sum_k(arr, 7)}")  # 2
```

2. Prefix XOR

Theory

Similar to prefix sum but using XOR operation. XOR has special properties:

- `a ⊕ a = 0`
- `a ⊕ 0 = a`
- XOR is associative and commutative

Applications

- Find subarrays with XOR = k
- Find missing number
- Swap numbers without temp variable

Examples

Example: Find subarray with XOR = k

```
text

Array: [4, 2, 2, 6, 4], k = 6
Prefix XOR: [4, 6, 4, 2, 6]

Logic: if prefix_xor[i] ^ k exists in prefix_xor[0..i-1],
then there's a subarray with XOR = k
```

Code Implementation

```
cpp

// C++ Implementation
class PrefixXOR {
public:
    vector<int> prefix;
```

```
    void build(const vector<int>& arr) {
        int n = arr.size();
        prefix.resize(n);
        prefix[0] = arr[0];
        for (int i = 1; i < n; i++) {
            prefix[i] = prefix[i-1] ^ arr[i];
        }
    }

    int rangeXOR(int l, int r) {
        if (l == 0) return prefix[r];
        return prefix[r] ^ prefix[l-1];
    }

    int countSubarraysWithXORK(const vector<int>& arr, int k) {
        unordered_map<int, int> freq;
        freq[0] = 1;
        int prefix_xor = 0;
        int count = 0;

        for (int num : arr) {
            prefix_xor ^= num;
            if (freq.find(prefix_xor ^ k) != freq.end()) {
                count += freq[prefix_xor ^ k];
            }
            freq[prefix_xor]++;
        }
        return count;
    }
};
```

python

```
# Python Implementation
class PrefixXOR:
    def __init__(self, arr):
        self.n = len(arr)
        self.prefix = [0] * self.n
        self.build(arr)

    def build(self, arr):
        self.prefix[0] = arr[0]
        for i in range(1, self.n):
            self.prefix[i] = self.prefix[i-1] ^ arr[i]

    def range_xor(self, l, r):
        if l == 0:
```

```
            return self.prefix[r]
        return self.prefix[r] ^ self.prefix[l-1]

    def count_subarrays_with_xor_k(self, arr, k):
        from collections import defaultdict
        freq = defaultdict(int)
        freq[0] = 1
        prefix_xor = 0
        count = 0

        for num in arr:
            prefix_xor ^= num
            count += freq.get(prefix_xor ^ k, 0)
            freq[prefix_xor] += 1

        return count

# Example
arr = [4, 2, 2, 6, 4]
px = PrefixXOR(arr)
print(f"XOR from 1 to 3: {px.range_xor(1, 3)}")  # 2^2^6 = 6
print(f"Subarrays with XOR 6: {px.count_subarrays_with_xor_k(arr, 6)}")  # 4
```

3. Prefix Max/Min

Theory

Store maximum/minimum value encountered till each index.

Applications

- Stock price problems (max profit)
- Trapping rainwater
- Largest rectangle in histogram

Examples

Example: Stock Buy Sell (One Transaction)

```
text
```

```
Prices: [7, 1, 5, 3, 6, 4]
Min till i: [7, 1, 1, 1, 1, 1]
Max profit = max(prices[i] - min_till_i[i])
```

Code Implementation

cpp

```cpp
// C++ Implementation
class PrefixMinMax {
public:
    vector<int> prefix_min;
    vector<int> prefix_max;

    void build(const vector<int>& arr) {
        int n = arr.size();
        prefix_min.resize(n);
        prefix_max.resize(n);

        prefix_min[0] = arr[0];
        prefix_max[0] = arr[0];

        for (int i = 1; i < n; i++) {
            prefix_min[i] = min(prefix_min[i-1], arr[i]);
            prefix_max[i] = max(prefix_max[i-1], arr[i]);
        }
    }

    // Maximum profit with one transaction
    int maxProfitOneTransaction(const vector<int>& prices) {
        int n = prices.size();
        if (n < 2) return 0;

        vector<int> min_till(n);
        min_till[0] = prices[0];
        int max_profit = 0;

        for (int i = 1; i < n; i++) {
            min_till[i] = min(min_till[i-1], prices[i]);
            max_profit = max(max_profit, prices[i] - min_till[i]);
        }

        return max_profit;
    }
```

```
    // Trapping Rain Water
    int trapWater(const vector<int>& height) {
        int n = height.size();
        if (n == 0) return 0;

        vector<int> left_max(n), right_max(n);

        left_max[0] = height[0];
        for (int i = 1; i < n; i++) {
            left_max[i] = max(left_max[i-1], height[i]);
        }

        right_max[n-1] = height[n-1];
        for (int i = n-2; i >= 0; i--) {
            right_max[i] = max(right_max[i+1], height[i]);
        }

        int water = 0;
        for (int i = 0; i < n; i++) {
            water += min(left_max[i], right_max[i]) - height[i];
        }

        return water;
    }
};
```

python

```
# Python Implementation
class PrefixMinMax:
    def __init__(self, arr):
        self.n = len(arr)
        self.prefix_min = [0] * self.n
        self.prefix_max = [0] * self.n
        self.build(arr)

    def build(self, arr):
        self.prefix_min[0] = self.prefix_max[0] = arr[0]

        for i in range(1, self.n):
            self.prefix_min[i] = min(self.prefix_min[i-1], arr[i])
            self.prefix_max[i] = max(self.prefix_max[i-1], arr[i])

    def max_profit_one_transaction(self, prices):
        if len(prices) < 2:
            return 0
```

```
        min_price = prices[0]
        max_profit = 0

        for price in prices[1:]:
            min_price = min(min_price, price)
            max_profit = max(max_profit, price - min_price)

        return max_profit

    def trap_water(self, height):
        n = len(height)
        if n == 0:
            return 0

        left_max = [0] * n
        right_max = [0] * n

        left_max[0] = height[0]
        for i in range(1, n):
            left_max[i] = max(left_max[i-1], height[i])

        right_max[n-1] = height[n-1]
        for i in range(n-2, -1, -1):
            right_max[i] = max(right_max[i+1], height[i])

        water = 0
        for i in range(n):
            water += min(left_max[i], right_max[i]) - height[i]

        return water

# Example
prices = [7, 1, 5, 3, 6, 4]
pmm = PrefixMinMax(prices)
print(f"Max profit one transaction: {pmm.max_profit_one_transaction(prices)}")  # 5

height = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]
print(f"Trapped water: {pmm.trap_water(height)}")  # 6
```

4. Prefix Frequency

Theory

Maintain frequency of elements up to each index. Useful for range queries about element counts.

Applications

- Range queries: "How many times does X appear in [l, r]?"
- Majority element in subarray
- Frequency-based constraints

Examples

Example: Range Frequency Query

```
text

Array: [1, 2, 1, 3, 1, 2, 4]
Query: Count of '1' in [2, 5]
Prefix freq of 1: [1, 1, 2, 2, 3, 3, 3]
Answer = pref[5] - pref[1] = 3 - 1 = 2
```

Code Implementation

```cpp
// C++ Implementation
class PrefixFrequency {
private:
    vector<unordered_map<int, int>> prefix_freq;

public:
    void build(const vector<int>& arr) {
        int n = arr.size();
        prefix_freq.resize(n);

        if (n == 0) return;

        // Initialize first element
        prefix_freq[0][arr[0]] = 1;

        // Build for rest
        for (int i = 1; i < n; i++) {
            prefix_freq[i] = prefix_freq[i-1];
            prefix_freq[i][arr[i]]++;
```

```
        }
    }

    int rangeFrequency(int l, int r, int target) {
        if (l == 0) {
            return prefix_freq[r][target];
        }

        int left_count = (prefix_freq[l-1].find(target) != prefix_freq[l-1].end())
                         ? prefix_freq[l-1][target] : 0;
        int right_count = (prefix_freq[r].find(target) != prefix_freq[r].end())
                          ? prefix_freq[r][target] : 0;

        return right_count - left_count;
    }

    // Alternative: Using vector of vectors for bounded values
    vector<vector<int>> buildFrequencyArray(const vector<int>& arr, int max_val) {
        int n = arr.size();
        vector<vector<int>> freq(n+1, vector<int>(max_val+1, 0));

        for (int i = 0; i < n; i++) {
            for (int val = 0; val <= max_val; val++) {
                freq[i+1][val] = freq[i][val];
            }
            freq[i+1][arr[i]]++;
        }

        return freq;
    }

    int rangeFrequencyWithArray(const vector<vector<int>>& freq,
                                int l, int r, int target, int max_val) {
        if (target < 0 || target > max_val) return 0;
        return freq[r+1][target] - freq[l][target];
    }
};
```

python

```
# Python Implementation
class PrefixFrequency:
    def __init__(self, arr, max_val=None):
        self.arr = arr
        self.n = len(arr)
        self.max_val = max_val
        self.prefix_freq = self.build_frequency_array()
```

```
    def build_frequency_array(self):
        if self.max_val is None:
            # Use dictionary approach
            prefix_freq = [{} for _ in range(self.n)]
            if self.n == 0:
                return prefix_freq

            prefix_freq[0][self.arr[0]] = 1

            for i in range(1, self.n):
                prefix_freq[i] = prefix_freq[i-1].copy()
                prefix_freq[i][self.arr[i]] = prefix_freq[i].get(self.arr[i], 0) +
1

            return prefix_freq
        else:
            # Use 2D array approach
            prefix_freq = [[0] * (self.max_val + 1) for _ in range(self.n + 1)]

            for i in range(self.n):
                for val in range(self.max_val + 1):
                    prefix_freq[i+1][val] = prefix_freq[i][val]
                prefix_freq[i+1][self.arr[i]] += 1

            return prefix_freq

    def range_frequency(self, l, r, target):
        if self.max_val is None:
            # Dictionary approach
            if l == 0:
                return self.prefix_freq[r].get(target, 0)
            left_count = self.prefix_freq[l-1].get(target, 0)
            right_count = self.prefix_freq[r].get(target, 0)
            return right_count - left_count
        else:
            # Array approach
            if target < 0 or target > self.max_val:
                return 0
            return self.prefix_freq[r+1][target] - self.prefix_freq[l][target]

# Example
arr = [1, 2, 1, 3, 1, 2, 4]
pf = PrefixFrequency(arr)
print(f"Frequency of 1 in [2, 5]: {pf.range_frequency(2, 5, 1)}")  # 2

# With bounded values
```

```
arr2 = [1, 2, 1, 3, 1, 2]
pf2 = PrefixFrequency(arr2, max_val=3)
print(f"Frequency of 2 in [1, 4]: {pf2.range_frequency(1, 4, 2)}")  # 1
```

5. Prefix Hash (for Strings)

Theory

Computes hash value for prefixes of strings. Uses polynomial rolling hash for substring comparison.

Hash Formula

```
text

hash(s[0..i]) = (s[0] * p⁰ + s[1] * p¹ + ... + s[i] * pⁱ) % mod
where p is prime (31, 37, etc.)
```

Applications

- String matching
- Palindrome checking
- Longest common substring

Code Implementation

```
cpp

// C++ Implementation
class PrefixHash {
private:
    vector<long long> prefix_hash;
    vector<long long> power;
    long long p = 31;
    long long mod = 1e9 + 7;
    int n;

public:
    void build(const string& s) {
        n = s.length();
        prefix_hash.resize(n + 1, 0);
```

```
        power.resize(n + 1, 1);

        for (int i = 0; i < n; i++) {
            power[i+1] = (power[i] * p) % mod;
            prefix_hash[i+1] = (prefix_hash[i] + (s[i] - 'a' + 1) * power[i]) % mo
d;
        }
    }

    long long getHash(int l, int r) {
        // Hash of substring s[l..r]
        long long hash_val = (prefix_hash[r+1] - prefix_hash[l] + mod) % mod;
        // Normalize by dividing by p^l
        hash_val = (hash_val * power[n-l]) % mod;
        return hash_val;
    }

    bool compareSubstrings(int l1, int r1, int l2, int r2) {
        if (r1 - l1 != r2 - l2) return false;
        return getHash(l1, r1) == getHash(l2, r2);
    }

    // Find longest palindrome using prefix and reverse prefix hash
    string longestPalindrome(const string& s) {
        string rev_s = s;
        reverse(rev_s.begin(), rev_s.end());

        PrefixHash forward, backward;
        forward.build(s);
        backward.build(rev_s);

        int n = s.length();
        int max_len = 0, start = 0;

        // Check odd length palindromes
        for (int center = 0; center < n; center++) {
            int low = 0, high = min(center, n-1-center);
            while (low <= high) {
                int mid = (low + high) / 2;
                int l = center - mid;
                int r = center + mid;

                long long forward_hash = forward.getHash(l, r);
                long long backward_hash = backward.getHash(n-1-r, n-1-l);

                if (forward_hash == backward_hash) {
                    if (2*mid + 1 > max_len) {
```

```
                    max_len = 2*mid + 1;
                    start = l;
                }
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
    }

    // Check even length palindromes
    for (int center = 0; center < n-1; center++) {
        int low = 0, high = min(center, n-2-center);
        while (low <= high) {
            int mid = (low + high) / 2;
            int l = center - mid;
            int r = center + 1 + mid;

            if (r >= n) break;

            long long forward_hash = forward.getHash(l, r);
            long long backward_hash = backward.getHash(n-1-r, n-1-l);

            if (forward_hash == backward_hash) {
                if (2*mid + 2 > max_len) {
                    max_len = 2*mid + 2;
                    start = l;
                }
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
    }

    return s.substr(start, max_len);
}
};
```

python

```
# Python Implementation
class PrefixHash:
    def __init__(self, s, p=31, mod=10**9+7):
        self.s = s
        self.n = len(s)
        self.p = p
```

```
        self.mod = mod
        self.prefix_hash = [0] * (self.n + 1)
        self.power = [1] * (self.n + 1)
        self.build()

    def build(self):
        for i in range(self.n):
            self.power[i+1] = (self.power[i] * self.p) % self.mod
            char_val = ord(self.s[i]) - ord('a') + 1
            self.prefix_hash[i+1] = (self.prefix_hash[i] + char_val * self.power
[i]) % self.mod

    def get_hash(self, l, r):
        # Hash of substring s[l..r]
        hash_val = (self.prefix_hash[r+1] - self.prefix_hash[l]) % self.mod
        # Normalize
        hash_val = (hash_val * self.power[self.n - l]) % self.mod
        return hash_val

    def compare_substrings(self, l1, r1, l2, r2):
        if r1 - l1 != r2 - l2:
            return False
        return self.get_hash(l1, r1) == self.get_hash(l2, r2)

    def longest_palindrome(self):
        rev_s = self.s[::-1]
        forward = PrefixHash(self.s)
        backward = PrefixHash(rev_s)

        n = self.n
        max_len = 0
        start = 0

        # Check odd length
        for center in range(n):
            low, high = 0, min(center, n-1-center)
            while low <= high:
                mid = (low + high) // 2
                l, r = center - mid, center + mid

                f_hash = forward.get_hash(l, r)
                b_hash = backward.get_hash(n-1-r, n-1-l)

                if f_hash == b_hash:
                    if 2*mid + 1 > max_len:
                        max_len = 2*mid + 1
                        start = l
```

```
                    low = mid + 1
                else:
                    high = mid - 1

        # Check even length
        for center in range(n-1):
            low, high = 0, min(center, n-2-center)
            while low <= high:
                mid = (low + high) // 2
                l, r = center - mid, center + 1 + mid

                if r >= n:
                    break

                f_hash = forward.get_hash(l, r)
                b_hash = backward.get_hash(n-1-r, n-1-l)

                if f_hash == b_hash:
                    if 2*mid + 2 > max_len:
                        max_len = 2*mid + 2
                        start = l
                    low = mid + 1
                else:
                    high = mid - 1

        return self.s[start:start+max_len]

# Example
s = "babad"
ph = PrefixHash(s)
print(f"Hash of 'aba' in 'babad': {ph.get_hash(1, 3)}")
print(f"Longest palindrome: {ph.longest_palindrome()}")  # "bab" or "aba"
```

6. Prefix GCD

Theory

Computes GCD of elements from start to each index. GCD properties:

- `gcd(a, b, c) = gcd(gcd(a, b), c)`
- `gcd(a, 0) = a`

Applications

- Range GCD queries
- Divide array into subarrays with same GCD
- Number theory problems

Examples

Example: Range GCD Query

```text
Array: [4, 8, 12, 16, 20]
Prefix GCD: [4, 4, 4, 4, 4]
GCD(1, 3) = gcd(pref[3], arr[1..3]) = 4
```

Code Implementation

```cpp
// C++ Implementation
#include <numeric>
#include <vector>
#include <iostream>

class PrefixGCD {
public:
    vector<int> prefix_gcd;

    void build(const vector<int>& arr) {
        int n = arr.size();
        prefix_gcd.resize(n);

        if (n == 0) return;

        prefix_gcd[0] = arr[0];
        for (int i = 1; i < n; i++) {
            prefix_gcd[i] = gcd(prefix_gcd[i-1], arr[i]);
        }
    }

    int getPrefixGCD(int idx) {
        return prefix_gcd[idx];
    }

    // Range GCD using segment tree concept
```

```cpp
int rangeGCD(const vector<int>& arr, int l, int r) {
    int n = arr.size();

    // Build prefix and suffix GCD arrays
    vector<int> prefix(n), suffix(n);

    prefix[0] = arr[0];
    for (int i = 1; i < n; i++) {
        prefix[i] = gcd(prefix[i-1], arr[i]);
    }

    suffix[n-1] = arr[n-1];
    for (int i = n-2; i >= 0; i--) {
        suffix[i] = gcd(suffix[i+1], arr[i]);
    }

    if (l == 0) {
        return (r == n-1) ? prefix[n-1] : suffix[r+1];
    }
    if (r == n-1) {
        return prefix[l-1];
    }

    return gcd(prefix[l-1], suffix[r+1]);
}

// Count subarrays with GCD = k
int countSubarraysWithGCDK(const vector<int>& arr, int k) {
    int n = arr.size();
    unordered_map<int, int> gcd_count;
    unordered_map<int, int> temp;
    int total = 0;

    for (int i = 0; i < n; i++) {
        temp[arr[i]]++;

        unordered_map<int, int> new_gcd;
        for (auto& [g, cnt] : gcd_count) {
            int new_g = gcd(g, arr[i]);
            new_gcd[new_g] += cnt;
        }

        for (auto& [g, cnt] : new_gcd) {
            temp[g] += cnt;
        }

        total += temp[k];
```

```
            gcd_count = temp;
            temp.clear();
        }

        return total;
    }
};
```

python

```python
# Python Implementation
import math
from collections import defaultdict

class PrefixGCD:
    def __init__(self, arr):
        self.arr = arr
        self.n = len(arr)
        self.prefix_gcd = self.build_prefix()

    def build_prefix(self):
        prefix = [0] * self.n
        if self.n == 0:
            return prefix

        prefix[0] = self.arr[0]
        for i in range(1, self.n):
            prefix[i] = math.gcd(prefix[i-1], self.arr[i])

        return prefix

    def get_prefix_gcd(self, idx):
        return self.prefix_gcd[idx]

    def range_gcd(self, l, r):
        # Build prefix and suffix GCD arrays
        prefix = [0] * self.n
        suffix = [0] * self.n

        prefix[0] = self.arr[0]
        for i in range(1, self.n):
            prefix[i] = math.gcd(prefix[i-1], self.arr[i])

        suffix[self.n-1] = self.arr[self.n-1]
        for i in range(self.n-2, -1, -1):
            suffix[i] = math.gcd(suffix[i+1], self.arr[i])
```

```
        if l == 0:
            return suffix[r+1] if r < self.n-1 else prefix[self.n-1]
        if r == self.n-1:
            return prefix[l-1]

        return math.gcd(prefix[l-1], suffix[r+1])

    def count_subarrays_with_gcd_k(self, k):
        gcd_count = defaultdict(int)
        result = 0

        for num in self.arr:
            new_gcd_count = defaultdict(int)
            new_gcd_count[num] += 1

            for g, cnt in gcd_count.items():
                new_g = math.gcd(g, num)
                new_gcd_count[new_g] += cnt

            result += new_gcd_count.get(k, 0)
            gcd_count = new_gcd_count

        return result

# Example
arr = [4, 8, 12, 16, 20]
pg = PrefixGCD(arr)
print(f"Prefix GCD at index 3: {pg.get_prefix_gcd(3)}")  # 4
print(f"Range GCD [1, 3]: {pg.range_gcd(1, 3)}")  # 4
print(f"Subarrays with GCD 4: {pg.count_subarrays_with_gcd_k(4)}")  # 10
```

7. Prefix AND/OR

Theory

Computes bitwise AND/OR of elements from start to each index.

Properties

- **AND**: Can only decrease or stay same (monotonic decreasing)
- **OR**: Can only increase or stay same (monotonic increasing)
- Useful for bitmask problems

Applications

- Subarray bitwise operations
- Bitmask DP
- Range queries with bit operations

Code Implementation

cpp

```cpp
// C++ Implementation
class PrefixBitwise {
public:
    vector<int> prefix_and;
    vector<int> prefix_or;

    void build(const vector<int>& arr) {
        int n = arr.size();
        prefix_and.resize(n);
        prefix_or.resize(n);

        if (n == 0) return;

        prefix_and[0] = arr[0];
        prefix_or[0] = arr[0];

        for (int i = 1; i < n; i++) {
            prefix_and[i] = prefix_and[i-1] & arr[i];
            prefix_or[i] = prefix_or[i-1] | arr[i];
        }
    }

    int rangeAND(int l, int r) {
        if (l == 0) return prefix_and[r];

        // For AND, we need different approach
        // Using sparse table or segment tree is better
        int result = arr[l];
        for (int i = l+1; i <= r; i++) {
            result &= arr[i];
        }
        return result;
    }

    int rangeOR(int l, int r) {
```

```
        if (l == 0) return prefix_or[r];

        int result = arr[l];
        for (int i = l+1; i <= r; i++) {
            result |= arr[i];
        }
        return result;
    }

    // Count subarrays with AND >= K
    int countSubarraysWithANDGreaterThanK(const vector<int>& arr, int K) {
        int n = arr.size();
        int count = 0;

        for (int i = 0; i < n; i++) {
            int current_and = arr[i];
            if (current_and >= K) count++;

            for (int j = i+1; j < n; j++) {
                current_and &= arr[j];
                if (current_and >= K) {
                    count++;
                } else {
                    break; // AND only decreases
                }
            }
        }

        return count;
    }

    // Maximum subarray OR
    int maxSubarrayOR(const vector<int>& arr) {
        int n = arr.size();
        unordered_set<int> current, next, all_values;

        for (int num : arr) {
            next.clear();
            next.insert(num);

            for (int val : current) {
                next.insert(val | num);
            }

            for (int val : next) {
                all_values.insert(val);
            }
```

```
            current = next;
        }

        return *max_element(all_values.begin(), all_values.end());
    }
};
```

python

```python
# Python Implementation
class PrefixBitwise:
    def __init__(self, arr):
        self.arr = arr
        self.n = len(arr)
        self.prefix_and, self.prefix_or = self.build()

    def build(self):
        if self.n == 0:
            return [], []

        prefix_and = [0] * self.n
        prefix_or = [0] * self.n

        prefix_and[0] = self.arr[0]
        prefix_or[0] = self.arr[0]

        for i in range(1, self.n):
            prefix_and[i] = prefix_and[i-1] & self.arr[i]
            prefix_or[i] = prefix_or[i-1] | self.arr[i]

        return prefix_and, prefix_or

    def range_and(self, l, r):
        if l == 0:
            return self.prefix_and[r]
        # Simple approach for small ranges
        result = self.arr[l]
        for i in range(l+1, r+1):
            result &= self.arr[i]
        return result

    def range_or(self, l, r):
        if l == 0:
            return self.prefix_or[r]
        result = self.arr[l]
        for i in range(l+1, r+1):
```

```
            result |= self.arr[i]
        return result

    def count_subarrays_with_and_greater_than_k(self, K):
        count = 0
        n = self.n

        for i in range(n):
            current_and = self.arr[i]
            if current_and >= K:
                count += 1

            for j in range(i+1, n):
                current_and &= self.arr[j]
                if current_and >= K:
                    count += 1
                else:
                    break  # AND only decreases

        return count

    def max_subarray_or(self):
        current = set()
        all_values = set()

        for num in self.arr:
            next_set = {num}
            for val in current:
                next_set.add(val | num)

            all_values.update(next_set)
            current = next_set

        return max(all_values)

# Example
arr = [1, 2, 3, 4, 5]
pb = PrefixBitwise(arr)
print(f"Prefix AND: {pb.prefix_and}")  # [1, 0, 0, 0, 0]
print(f"Prefix OR: {pb.prefix_or}")    # [1, 3, 3, 7, 7]
print(f"Range OR [1, 3]: {pb.range_or(1, 3)}")  # 2|3|4 = 7
print(f"Max subarray OR: {pb.max_subarray_or()}")  # 7
```

8. Prefix Common Count

Theory

Counts number of common elements between two sequences up to each index.

Applications

- String similarity
- Sequence alignment
- Edit distance variants

Examples

Example: Longest Common Prefix

```text

String 1: "abcde"
String 2: "abfde"
Common prefix count: [1, 2, 2, 2, 3]
```

Code Implementation

```cpp

// C++ Implementation
class PrefixCommon {
public:
    // Longest Common Prefix between two strings
    vector<int> buildLCP(const string& s1, const string& s2) {
        int n = min(s1.length(), s2.length());
        vector<int> lcp(n, 0);

        if (n == 0) return lcp;

        lcp[0] = (s1[0] == s2[0]) ? 1 : 0;
        for (int i = 1; i < n; i++) {
            if (s1[i] == s2[i]) {
                lcp[i] = lcp[i-1] + 1;
            } else {
                lcp[i] = 0;
            }
        }
```

```
        return lcp;
    }

    // Count of common elements in two arrays up to each index
    vector<int> countCommonElements(const vector<int>& arr1, const vector<int>& arr
2) {
        int n = min(arr1.size(), arr2.size());
        vector<int> common_count(n, 0);

        if (n == 0) return common_count;

        unordered_map<int, int> freq1, freq2;

        for (int i = 0; i < n; i++) {
            freq1[arr1[i]]++;
            freq2[arr2[i]]++;

            // Count common elements
            int count = 0;
            for (auto& [num, cnt1] : freq1) {
                if (freq2.find(num) != freq2.end()) {
                    count += min(cnt1, freq2[num]);
                }
            }
            common_count[i] = count;
        }

        return common_count;
    }

    // Edit distance with prefix common optimization
    int editDistanceWithPrefix(const string& word1, const string& word2) {
        int m = word1.length(), n = word2.length();
        vector<vector<int>> dp(m + 1, vector<int>(n + 1));

        // Build LCP array for optimization
        vector<int> lcp = buildLCP(word1, word2);

        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
                if (i == 0) {
                    dp[i][j] = j;
                } else if (j == 0) {
                    dp[i][j] = i;
                } else if (word1[i-1] == word2[j-1]) {
                    dp[i][j] = dp[i-1][j-1];
                } else {
```

```
                    dp[i][j] = 1 + min({dp[i-1][j], dp[i][j-1], dp[i-1][j-1]});
                }
            }
        }

        return dp[m][n];
    }
};
```

python

```python
# Python Implementation
class PrefixCommon:
    def build_lcp(self, s1, s2):
        n = min(len(s1), len(s2))
        lcp = [0] * n

        if n == 0:
            return lcp

        lcp[0] = 1 if s1[0] == s2[0] else 0
        for i in range(1, n):
            if s1[i] == s2[i]:
                lcp[i] = lcp[i-1] + 1
            else:
                lcp[i] = 0

        return lcp

    def count_common_elements(self, arr1, arr2):
        n = min(len(arr1), len(arr2))
        common_count = [0] * n

        if n == 0:
            return common_count

        from collections import defaultdict
        freq1 = defaultdict(int)
        freq2 = defaultdict(int)

        for i in range(n):
            freq1[arr1[i]] += 1
            freq2[arr2[i]] += 1

            # Count common elements
            count = 0
            for num, cnt1 in freq1.items():
```

```
                if num in freq2:
                    count += min(cnt1, freq2[num])
            common_count[i] = count

        return common_count

    def edit_distance_with_prefix(self, word1, word2):
        m, n = len(word1), len(word2)
        dp = [[0] * (n + 1) for _ in range(m + 1)]

        # Build LCP array for optimization
        lcp = self.build_lcp(word1, word2)

        for i in range(m + 1):
            for j in range(n + 1):
                if i == 0:
                    dp[i][j] = j
                elif j == 0:
                    dp[i][j] = i
                elif word1[i-1] == word2[j-1]:
                    dp[i][j] = dp[i-1][j-1]
                else:
                    dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])

        return dp[m][n]

# Example
s1 = "abcde"
s2 = "abfde"
pc = PrefixCommon()
lcp = pc.build_lcp(s1, s2)
print(f"LCP array: {lcp}")  # [1, 2, 0, 0, 1]

arr1 = [1, 2, 3, 4, 5]
arr2 = [2, 3, 4, 4, 5]
common = pc.count_common_elements(arr1, arr2)
print(f"Common elements count: {common}")  # [0, 1, 2, 3, 4]
```

9. Suffix Arrays (All Types)

Theory

Suffix arrays store information from the end to each position. Useful for problems where we need to consider suffix/subarray from i to end.

Applications

- Maximum subarray from each position
- Suffix arrays in string algorithms
- Problems considering "remaining array"

Code Implementation

cpp

```cpp
// C++ Implementation - Comprehensive Suffix Arrays
class SuffixArrays {
public:
    // Suffix Sum
    vector<int> buildSuffixSum(const vector<int>& arr) {
        int n = arr.size();
        vector<int> suffix_sum(n);

        if (n == 0) return suffix_sum;

        suffix_sum[n-1] = arr[n-1];
        for (int i = n-2; i >= 0; i--) {
            suffix_sum[i] = suffix_sum[i+1] + arr[i];
        }

        return suffix_sum;
    }

    // Suffix Max/Min
    pair<vector<int>, vector<int>> buildSuffixMinMax(const vector<int>& arr) {
        int n = arr.size();
        vector<int> suffix_min(n), suffix_max(n);

        if (n == 0) return {suffix_min, suffix_max};

        suffix_min[n-1] = suffix_max[n-1] = arr[n-1];

        for (int i = n-2; i >= 0; i--) {
            suffix_min[i] = min(suffix_min[i+1], arr[i]);
            suffix_max[i] = max(suffix_max[i+1], arr[i]);
        }

        return {suffix_min, suffix_max};
    }
```

```cpp
// Suffix GCD
vector<int> buildSuffixGCD(const vector<int>& arr) {
    int n = arr.size();
    vector<int> suffix_gcd(n);

    if (n == 0) return suffix_gcd;

    suffix_gcd[n-1] = arr[n-1];
    for (int i = n-2; i >= 0; i--) {
        suffix_gcd[i] = gcd(suffix_gcd[i+1], arr[i]);
    }

    return suffix_gcd;
}

// Suffix AND/OR
pair<vector<int>, vector<int>> buildSuffixAndOr(const vector<int>& arr) {
    int n = arr.size();
    vector<int> suffix_and(n), suffix_or(n);

    if (n == 0) return {suffix_and, suffix_or};

    suffix_and[n-1] = suffix_or[n-1] = arr[n-1];

    for (int i = n-2; i >= 0; i--) {
        suffix_and[i] = suffix_and[i+1] & arr[i];
        suffix_or[i] = suffix_or[i+1] | arr[i];
    }

    return {suffix_and, suffix_or};
}

// Example: Maximum subarray sum starting at each position
vector<int> maxSubarraySumFromEachPosition(const vector<int>& arr) {
    int n = arr.size();
    vector<int> suffix_max_sum(n);

    if (n == 0) return suffix_max_sum;

    suffix_max_sum[n-1] = arr[n-1];
    int current_max = arr[n-1];

    for (int i = n-2; i >= 0; i--) {
        current_max = max(arr[i], current_max + arr[i]);
        suffix_max_sum[i] = max(suffix_max_sum[i+1], current_max);
    }
```

```
        return suffix_max_sum;
    }

    // Example: Next greater element (using suffix concept)
    vector<int> nextGreaterElements(const vector<int>& arr) {
        int n = arr.size();
        vector<int> result(n, -1);
        stack<int> st;

        for (int i = n-1; i >= 0; i--) {
            while (!st.empty() && arr[st.top()] <= arr[i]) {
                st.pop();
            }
            if (!st.empty()) {
                result[i] = st.top();
            }
            st.push(i);
        }

        return result;
    }
};
```

python

```
# Python Implementation
import math

class SuffixArrays:
    def build_suffix_sum(self, arr):
        n = len(arr)
        suffix_sum = [0] * n

        if n == 0:
            return suffix_sum

        suffix_sum[-1] = arr[-1]
        for i in range(n-2, -1, -1):
            suffix_sum[i] = suffix_sum[i+1] + arr[i]

        return suffix_sum

    def build_suffix_min_max(self, arr):
        n = len(arr)
        suffix_min = [0] * n
        suffix_max = [0] * n
```

```
        if n == 0:
            return suffix_min, suffix_max

        suffix_min[-1] = suffix_max[-1] = arr[-1]

        for i in range(n-2, -1, -1):
            suffix_min[i] = min(suffix_min[i+1], arr[i])
            suffix_max[i] = max(suffix_max[i+1], arr[i])

        return suffix_min, suffix_max

    def build_suffix_gcd(self, arr):
        n = len(arr)
        suffix_gcd = [0] * n

        if n == 0:
            return suffix_gcd

        suffix_gcd[-1] = arr[-1]
        for i in range(n-2, -1, -1):
            suffix_gcd[i] = math.gcd(suffix_gcd[i+1], arr[i])

        return suffix_gcd

    def build_suffix_and_or(self, arr):
        n = len(arr)
        suffix_and = [0] * n
        suffix_or = [0] * n

        if n == 0:
            return suffix_and, suffix_or

        suffix_and[-1] = suffix_or[-1] = arr[-1]

        for i in range(n-2, -1, -1):
            suffix_and[i] = suffix_and[i+1] & arr[i]
            suffix_or[i] = suffix_or[i+1] | arr[i]

        return suffix_and, suffix_or

    def max_subarray_sum_from_each_position(self, arr):
        n = len(arr)
        suffix_max_sum = [0] * n

        if n == 0:
            return suffix_max_sum
```

```
        suffix_max_sum[-1] = arr[-1]
        current_max = arr[-1]

        for i in range(n-2, -1, -1):
            current_max = max(arr[i], current_max + arr[i])
            suffix_max_sum[i] = max(suffix_max_sum[i+1], current_max)

        return suffix_max_sum

    def next_greater_elements(self, arr):
        n = len(arr)
        result = [-1] * n
        stack = []

        for i in range(n-1, -1, -1):
            while stack and arr[stack[-1]] <= arr[i]:
                stack.pop()
            if stack:
                result[i] = stack[-1]
            stack.append(i)

        return result

# Example
arr = [3, 1, 4, 2, 5]
sa = SuffixArrays()

suffix_sum = sa.build_suffix_sum(arr)
print(f"Suffix sum: {suffix_sum}")  # [15, 12, 11, 7, 5]

suffix_min, suffix_max = sa.build_suffix_min_max(arr)
print(f"Suffix min: {suffix_min}")  # [1, 1, 2, 2, 5]
print(f"Suffix max: {suffix_max}")  # [5, 5, 5, 5, 5]

max_sums = sa.max_subarray_sum_from_each_position(arr)
print(f"Max subarray sum from each position: {max_sums}")  # [15, 12, 11, 7, 5]
```

10. Prefix + Suffix Arrays Technique

Theory

Combining prefix and suffix arrays allows solving problems where we remove or split at each position.

Applications

- Remove one element problems
- Split array problems
- Maximum sum with one element removal

Code Implementation

cpp

```cpp
// C++ Implementation
class PrefixSuffixCombination {
public:
    // Maximum sum subarray with at most one deletion
    int maximumSumWithOneDeletion(const vector<int>& arr) {
        int n = arr.size();
        if (n == 0) return 0;

        vector<int> prefix_max(n), suffix_max(n);

        // Kadane's from left
        prefix_max[0] = arr[0];
        int current_max = arr[0];
        for (int i = 1; i < n; i++) {
            current_max = max(arr[i], current_max + arr[i]);
            prefix_max[i] = max(prefix_max[i-1], current_max);
        }

        // Kadane's from right
        suffix_max[n-1] = arr[n-1];
        current_max = arr[n-1];
        for (int i = n-2; i >= 0; i--) {
            current_max = max(arr[i], current_max + arr[i]);
            suffix_max[i] = max(suffix_max[i+1], current_max);
        }

        // Try removing each element
        int max_sum = prefix_max[n-1];  // no deletion

        for (int i = 0; i < n; i++) {
            int left = (i > 0) ? prefix_max[i-1] : 0;
            int right = (i < n-1) ? suffix_max[i+1] : 0;
            max_sum = max(max_sum, left + right);
        }
```

```
        return max_sum;
    }

    // Product of array except self
    vector<int> productExceptSelf(const vector<int>& nums) {
        int n = nums.size();
        vector<int> result(n, 1);

        // Prefix product
        int prefix = 1;
        for (int i = 0; i < n; i++) {
            result[i] = prefix;
            prefix *= nums[i];
        }

        // Suffix product
        int suffix = 1;
        for (int i = n-1; i >= 0; i--) {
            result[i] *= suffix;
            suffix *= nums[i];
        }

        return result;
    }

    // Maximum sum of two non-overlapping subarrays
    int maxSumTwoNoOverlap(const vector<int>& nums, int firstLen, int secondLen) {
        int n = nums.size();

        // Prefix sum
        vector<int> prefix_sum(n+1, 0);
        for (int i = 0; i < n; i++) {
            prefix_sum[i+1] = prefix_sum[i] + nums[i];
        }

        // DP arrays
        vector<int> left_first(n+1, 0), left_second(n+1, 0);
        vector<int> right_first(n+1, 0), right_second(n+1, 0);

        // Left to right
        for (int i = firstLen; i <= n; i++) {
            left_first[i] = max(left_first[i-1],
                                prefix_sum[i] - prefix_sum[i-firstLen]);
        }
        for (int i = secondLen; i <= n; i++) {
            left_second[i] = max(left_second[i-1],
                                 prefix_sum[i] - prefix_sum[i-secondLen]);
```

```
        }

        // Right to left
        for (int i = n - firstLen; i >= 0; i--) {
            right_first[i] = max(right_first[i+1],
                                 prefix_sum[i+firstLen] - prefix_sum[i]);
        }
        for (int i = n - secondLen; i >= 0; i--) {
            right_second[i] = max(right_second[i+1],
                                  prefix_sum[i+secondLen] - prefix_sum[i]);
        }

        // Try all splits
        int max_sum = 0;

        // First then second
        for (int i = firstLen; i <= n - secondLen; i++) {
            max_sum = max(max_sum, left_first[i] + right_second[i]);
        }

        // Second then first
        for (int i = secondLen; i <= n - firstLen; i++) {
            max_sum = max(max_sum, left_second[i] + right_first[i]);
        }

        return max_sum;
    }
};
```

python

```
# Python Implementation
class PrefixSuffixCombination:
    def maximum_sum_with_one_deletion(self, arr):
        n = len(arr)
        if n == 0:
            return 0

        prefix_max = [0] * n
        suffix_max = [0] * n

        # Kadane's from left
        prefix_max[0] = arr[0]
        current_max = arr[0]
        for i in range(1, n):
            current_max = max(arr[i], current_max + arr[i])
            prefix_max[i] = max(prefix_max[i-1], current_max)
```

```
        # Kadane's from right
        suffix_max[-1] = arr[-1]
        current_max = arr[-1]
        for i in range(n-2, -1, -1):
            current_max = max(arr[i], current_max + arr[i])
            suffix_max[i] = max(suffix_max[i+1], current_max)

        # Try removing each element
        max_sum = prefix_max[-1]  # no deletion

        for i in range(n):
            left = prefix_max[i-1] if i > 0 else 0
            right = suffix_max[i+1] if i < n-1 else 0
            max_sum = max(max_sum, left + right)

        return max_sum

    def product_except_self(self, nums):
        n = len(nums)
        result = [1] * n

        # Prefix product
        prefix = 1
        for i in range(n):
            result[i] = prefix
            prefix *= nums[i]

        # Suffix product
        suffix = 1
        for i in range(n-1, -1, -1):
            result[i] *= suffix
            suffix *= nums[i]

        return result

    def max_sum_two_no_overlap(self, nums, first_len, second_len):
        n = len(nums)

        # Prefix sum
        prefix_sum = [0] * (n + 1)
        for i in range(n):
            prefix_sum[i+1] = prefix_sum[i] + nums[i]

        # DP arrays
        left_first = [0] * (n + 1)
        left_second = [0] * (n + 1)
```

```
        right_first = [0] * (n + 1)
        right_second = [0] * (n + 1)

        # Left to right
        for i in range(first_len, n + 1):
            left_first[i] = max(left_first[i-1],
                                prefix_sum[i] - prefix_sum[i-first_len])
        for i in range(second_len, n + 1):
            left_second[i] = max(left_second[i-1],
                                 prefix_sum[i] - prefix_sum[i-second_len])

        # Right to left
        for i in range(n - first_len, -1, -1):
            right_first[i] = max(right_first[i+1],
                                 prefix_sum[i+first_len] - prefix_sum[i])
        for i in range(n - second_len, -1, -1):
            right_second[i] = max(right_second[i+1],
                                  prefix_sum[i+second_len] - prefix_sum[i])

        # Try all splits
        max_sum = 0

        # First then second
        for i in range(first_len, n - second_len + 1):
            max_sum = max(max_sum, left_first[i] + right_second[i])

        # Second then first
        for i in range(second_len, n - first_len + 1):
            max_sum = max(max_sum, left_second[i] + right_first[i])

        return max_sum

# Example
arr = [1, -2, 0, 3]
psc = PrefixSuffixCombination()
print(f"Max sum with one deletion: {psc.maximum_sum_with_one_deletion(arr)}")  # 4

nums = [1, 2, 3, 4]
print(f"Product except self: {psc.product_except_self(nums)}")  # [24, 12, 8, 6]

arr2 = [0, 6, 5, 2, 2, 5, 1, 9, 4]
print(f"Max sum two non-overlap: {psc.max_sum_two_no_overlap(arr2, 1, 2)}")  # 20
```

11. Difference Array & Imos Technique

Theory

Difference Array: Technique for range updates in O(1).

Imos Technique: 2D version of difference array.

Applications

- Range add operations
- Calendar/interval problems
- Overlap counting

Examples

Example: Range Add

```text

Array: [0, 0, 0, 0, 0]
Update: Add 5 to [1, 3]
Diff: [0, 5, 0, 0, -5]
After prefix sum: [0, 5, 5, 5, 0]
```

Code Implementation

```cpp

// C++ Implementation
class DifferenceArray {
public:
    // 1D Difference array
    vector<int> rangeAdd(const vector<int>& arr,
                         const vector<vector<int>>& updates) {
        int n = arr.size();
        vector<int> diff(n + 1, 0);

        // Apply updates
        for (const auto& update : updates) {
            int l = update[0], r = update[1], val = update[2];
            diff[l] += val;
            if (r + 1 < n) {
                diff[r + 1] -= val;
            }
        }
```

```cpp
    // Reconstruct array
    vector<int> result = arr;
    int current = 0;
    for (int i = 0; i < n; i++) {
        current += diff[i];
        result[i] += current;
    }

    return result;
}

// 2D Difference array (Imos technique)
vector<vector<int>> imos2D(int rows, int cols,
                           const vector<vector<int>>& updates) {
    // updates: {x1, y1, x2, y2, val}
    vector<vector<int>> diff(rows + 2, vector<int>(cols + 2, 0));

    // Apply updates
    for (const auto& update : updates) {
        int x1 = update[0], y1 = update[1];
        int x2 = update[2], y2 = update[3];
        int val = update[4];

        diff[x1][y1] += val;
        diff[x1][y2 + 1] -= val;
        diff[x2 + 1][y1] -= val;
        diff[x2 + 1][y2 + 1] += val;
    }

    // Reconstruct
    vector<vector<int>> result(rows, vector<int>(cols, 0));

    // First pass: horizontal
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (j > 0) diff[i][j] += diff[i][j - 1];
        }
    }

    // Second pass: vertical
    for (int j = 0; j < cols; j++) {
        for (int i = 0; i < rows; i++) {
            if (i > 0) diff[i][j] += diff[i - 1][j];
            result[i][j] = diff[i][j];
        }
    }
```

```
    return result;
}

// Car pooling problem
bool carPooling(vector<vector<int>>& trips, int capacity) {
    vector<int> diff(1001, 0);

    for (const auto& trip : trips) {
        int passengers = trip[0];
        int start = trip[1];
        int end = trip[2];

        diff[start] += passengers;
        diff[end] -= passengers;
    }

    int current = 0;
    for (int i = 0; i <= 1000; i++) {
        current += diff[i];
        if (current > capacity) {
            return false;
        }
    }

    return true;
}

// Maximum population year
int maximumPopulation(vector<vector<int>>& logs) {
    vector<int> diff(2051, 0);

    for (const auto& log : logs) {
        int birth = log[0];
        int death = log[1];

        diff[birth]++;
        diff[death]--;
    }

    int max_pop = 0;
    int current = 0;
    int max_year = 0;

    for (int year = 1950; year <= 2050; year++) {
        current += diff[year];
        if (current > max_pop) {
```

```
                max_pop = current;
                max_year = year;
            }
        }

        return max_year;
    }
};
```

python

```python
# Python Implementation
class DifferenceArray:
    def range_add(self, arr, updates):
        """
        updates: list of [l, r, val]
        """
        n = len(arr)
        diff = [0] * (n + 1)

        # Apply updates
        for l, r, val in updates:
            diff[l] += val
            if r + 1 < n:
                diff[r + 1] -= val

        # Reconstruct array
        result = arr[:]
        current = 0
        for i in range(n):
            current += diff[i]
            result[i] += current

        return result

    def imos_2d(self, rows, cols, updates):
        """
        updates: list of [x1, y1, x2, y2, val]
        """
        # Add extra row and column for boundaries
        diff = [[0] * (cols + 2) for _ in range(rows + 2)]

        # Apply updates
        for x1, y1, x2, y2, val in updates:
            diff[x1][y1] += val
            diff[x1][y2 + 1] -= val
            diff[x2 + 1][y1] -= val
```

```
            diff[x2 + 1][y2 + 1] += val

        # Reconstruct
        result = [[0] * cols for _ in range(rows)]

        # First pass: horizontal
        for i in range(rows):
            for j in range(cols):
                if j > 0:
                    diff[i][j] += diff[i][j - 1]

        # Second pass: vertical
        for j in range(cols):
            for i in range(rows):
                if i > 0:
                    diff[i][j] += diff[i - 1][j]
                result[i][j] = diff[i][j]

        return result

    def car_pooling(self, trips, capacity):
        diff = [0] * 1001

        for passengers, start, end in trips:
            diff[start] += passengers
            diff[end] -= passengers

        current = 0
        for i in range(1001):
            current += diff[i]
            if current > capacity:
                return False

        return True

    def maximum_population(self, logs):
        diff = [0] * 2051

        for birth, death in logs:
            diff[birth] += 1
            diff[death] -= 1

        max_pop = 0
        current = 0
        max_year = 0

        for year in range(1950, 2051):
```

```
            current += diff[year]
            if current > max_pop:
                max_pop = current
                max_year = year

        return max_year

# Example
da = DifferenceArray()
arr = [0, 0, 0, 0, 0]
updates = [[1, 3, 5], [2, 4, 3]]
print(f"After range add: {da.range_add(arr, updates)}")  # [0, 5, 8, 8, 3]

trips = [[2, 1, 5], [3, 3, 7]]
print(f"Car pooling possible with capacity 4: {da.car_pooling(trips, 4)}")  # True

logs = [[1993, 1999], [2000, 2010]]
print(f"Maximum population year: {da.maximum_population(logs)}")  # 1993
```

12. Sliding Window & Two Pointers

Theory

Sliding Window: Maintain a window that satisfies certain constraints.
Two Pointers: Use two pointers to traverse array, often from both ends.

Applications

- Subarray/substring with constraints
- Pair sum problems
- Remove duplicates

Code Implementation

cpp

```
// C++ Implementation
class SlidingWindowTwoPointers {
public:
    // Maximum sum subarray of size k
    int maxSumSubarrayOfSizeK(const vector<int>& arr, int k) {
        int n = arr.size();
```

```
    if (n < k) return -1;

    int window_sum = 0;
    for (int i = 0; i < k; i++) {
        window_sum += arr[i];
    }

    int max_sum = window_sum;
    for (int i = k; i < n; i++) {
        window_sum = window_sum + arr[i] - arr[i - k];
        max_sum = max(max_sum, window_sum);
    }

    return max_sum;
}

// Longest substring without repeating characters
int longestUniqueSubstring(const string& s) {
    int n = s.length();
    unordered_map<char, int> last_seen;
    int max_len = 0;
    int start = 0;

    for (int end = 0; end < n; end++) {
        char c = s[end];

        if (last_seen.find(c) != last_seen.end() && last_seen[c] >= start) {
            start = last_seen[c] + 1;
        }

        last_seen[c] = end;
        max_len = max(max_len, end - start + 1);
    }

    return max_len;
}

// Two pointers: Container with most water
int maxArea(vector<int>& height) {
    int n = height.size();
    int left = 0, right = n - 1;
    int max_water = 0;

    while (left < right) {
        int current_water = (right - left) * min(height[left], height[right]);
        max_water = max(max_water, current_water);
```

```
            if (height[left] < height[right]) {
                left++;
            } else {
                right--;
            }
        }

        return max_water;
    }

    // Two pointers: Three sum
    vector<vector<int>> threeSum(vector<int>& nums) {
        vector<vector<int>> result;
        int n = nums.size();
        sort(nums.begin(), nums.end());

        for (int i = 0; i < n - 2; i++) {
            if (i > 0 && nums[i] == nums[i-1]) continue;

            int left = i + 1, right = n - 1;
            while (left < right) {
                int sum = nums[i] + nums[left] + nums[right];

                if (sum == 0) {
                    result.push_back({nums[i], nums[left], nums[right]});

                    // Skip duplicates
                    while (left < right && nums[left] == nums[left+1]) left++;
                    while (left < right && nums[right] == nums[right-1]) right--;

                    left++;
                    right--;
                } else if (sum < 0) {
                    left++;
                } else {
                    right--;
                }
            }
        }

        return result;
    }

    // Variable size sliding window: Minimum window substring
    string minWindow(string s, string t) {
        unordered_map<char, int> target, window;
```

```
        for (char c : t) {
            target[c]++;
        }

        int left = 0, right = 0;
        int formed = 0;
        int required = target.size();

        // Result: [length, left, right]
        vector<int> result = {-1, 0, 0};

        while (right < s.length()) {
            char c = s[right];
            window[c]++;

            if (target.find(c) != target.end() && window[c] == target[c]) {
                formed++;
            }

            while (left <= right && formed == required) {
                c = s[left];

                // Update result
                if (result[0] == -1 || right - left + 1 < result[0]) {
                    result[0] = right - left + 1;
                    result[1] = left;
                    result[2] = right;
                }

                window[c]--;
                if (target.find(c) != target.end() && window[c] < target[c]) {
                    formed--;
                }

                left++;
            }

            right++;
        }

        return result[0] == -1 ? "" : s.substr(result[1], result[0]);
    }
};
```

python

```python
# Python Implementation
class SlidingWindowTwoPointers:
    def max_sum_subarray_of_size_k(self, arr, k):
        n = len(arr)
        if n < k:
            return -1

        window_sum = sum(arr[:k])
        max_sum = window_sum

        for i in range(k, n):
            window_sum = window_sum + arr[i] - arr[i - k]
            max_sum = max(max_sum, window_sum)

        return max_sum

    def longest_unique_substring(self, s):
        last_seen = {}
        max_len = 0
        start = 0

        for end, char in enumerate(s):
            if char in last_seen and last_seen[char] >= start:
                start = last_seen[char] + 1

            last_seen[char] = end
            max_len = max(max_len, end - start + 1)

        return max_len

    def max_area(self, height):
        left, right = 0, len(height) - 1
        max_water = 0

        while left < right:
            current_water = (right - left) * min(height[left], height[right])
            max_water = max(max_water, current_water)

            if height[left] < height[right]:
                left += 1
            else:
                right -= 1

        return max_water

    def three_sum(self, nums):
```

```
        nums.sort()
        n = len(nums)
        result = []

        for i in range(n - 2):
            if i > 0 and nums[i] == nums[i-1]:
                continue

            left, right = i + 1, n - 1
            while left < right:
                total = nums[i] + nums[left] + nums[right]

                if total == 0:
                    result.append([nums[i], nums[left], nums[right]])

                    # Skip duplicates
                    while left < right and nums[left] == nums[left+1]:
                        left += 1
                    while left < right and nums[right] == nums[right-1]:
                        right -= 1

                    left += 1
                    right -= 1
                elif total < 0:
                    left += 1
                else:
                    right -= 1

        return result

    def min_window(self, s, t):
        from collections import Counter

        target = Counter(t)
        window = Counter()

        left, right = 0, 0
        formed = 0
        required = len(target)

        result = (float('inf'), 0, 0)  # (length, left, right)

        while right < len(s):
            char = s[right]
            window[char] += 1

            if char in target and window[char] == target[char]:
```

```
                formed += 1

            while left <= right and formed == required:
                char = s[left]

                # Update result
                if right - left + 1 < result[0]:
                    result = (right - left + 1, left, right)

                window[char] -= 1
                if char in target and window[char] < target[char]:
                    formed -= 1

                left += 1

            right += 1

        return "" if result[0] == float('inf') else s[result[1]:result[2]+1]

# Example
sw = SlidingWindowTwoPointers()
arr = [1, 4, 2, 10, 23, 3, 1, 0, 20]
print(f"Max sum of size 4: {sw.max_sum_subarray_of_size_k(arr, 4)}")  # 39

s = "abcabcbb"
print(f"Longest unique substring: {sw.longest_unique_substring(s)}")  # 3

height = [1, 8, 6, 2, 5, 4, 8, 3, 7]
print(f"Max water container: {sw.max_area(height)}")  # 49

nums = [-1, 0, 1, 2, -1, -4]
print(f"Three sum: {sw.three_sum(nums)}")  # [[-1, -1, 2], [-1, 0, 1]]
```

13. Meet in the Middle

Theory

Split problem into two halves, solve each independently, then combine results. Reduces exponential complexity.

Applications

- Subset sum for large n

- 4Sum problem
- Divide and conquer optimization

Code Implementation

cpp

```cpp
// C++ Implementation
class MeetInMiddle {
public:
    // Subset sum for n up to 40
    long long countSubsetsWithSum(const vector<int>& arr, int target) {
        int n = arr.size();
        int n1 = n / 2;
        int n2 = n - n1;

        // Generate all subsets for first half
        vector<long long> sums1;
        for (int mask = 0; mask < (1 << n1); mask++) {
            long long sum = 0;
            for (int i = 0; i < n1; i++) {
                if (mask & (1 << i)) {
                    sum += arr[i];
                }
            }
            sums1.push_back(sum);
        }

        // Generate all subsets for second half
        vector<long long> sums2;
        for (int mask = 0; mask < (1 << n2); mask++) {
            long long sum = 0;
            for (int i = 0; i < n2; i++) {
                if (mask & (1 << i)) {
                    sum += arr[n1 + i];
                }
            }
            sums2.push_back(sum);
        }

        // Sort second half
        sort(sums2.begin(), sums2.end());

        // Count pairs that sum to target
        long long count = 0;
        for (long long sum1 : sums1) {
```

```
            long long needed = target - sum1;
            auto range = equal_range(sums2.begin(), sums2.end(), needed);
            count += (range.second - range.first);
        }

        return count;
    }

    // 4Sum using meet in the middle
    vector<vector<int>> fourSum(vector<int>& nums, int target) {
        int n = nums.size();
        sort(nums.begin(), nums.end());

        unordered_map<int, vector<pair<int, int>>> pair_sums;

        // Store all pair sums
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                int sum = nums[i] + nums[j];
                pair_sums[sum].push_back({i, j});
            }
        }

        set<vector<int>> result_set;

        // Find complementary pairs
        for (auto& [sum, pairs] : pair_sums) {
            int needed = target - sum;

            if (pair_sums.find(needed) != pair_sums.end()) {
                for (auto& p1 : pairs) {
                    for (auto& p2 : pair_sums[needed]) {
                        // Check if all indices are distinct
                        if (p1.first != p2.first && p1.first != p2.second &&
                            p1.second != p2.first && p1.second != p2.second) {
                            vector<int> quadruplet = {
                                nums[p1.first], nums[p1.second],
                                nums[p2.first], nums[p2.second]
                            };
                            sort(quadruplet.begin(), quadruplet.end());
                            result_set.insert(quadruplet);
                        }
                    }
                }
            }
        }
```

```
    vector<vector<int>> result(result_set.begin(), result_set.end());
    return result;
}

// Closest sum to target using meet in the middle
int closestSubsetSum(const vector<int>& arr, int target) {
    int n = arr.size();
    int n1 = n / 2;
    int n2 = n - n1;

    // Generate all subset sums for first half
    vector<int> sums1;
    for (int mask = 0; mask < (1 << n1); mask++) {
        int sum = 0;
        for (int i = 0; i < n1; i++) {
            if (mask & (1 << i)) {
                sum += arr[i];
            }
        }
        sums1.push_back(sum);
    }

    // Generate all subset sums for second half
    vector<int> sums2;
    for (int mask = 0; mask < (1 << n2); mask++) {
        int sum = 0;
        for (int i = 0; i < n2; i++) {
            if (mask & (1 << i)) {
                sum += arr[n1 + i];
            }
        }
        sums2.push_back(sum);
    }

    // Sort and remove duplicates from second half
    sort(sums2.begin(), sums2.end());
    sums2.erase(unique(sums2.begin(), sums2.end()), sums2.end());

    int closest = INT_MAX;

    // For each sum in first half, find complementary sum in second half
    for (int sum1 : sums1) {
        int needed = target - sum1;

        // Binary search in sums2
        auto it = lower_bound(sums2.begin(), sums2.end(), needed);
```

```
            if (it != sums2.end()) {
                int total = sum1 + *it;
                if (abs(total - target) < abs(closest - target)) {
                    closest = total;
                }
            }

            if (it != sums2.begin()) {
                it--;
                int total = sum1 + *it;
                if (abs(total - target) < abs(closest - target)) {
                    closest = total;
                }
            }
        }

        return closest;
    }
};
```

python

```python
# Python Implementation
from bisect import bisect_left, bisect_right
from itertools import combinations

class MeetInMiddle:
    def count_subsets_with_sum(self, arr, target):
        n = len(arr)
        n1 = n // 2
        n2 = n - n1

        # Generate all subset sums for first half
        sums1 = []
        for mask in range(1 << n1):
            total = 0
            for i in range(n1):
                if mask & (1 << i):
                    total += arr[i]
            sums1.append(total)

        # Generate all subset sums for second half
        sums2 = []
        for mask in range(1 << n2):
            total = 0
            for i in range(n2):
                if mask & (1 << i):
```

```
                total += arr[n1 + i]
            sums2.append(total)

        # Sort second half
        sums2.sort()

        # Count pairs that sum to target
        count = 0
        for sum1 in sums1:
            needed = target - sum1
            left = bisect_left(sums2, needed)
            right = bisect_right(sums2, needed)
            count += (right - left)

        return count

    def four_sum(self, nums, target):
        nums.sort()
        n = len(nums)
        pair_sums = {}

        # Store all pair sums
        for i in range(n):
            for j in range(i + 1, n):
                pair_sum = nums[i] + nums[j]
                if pair_sum not in pair_sums:
                    pair_sums[pair_sum] = []
                pair_sums[pair_sum].append((i, j))

        result = set()

        # Find complementary pairs
        for sum1, pairs1 in pair_sums.items():
            sum2 = target - sum1

            if sum2 in pair_sums:
                for i1, j1 in pairs1:
                    for i2, j2 in pair_sums[sum2]:
                        # Check if indices are distinct
                        if i1 != i2 and i1 != j2 and j1 != i2 and j1 != j2:
                            quadruplet = tuple(sorted([
                                nums[i1], nums[j1], nums[i2], nums[j2]
                            ]))
                            result.add(quadruplet)

        return [list(q) for q in result]
```

```
    def closest_subset_sum(self, arr, target):
        n = len(arr)
        n1 = n // 2
        n2 = n - n1

        # Generate all subset sums for first half
        sums1 = []
        for mask in range(1 << n1):
            total = 0
            for i in range(n1):
                if mask & (1 << i):
                    total += arr[i]
            sums1.append(total)

        # Generate all subset sums for second half
        sums2 = []
        for mask in range(1 << n2):
            total = 0
            for i in range(n2):
                if mask & (1 << i):
                    total += arr[n1 + i]
            sums2.append(total)

        # Remove duplicates and sort
        sums2 = list(set(sums2))
        sums2.sort()

        closest = float('inf')

        # For each sum in first half, find complementary in second half
        for sum1 in sums1:
            needed = target - sum1

            # Binary search
            idx = bisect_left(sums2, needed)

            if idx < len(sums2):
                total = sum1 + sums2[idx]
                if abs(total - target) < abs(closest - target):
                    closest = total

            if idx > 0:
                total = sum1 + sums2[idx - 1]
                if abs(total - target) < abs(closest - target):
                    closest = total

        return closest
```

```
# Example
arr = [1, 2, 3, 4, 5]
mim = MeetInMiddle()
print(f"Subsets with sum 5: {mim.count_subsets_with_sum(arr, 5)}")  # 3

nums = [1, 0, -1, 0, -2, 2]
print(f"4Sum for target 0: {mim.four_sum(nums, 0)}")

arr2 = [1, 2, 3, 4, 5]
print(f"Closest sum to 9: {mim.closest_subset_sum(arr2, 9)}")  # 9
```

Complete Example: Remove One Element Problems

Let's solve a comprehensive example combining multiple techniques:

cpp

```
// C++ - Remove One Element Maximum Sum
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>

using namespace std;

class RemoveOneElement {
public:
    // Method 1: Prefix + Suffix arrays
    int maxSumRemoveOnePrefixSuffix(const vector<int>& arr) {
        int n = arr.size();
        if (n <= 1) return 0;

        vector<int> prefix_max(n), suffix_max(n);

        // Build prefix max subarray sum
        prefix_max[0] = arr[0];
        int current_max = arr[0];
        for (int i = 1; i < n; i++) {
            current_max = max(arr[i], current_max + arr[i]);
            prefix_max[i] = max(prefix_max[i-1], current_max);
        }

        // Build suffix max subarray sum
        suffix_max[n-1] = arr[n-1];
        current_max = arr[n-1];
```

```
    for (int i = n-2; i >= 0; i--) {
        current_max = max(arr[i], current_max + arr[i]);
        suffix_max[i] = max(suffix_max[i+1], current_max);
    }

    // Try removing each element
    int max_sum = max(prefix_max[n-1], 0);  // no removal or empty

    for (int i = 0; i < n; i++) {
        int left = (i > 0) ? prefix_max[i-1] : 0;
        int right = (i < n-1) ? suffix_max[i+1] : 0;
        max_sum = max(max_sum, left + right);
    }

    return max_sum;
}

// Method 2: Dynamic programming approach
int maxSumRemoveOneDP(const vector<int>& arr) {
    int n = arr.size();
    if (n <= 1) return 0;

    // dp1[i]: max subarray sum ending at i
    // dp2[i]: max subarray sum starting at i
    vector<int> dp1(n), dp2(n);

    dp1[0] = arr[0];
    for (int i = 1; i < n; i++) {
        dp1[i] = max(arr[i], dp1[i-1] + arr[i]);
    }

    dp2[n-1] = arr[n-1];
    for (int i = n-2; i >= 0; i--) {
        dp2[i] = max(arr[i], dp2[i+1] + arr[i]);
    }

    // prefix_max[i]: max subarray sum in arr[0..i]
    // suffix_max[i]: max subarray sum in arr[i..n-1]
    vector<int> prefix_max(n), suffix_max(n);

    prefix_max[0] = dp1[0];
    for (int i = 1; i < n; i++) {
        prefix_max[i] = max(prefix_max[i-1], dp1[i]);
    }

    suffix_max[n-1] = dp2[n-1];
    for (int i = n-2; i >= 0; i--) {
```

```
            suffix_max[i] = max(suffix_max[i+1], dp2[i]);
        }

        int max_sum = max(prefix_max[n-1], 0);

        for (int i = 0; i < n; i++) {
            int left = (i > 0) ? prefix_max[i-1] : 0;
            int right = (i < n-1) ? suffix_max[i+1] : 0;
            max_sum = max(max_sum, left + right);
        }

        return max_sum;
    }

    // Method 3: One pass solution
    int maxSumRemoveOneOnePass(const vector<int>& arr) {
        int n = arr.size();
        if (n <= 1) return 0;

        int max_without_deletion = 0;
        int current = 0;
        for (int num : arr) {
            current = max(num, current + num);
            max_without_deletion = max(max_without_deletion, current);
        }

        int forward = 0, backward = 0;
        vector<int> forward_max(n), backward_max(n);

        forward_max[0] = arr[0];
        for (int i = 1; i < n; i++) {
            forward = max(arr[i], forward + arr[i]);
            forward_max[i] = max(forward_max[i-1], forward);
        }

        backward_max[n-1] = arr[n-1];
        for (int i = n-2; i >= 0; i--) {
            backward = max(arr[i], backward + arr[i]);
            backward_max[i] = max(backward_max[i+1], backward);
        }

        int max_with_deletion = max_without_deletion;

        for (int i = 0; i < n; i++) {
            int left = (i > 0) ? forward_max[i-1] : 0;
            int right = (i < n-1) ? backward_max[i+1] : 0;
            max_with_deletion = max(max_with_deletion, left + right);
```

```
        }

        return max_with_deletion;
    }

    // Edge case: all negative numbers
    int handleAllNegative(const vector<int>& arr) {
        int max_val = INT_MIN;
        for (int num : arr) {
            max_val = max(max_val, num);
        }
        return max_val < 0 ? max_val : maxSumRemoveOneOnePass(arr);
    }
};

int main() {
    RemoveOneElement roe;

    // Test cases
    vector<vector<int>> test_cases = {
        {1, -2, 0, 3},           // Expected: 4
        {1, -2, -2, 3},          // Expected: 3
        {-1, -1, -1, -1},        // Expected: -1
        {1, 2, 3, 4, 5},         // Expected: 14
        {1, -2, 3, 10, -4, 7, 2, -5}, // Expected: 18
    };

    for (int i = 0; i < test_cases.size(); i++) {
        cout << "Test case " << i+1 << ": ";
        cout << "Array: [";
        for (int num : test_cases[i]) {
            cout << num << " ";
        }
        cout << "]" << endl;

        cout << "Method 1 (Prefix+Suffix): "
             << roe.maxSumRemoveOnePrefixSuffix(test_cases[i]) << endl;
        cout << "Method 2 (DP): "
             << roe.maxSumRemoveOneDP(test_cases[i]) << endl;
        cout << "Method 3 (One Pass): "
             << roe.maxSumRemoveOneOnePass(test_cases[i]) << endl;
        cout << "Edge Case Handler: "
             << roe.handleAllNegative(test_cases[i]) << endl;
        cout << endl;
    }
```

```
    return 0;
}
```

python

```python
# Python - Remove One Element Maximum Sum
import sys

class RemoveOneElement:
    def max_sum_remove_one_prefix_suffix(self, arr):
        n = len(arr)
        if n <= 1:
            return 0

        prefix_max = [0] * n
        suffix_max = [0] * n

        # Build prefix max subarray sum
        prefix_max[0] = arr[0]
        current_max = arr[0]
        for i in range(1, n):
            current_max = max(arr[i], current_max + arr[i])
            prefix_max[i] = max(prefix_max[i-1], current_max)

        # Build suffix max subarray sum
        suffix_max[-1] = arr[-1]
        current_max = arr[-1]
        for i in range(n-2, -1, -1):
            current_max = max(arr[i], current_max + arr[i])
            suffix_max[i] = max(suffix_max[i+1], current_max)

        # Try removing each element
        max_sum = max(prefix_max[-1], 0)  # no removal or empty

        for i in range(n):
            left = prefix_max[i-1] if i > 0 else 0
            right = suffix_max[i+1] if i < n-1 else 0
            max_sum = max(max_sum, left + right)

        return max_sum

    def max_sum_remove_one_dp(self, arr):
        n = len(arr)
        if n <= 1:
            return 0

        # dp1[i]: max subarray sum ending at i
```

```
        # dp2[i]: max subarray sum starting at i
        dp1 = [0] * n
        dp2 = [0] * n

        dp1[0] = arr[0]
        for i in range(1, n):
            dp1[i] = max(arr[i], dp1[i-1] + arr[i])

        dp2[-1] = arr[-1]
        for i in range(n-2, -1, -1):
            dp2[i] = max(arr[i], dp2[i+1] + arr[i])

        # prefix_max[i]: max subarray sum in arr[0..i]
        # suffix_max[i]: max subarray sum in arr[i..n-1]
        prefix_max = [0] * n
        suffix_max = [0] * n

        prefix_max[0] = dp1[0]
        for i in range(1, n):
            prefix_max[i] = max(prefix_max[i-1], dp1[i])

        suffix_max[-1] = dp2[-1]
        for i in range(n-2, -1, -1):
            suffix_max[i] = max(suffix_max[i+1], dp2[i])

        max_sum = max(prefix_max[-1], 0)

        for i in range(n):
            left = prefix_max[i-1] if i > 0 else 0
            right = suffix_max[i+1] if i < n-1 else 0
            max_sum = max(max_sum, left + right)

        return max_sum

    def max_sum_remove_one_one_pass(self, arr):
        n = len(arr)
        if n <= 1:
            return 0

        max_without_deletion = 0
        current = 0
        for num in arr:
            current = max(num, current + num)
            max_without_deletion = max(max_without_deletion, current)

        forward_max = [0] * n
        backward_max = [0] * n
```

```
        forward_max[0] = arr[0]
        forward = arr[0]
        for i in range(1, n):
            forward = max(arr[i], forward + arr[i])
            forward_max[i] = max(forward_max[i-1], forward)

        backward_max[-1] = arr[-1]
        backward = arr[-1]
        for i in range(n-2, -1, -1):
            backward = max(arr[i], backward + arr[i])
            backward_max[i] = max(backward_max[i+1], backward)

        max_with_deletion = max_without_deletion

        for i in range(n):
            left = forward_max[i-1] if i > 0 else 0
            right = backward_max[i+1] if i < n-1 else 0
            max_with_deletion = max(max_with_deletion, left + right)

        return max_with_deletion

    def handle_all_negative(self, arr):
        max_val = max(arr)
        return max_val if max_val < 0 else self.max_sum_remove_one_one_pass(arr)

# Test
roe = RemoveOneElement()
test_cases = [
    [1, -2, 0, 3],            # Expected: 4
    [1, -2, -2, 3],           # Expected: 3
    [-1, -1, -1, -1],         # Expected: -1
    [1, 2, 3, 4, 5],          # Expected: 14
    [1, -2, 3, 10, -4, 7, 2, -5],  # Expected: 18
]

for i, test_case in enumerate(test_cases):
    print(f"Test case {i+1}: {test_case}")
    print(f"Method 1 (Prefix+Suffix): {roe.max_sum_remove_one_prefix_suffix(test_ca
se)}")
    print(f"Method 2 (DP): {roe.max_sum_remove_one_dp(test_case)}")
    print(f"Method 3 (One Pass): {roe.max_sum_remove_one_one_pass(test_case)}")
    print(f"Edge Case Handler: {roe.handle_all_negative(test_case)}")
    print()
```

Summary of Key Points

1. **Prefix Sum**: Foundation for range queries, O(1) sum queries after O(n) preprocessing.
2. **Prefix XOR**: Useful for bit manipulation problems, shares similar properties with prefix sum.
3. **Prefix Max/Min**: Enables solving problems like stock prices, trapping rainwater.
4. **Prefix Frequency**: Efficient range frequency queries.
5. **Prefix Hash**: String manipulation, substring comparison in O(1).
6. **Prefix GCD**: Number theory problems, range GCD queries.
7. **Prefix AND/OR**: Bitmask problems, monotonic properties.
8. **Prefix Common Count**: Sequence comparison problems.
9. **Suffix Arrays**: Mirror of prefix arrays, useful for "from i to end" problems.
10. **Prefix + Suffix**: Powerful for "remove one element" and split problems.
11. **Difference Array**: O(1) range updates, essential for interval problems.
12. **Sliding Window**: O(n) solutions for subarray/substring problems.
13. **Two Pointers**: Efficient pair/triplet finding.
14. **Meet in Middle**: Reduces exponential complexity for subset problems.

When to Use Which Technique

Problem Type	Recommended Techniques
Range sum queries	Prefix Sum
Subarray XOR problems	Prefix XOR
Maximum/minimum in subarray	Prefix/Suffix Max/Min, Sliding Window
String matching	Prefix Hash, KMP, Z-algorithm
Range updates	Difference Array
Remove one element	Prefix + Suffix arrays
Pair/triplet problems	Two Pointers, Hash Map
Subset problems (n ≤ 40)	Meet in Middle

Problem Type	Recommended Techniques
Overlapping intervals	Imos Technique
Bit manipulation	Prefix AND/OR
Frequency queries	Prefix Frequency
GCD/LCM problems	Prefix GCD

Complexity Analysis

Technique	Build Time	Query Time	Space
Prefix Sum	O(n)	O(1)	O(n)
Prefix XOR	O(n)	O(1)	O(n)
Difference Array	O(k) updates	O(n) reconstruction	O(n)
Sliding Window	O(n)	O(1) per step	O(1) or O(k)
Two Pointers	O(n)	O(1) per step	O(1)
Meet in Middle	O(2^{n/2})	O(2^{n/2} log n)	O(2^{n/2})

Practice Problems

Easy: