```cpp
#include <iostream>
#include <stack>
#include <vector>

using namespace std;

vector<int> findNearestSmallerToLeft(const vector<int>& arr) {
  int n = arr.size();
  vector<int> result(n, -1); // Initialize result with -1 for all elements
  stack<int> st;

  for (int i = 0; i < n; i++) {
    // Pop elements from stack until the top element is smaller than current element
    while (!st.empty() && arr[st.top()] >= arr[i]) {
      st.pop();
    }

    // If stack is not empty, the top element is the nearest smaller to the left
    if (!st.empty()) {
      result[i] = st.top();
    }

    // Push the current element onto the stack
    st.push(i);
  }

  return result;
}

int main() {
```

```
  vector<int> arr = {4, 5, 2, 10, 8};
  vector<int> nearestSmaller = findNearestSmallerToLeft(arr);

  cout << "Input array: ";
  for (int x : arr) {
    cout << x << " ";
  }
  cout << endl;

  cout << "Nearest smaller element to the left for each element: ";
  for (int i = 0; i < nearestSmaller.size(); i++) {
    cout << nearestSmaller[i] << " ";
  }
  cout << endl;

  return 0;
}
```

Explanation:

1. **Initialization:**
   - `result` vector is initialized with `-1` for all elements, indicating no smaller element found to the left initially.
   - `st` is an empty stack to store indices of elements.

2. **Iteration:**
   - The code iterates through the input array `arr`.

3. **Finding Nearest Smaller:**

- **Pop from stack:** While the stack is not empty and the element at the top of the stack is greater than or equal to the current element (`arr[i]`), pop elements from the stack. This ensures that the stack only contains elements smaller than the current element.

- **Update result:** If the stack is not empty after popping, the top element of the stack holds the index of the nearest smaller element to the left of the current element. Store this index in `result[i]`.

4. **Pushing to stack:**

- Push the index `i` of the current element onto the stack. This prepares the stack for future comparisons.

5. **Output:**

- The `result` vector contains the indices of the nearest smaller elements to the left for each element in the input array.

Time Complexity:

- **O(n)**: The algorithm iterates through the array once, and each element is pushed and popped from the stack at most once. Therefore, the overall time complexity is linear.

Example Output:

```
Input array: 4 5 2 10 8

Nearest smaller element to the left for each element: -1 -1 -1 2 2
```

```cpp
#include <iostream>
#include <queue>
#include <vector>

using namespace std;

vector<int> findSmallestInWindow(const vector<int>& arr, int k) {
 int n = arr.size();
 vector<int> result;
 deque<int> window; // Use a deque for efficient insertion/deletion at both ends

 // Iterate through the array
 for (int i = 0; i < n; i++) {
  // Remove elements outside the current window from the deque
  while (!window.empty() && window.front() <= i - k) {
   window.pop_front();
  }

  // Add the current element to the deque, maintaining an increasing order
  while (!window.empty() && arr[window.back()] >= arr[i]) {
   window.pop_back();
  }
  window.push_back(i);

  // If the window is full (k elements), add the smallest element to the result
  if (i >= k - 1) {
   result.push_back(arr[window.front()]);
  }
 }
```

```
  return result;
}

int main() {
  vector<int> arr = {1, 3, -1, -3, 5, 3, 6, 7};
  int k = 3;

  cout << "Input array: ";
  for (int x : arr) {
    cout << x << " ";
  }
  cout << endl;

  cout << "Smallest elements in each window of size " << k << ": ";
  vector<int> smallestElements = findSmallestInWindow(arr, k);
  for (int x : smallestElements) {
    cout << x << " ";
  }
  cout << endl;

  return 0;
}
```

Explanation:

1. **Initialization:**
   - `result` vector stores the smallest elements in each window.
   - `window` is a deque to hold the indices of elements within the current window.

2. **Iteration:**

- The code iterates through the array `arr`.

3. **Maintaining the Window:**

- **Remove elements outside the window:** If the front element of the deque is outside the current window (index `i - k`), remove it from the front.

- **Maintain increasing order:** While the last element in the deque is greater than or equal to the current element (`arr[i]`), remove it from the back. This ensures that the deque always stores elements in increasing order from front to back.

- **Add the current element:** Push the index `i` of the current element to the back of the deque.

4. **Finding Smallest in Window:**

- Once the window has `k` elements (`i >= k - 1`), the smallest element within the window is at the front of the deque (`window.front()`). Add this element to the `result` vector.

5. **Output:**

- The `result` vector contains the smallest elements in each window of size `k`.

Time Complexity:

- **O(n)**: Each element is added to and removed from the deque at most once. The overall time complexity is linear.

Example Output:

```
Input array: 1 3 -1 -3 5 3 6 7

Smallest elements in each window of size 3: -1 -3 -3 3 3
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// 2SUM Problem
pair<int, int> findTwoSum(vector<int>& arr, int target) {
  sort(arr.begin(), arr.end()); // Sort the array

  int left = 0;
  int right = arr.size() - 1;

  while (left < right) {
    int sum = arr[left] + arr[right];

    if (sum == target) {
      return make_pair(arr[left], arr[right]); // Found the pair
    } else if (sum < target) {
      left++; // Increase left pointer if sum is too small
    } else {
      right--; // Decrease right pointer if sum is too large
    }
  }

  return make_pair(-1, -1); // No pair found
}

// 3SUM Problem
vector<vector<int>> findThreeSum(vector<int>& arr, int target) {
```

```
vector<vector<int>> result;
sort(arr.begin(), arr.end());

for (int i = 0; i < arr.size() - 2; i++) {
  // Skip duplicates for the first element
  if (i > 0 && arr[i] == arr[i - 1]) {
    continue;
  }

  int left = i + 1;
  int right = arr.size() - 1;

  while (left < right) {
    int sum = arr[i] + arr[left] + arr[right];

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

      // Skip duplicates for the second and third elements
      while (left < right && arr[left] == arr[left + 1]) {
        left++;
      }
      while (left < right && arr[right] == arr[right - 1]) {
        right--;
      }

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

```
      right--;
    }
   }
  }

  return result;
}

int main() {
  vector<int> arr = {1, 4, 7, 3, 5, 9, 2, 8};
  int target = 12;

  pair<int, int> twoSumPair = findTwoSum(arr, target);
  if (twoSumPair.first != -1) {
    cout << "2SUM: " << twoSumPair.first << " + " << twoSumPair.second << " = " << target << endl;
  } else {
    cout << "2SUM: No pair found." << endl;
  }

  vector<vector<int>> threeSumTriplets = findThreeSum(arr, 12);
  cout << "3SUM: " << endl;
  for (auto triplet : threeSumTriplets) {
    cout << triplet[0] << " + " << triplet[1] << " + " << triplet[2] << " = " << target << endl;
  }

  return 0;
}
```

Explanation:

2SUM Problem:

1. **Sort the array:** This allows us to use the two-pointer approach effectively.

2. **Two pointers:** Initialize `left` at the beginning and `right` at the end of the sorted array.

3. **Iteration:**

   - Calculate the `sum` of the elements pointed by `left` and `right`.

   - If `sum` is equal to `target`, we've found the pair.

   - If `sum` is less than `target`, move the `left` pointer one step to the right to increase the sum.

   - If `sum` is greater than `target`, move the `right` pointer one step to the left to decrease the sum.

4. **Return:** Return the found pair or `(-1, -1)` if no pair is found.

3SUM Problem:

1. **Sort the array:** Similar to 2SUM, this allows efficient searching.

2. **Outer loop:** Iterate through the array using the outer loop (`i`). For each `arr[i]`, we'll find the remaining two elements.

3. **Skip duplicates for the first element:** To avoid repeated triplets, we skip over duplicate values of `arr[i]` by checking if `i > 0` and `arr[i] == arr[i-1]`.

4. **Inner loop:** Use two pointers, `left` and `right`, starting from `i+1` and the end of the array, respectively.

5. **Find triplets:**

   - Calculate the `sum` of `arr[i]`, `arr[left]`, and `arr[right]`.

   - If `sum` equals `target`, we've found a triplet.

   - Skip duplicates for the second and third elements: Similar to the first element, skip over duplicates for `arr[left]` and `arr[right]` to avoid repeated triplets.

   - Move the pointers: If `sum` is less than `target`, move `left` to the right; if `sum` is greater than `target`, move `right` to the left.

6. **Return:** The `result` vector will contain all the unique triplets that sum up to the `target`.

Time Complexity:

- **2SUM: O(n log n)** due to sorting (the two-pointer search is O(n)).

- **3SUM: O(n^2)**. The outer loop runs for `n` iterations, and the inner loop (two pointers) runs for `n` iterations in the worst case, giving a total time complexity of O(n^2).

Example Output:

```
2SUM: 4 + 8 = 12

3SUM:

1 + 3 + 8 = 12
```

```cpp
#include <iostream>
#include <vector>

using namespace std;

// Function to find a subarray with the given sum
vector<int> findSubarraySum(const vector<int>& arr, int targetSum) {
  int n = arr.size();
  int currentSum = 0;
  int start = 0;
  int end = 0;

  while (end < n) {
    currentSum += arr[end];

    // If the current sum is equal to the target sum, we've found the subarray
    if (currentSum == targetSum) {
      return vector<int>(arr.begin() + start, arr.begin() + end + 1);
    }

    // If the current sum exceeds the target sum, we need to remove elements from the beginning
    while (currentSum > targetSum && start <= end) {
      currentSum -= arr[start];
      start++;

      // If the current sum is equal to the target sum, we've found the subarray
      if (currentSum == targetSum) {
        return vector<int>(arr.begin() + start, arr.begin() + end + 1);
      }
    }
```

```
    end++; // Move the right pointer to the next element
  }

  // If no subarray is found, return an empty vector
  return {};
}

int main() {
  vector<int> arr = {1, 4, 20, 3, 10, 5};
  int targetSum = 8;

  vector<int> subarray = findSubarraySum(arr, targetSum);

  if (!subarray.empty()) {
    cout << "Subarray with sum " << targetSum << ": ";
    for (int x : subarray) {
      cout << x << " ";
    }
    cout << endl;
  } else {
    cout << "No subarray found with sum " << targetSum << endl;
  }

  return 0;
}
```

Explanation:

1. **Initialization:**

- `currentSum`: Stores the sum of elements in the current window.
- `start`: Index of the starting element of the current window.
- `end`: Index of the ending element of the current window.

2. **Iteration:**
   - The code iterates through the array using the `end` pointer.

3. **Expanding the window:**
   - The `end` pointer is moved to include the next element in the window.
   - The `currentSum` is updated to include the newly added element.

4. **Shrinking the window:**
   - If `currentSum` becomes greater than `targetSum`, we need to shrink the window from the beginning.
   - The `start` pointer is moved to the right, removing elements from the window.
   - The `currentSum` is updated to remove the removed elements.

5. **Checking for target sum:**
   - If at any point `currentSum` becomes equal to `targetSum`, we have found the desired subarray.

6. **Return:**
   - If a subarray is found, it's returned as a vector.
   - If no subarray is found, an empty vector is returned.

Time Complexity:
- **O(n)**: Both pointers `start` and `end` move at most `n` times, making the time complexity linear.

Example Output:

```
Subarray with sum 8: 4 3 1
```

This code correctly identifies the subarray `[4, 3, 1]` that sums to 8.

Bài 9: Truy vấn phạm vi

Giới thiệu bài toán:

* Bài toán truy vấn phạm vi là một dạng bài toán phổ biến trong lập trình thi đấu. Trong bài toán này, ta cần xử lý các truy vấn trên một mảng con của mảng cho trước với độ phức tạp thấp.

* Ví dụ:

  * `Sumq(a,b)`: Tìm tổng các số trong đoạn `[a,b]` của mảng.

  * `Minq(a,b)`: Tìm giá trị nhỏ nhất trong đoạn `[a,b]` của mảng.

  * `Maxq(a,b)`: Tìm giá trị lớn nhất trong đoạn `[a,b]` của mảng.

* Sử dụng vòng lặp đơn giản cho mỗi truy vấn sẽ có độ phức tạp `O(n)`, với `q` truy vấn sẽ là `O(nq)`. Nếu `n` và `q` lớn, thuật toán sẽ bị TLE (Time Limit Exceeded).

Truy vấn mảng tĩnh:

* Xét bài toán mảng không thay đổi trong quá trình truy vấn (mảng tĩnh).

* Có một số phương pháp để xử lý các truy vấn cụ thể:

  * **Bài toán `Sum queries`:** Sử dụng mảng cộng dồn (prefix sum array).

  * **Bài toán `Minimum queries`:** Sử dụng bảng sparse table.

Prefix sum array:

* Tạo mảng cộng dồn `sumq(0,k)` là tổng giá trị từ `0` đến `k`.

* Ví dụ:

Mảng	1	2	3	4	5	6
Giá trị	2	5	1	6	3	2
`sumq`	2	7	8	14	17	19

* Tính tổng từ `a` đến `b`: `sumq(a,b) = sumq(0,b) - sumq(0,a-1)`.

* Mảng cộng dồn 2 chiều: Tính tổng mảng con 2 chiều bằng cách sử dụng 4 điểm góc của mảng con.

Minimum queries:

* Sử dụng bảng sparse table để lưu trữ giá trị `minq(a,b)` với chiều dài đoạn `[a,b]` là lũy thừa của 2.

* Xây dựng các bảng với độ dài đoạn là 2, 4, 8, ...

* Tính giá trị của bảng bằng công thức truy hồi: `minq(a,b) = min(minq(a,a + mid - 1),minq(a + mid, b))`, với `mid = (b-a+1)/2`.

* Truy vấn `minq(x,y)` bất kỳ với độ phức tạp `O(1)`:

    * `minq(x,y) = min(minq(x,x+k-1), minq(y-k+1,y))`, với `k = 2^x` là số lớn nhất thỏa mãn `2^x ≤ y-x+1`.

Binary indexed tree (BIT):

* Cây BIT hay cây Fenwick là cấu trúc dữ liệu hỗ trợ 2 loại truy vấn với độ phức tạp `O(logn)`:

    * Tổng giá trị của các phần tử trong đoạn `a, b`.

    * Cập nhật giá trị tại phần tử `x`.

* Cây BIT được biểu diễn bằng mảng.

* `p(k)` là giá trị lớn nhất có dạng `2^x` mà `k` chia hết cho nó.

* `tree[k] = sumq(k-p(k)+1,k)`.

* **Tính tổng từ `1` đến `x`:** Sử dụng phương pháp chia để trị.

    * `sumq(1,x) = sumq(x,x) + sumq(x-p(x),x-1) + sumq(x-p(x)-p(x-p(x)),x-p(x)-1) ...`.

* **Tính tổng từ `a` đến `b`:** `sumq(a,b) = sumq(1,b) - sumq(1,a-1)`.

* **Cập nhật giá trị:** Cập nhật theo chiều ngược lại, từ phần tử `k` lên node cha của nó.

* **Code tính `sumq(1,k)` và cập nhật `add(k,x)`:**

```cpp
```

```
int sumq(int k) {
   int sum = 0;
   while (k > 0) {
      sum += tree[k];
      k -= k & -k;
   }
   return sum;
}

void add(int k, int x) {
   while (k <= n) {
      tree[k] += x;
      k += k & -k;
   }
}
```

Segment tree:

* Cây phân đoạn là cấu trúc dữ liệu giúp ích cho việc truy vấn đoạn và cập nhật phần tử với độ phức tạp `O(logn)`.

* Nó có thể hỗ trợ nhiều loại truy vấn:

   * Tổng đoạn.

   * Giá trị nhỏ nhất/ lớn nhất trong đoạn.

   * UCLN nhất trọng đoạn.

   * Toán tử trên bit (and, or, xor) trong đoạn.

* Cấu trúc cây phân đoạn là một cây nhị phân, mỗi node chứa thông tin cần thiết để xử lý truy vấn đoạn chứa các node con của nó.

* Lưu trữ các node của cây phân đoạn bằng một mảng `2n` phần tử, với `n` là số phần tử của mảng input.

   * Node gốc: `tree[1]`.

   * Node con của `tree[k]`: `tree[2*k]` (trái) và `tree[2*k+1]` (phải).

* **Code tính tổng các phần tử trong đoạn `[a,b]`:**

```cpp
int query(int a, int b) {
    int sum = 0;
    a += n; // Adjust index for segment tree
    b += n;
    while (a <= b) {
        if (a % 2 == 1) sum += tree[a++];
        if (b % 2 == 0) sum += tree[b--];
        a /= 2;
        b /= 2;
    }
    return sum;
}
```

* **Code cập nhật giá trị tại phần tử `k`:**

```cpp
void update(int k, int x) {
    k += n;
    tree[k] = x;
    while (k > 1) {
        k /= 2;
        tree[k] = tree[2*k] + tree[2*k+1];
    }
}
```

Một số kỹ thuật liên quan:

* **Index compression:**
    * Sử dụng khi chỉ số của mảng lớn.
    * Nén chỉ số hiện tại `x` thành chỉ số mới `c(x)` sao cho:
        * Với `a<b` thì `c(a)<c(b)`.
    * Sử dụng `map` để thực hiện nén chỉ số.

* **Cây phân đoạn cập nhật đoạn:**
    * Cập nhật một đoạn giá trị.
    * Chuyển mảng thành mảng sai khác (difference array).
    * Chỉ cần cập nhật giá trị tại hai điểm đầu và cuối của đoạn.
    * Sử dụng BIT hoặc segment tree.

Bài tập:

* Viết hàm tìm vị trí `x` nhỏ nhất sao cho tổng các phần tử trong đoạn `[1,x]` lớn hơn hoặc bằng `f`. Trả về `x=n` nếu không tìm thấy.
* Viết hàm tìm vị trí của giá trị nhỏ nhất trong đoạn `[a,b]` với độ phức tạp `O(logn)`.

Lưu ý:

* Việc lựa chọn cấu trúc dữ liệu phù hợp phụ thuộc vào yêu cầu của bài toán.
* Cây BIT thường được sử dụng cho các bài toán tính tổng, còn cây phân đoạn có thể xử lý nhiều loại truy vấn hơn.
* Index compression và các kỹ thuật cập nhật đoạn giúp giải quyết các bài toán với chỉ số lớn hoặc cập nhật đoạn.

Bài 10: Làm việc với bit

1. Bit representation

* **Biểu diễn số nguyên:**
  * Trong C++, kiểu `int` là kiểu 32-bit, tức là mỗi số nguyên được lưu trữ dưới dạng nhị phân gồm 32 bit.
  * Ví dụ: Số 43 khi biểu diễn bằng kiểu `int`:

    ```
    00000000 00000000 00000000 00101011
    ```

  * Giá trị của một số nhị phân `bk…b2b1b0` là:

    ```
    b0 * 2^0 + b1 * 2^1 + ... + bk * 2^k
    ```

* **Kiểu Signed (có dấu) và Unsigned (không dấu):**
  * **Signed:** Biểu diễn các số từ `-2^(n-1)` đến `2^(n-1)-1`.
  * **Unsigned:** Biểu diễn các số từ `0` đến `2^n - 1`.
  * Số `Signed -x` và số `Unsigned 2^n-x` được biểu diễn bằng bit như nhau.
  * Ví dụ:
    * Số 43 (Unsigned): `00000000 00000000 00000000 00101011`
    * Số -43 (Signed): `11111111 11111111 11111111 11010101`

2. Bit operations

* **Toán tử:**
  * **AND (`&`):**
    * `x & y = 1` <=> `x = 1` và `y = 1`.
  * **OR (`|`):**
    * `x | y = 1` <=> `x = 1` hoặc `y = 1`.
  * **XOR (`^`):**

    * `x ^ y = 1` <=> `x != y`.
  * **NOT (`~`):**
    * Đảo các bit của `x`.

* **Dịch bit:**
  * **Dịch trái (`<<`):**
    * `x << k` tương đương với `x * 2^k`.
  * **Dịch phải (`>>`):**
    * `x >> k` tương đương với `x / 2^k`.

* **Hàm thông dụng:**
  * `__builtin_clz(x)`: Đếm số bit 0 đầu tiên.
  * `__builtin_ctz(x)`: Đếm số bit 0 cuối cùng.
  * `__builtin_popcount(x)`: Đếm số bit 1.
  * `__builtin_parity(x)`: Trả về 1 nếu số bit 1 là lẻ, 0 nếu số bit 1 là chẵn.

3. Representing sets

* **Biểu diễn tập hợp:**
  * Mỗi tập con của tập `{0,1,2,..., n-1}` được biểu diễn bằng một số nguyên `n`-bit.
  * Ví dụ: Tập `{1,3,4,8}` được biểu diễn bằng số:
    ```
    2^8 + 2^4 + 2^3 + 2^1 = 282
    ```

* **Code:**
  ```cpp
  #include <iostream>
  #include <bitset>

  using namespace std;
  ```

```
int main() {
    int n = 10;
    bitset<32> set1;

    // Thêm phần tử vào tập hợp
    set1[1] = 1;
    set1[3] = 1;
    set1[4] = 1;
    set1[8] = 1;

    // Đếm số phần tử trong tập hợp
    int count = set1.count();
    cout << "Số phần tử trong tập hợp: " << count << endl;

    // Duyệt qua các phần tử trong tập hợp
    for (int i = 0; i < n; i++) {
        if (set1[i] == 1) {
            cout << i << " ";
        }
    }
    cout << endl;

    return 0;
}
```

* **Toán tử trên tập hợp:**
    * **Union (`|`):** Hợp của hai tập.
    * **Intersection (`&`):** Giao của hai tập.
    * **Difference (`^`):** Hiệu của hai tập.

* **Complement (`~`):** Phần bù của tập hợp.

* **Code ví dụ: Tìm số lượng phần tử thuộc ít nhất một trong hai tập:**

```cpp
int countElements(int set1, int set2) {
    return __builtin_popcount(set1 | set2);
}
```

* **Tạo tất cả tập con:**

```cpp
for (int i = 0; i < (1 << n); i++) {
    // i là tập con hiện tại
    // ...
}
```

* **Xử lý tập có đúng k phần tử:**

```cpp
for (int i = 0; i < (1 << n); i++) {
    if (__builtin_popcount(i) == k) {
        // i là tập con có đúng k phần tử
        // ...
    }
}
```

4. Bit optimizations

* Có thể tối ưu hóa một số thuật toán bằng toán tử trên bit.

* Ví dụ: **Bài toán Hamming distances**

```cpp
// Cách thông thường
int hamming(string a, string b) {
    int count = 0;
    for (int i = 0; i < a.size(); i++) {
        if (a[i] != b[i]) {
            count++;
        }
    }
    return count;
}

// Sử dụng XOR
int hamming(int a, int b) {
    return __builtin_popcount(a ^ b);
}
```

* Ví dụ: **Bài toán Counting subgrids**
    * Tìm tất cả các hình chữ nhật con có 4 góc màu đen trong một hình chữ nhật n × m.
    * Sử dụng bit để lưu trữ thông tin về màu sắc các cột.
    * Code:
        ```cpp
        #include <iostream>
        #include <vector>

        using namespace std;

        int main() {
            int n, m;
```

```
    cin >> n >> m;

    vector<vector<char>> color(n, vector<char>(m));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> color[i][j];
        }
    }

    long long count = 0;
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            long long blackCols = 0;
            for (int k = 0; k < m; k++) {
                if (color[i][k] == 'B' && color[j][k] == 'B') {
                    blackCols |= (1LL << k); // Set bit tại vị trí k
                }
            }
            count += blackCols * (blackCols - 1) / 2; // Tính số hình chữ nhật con
        }
    }

    cout << count << endl;
    return 0;
}
```

5. Dynamic programming trên bit

* **Bài toán:** Cho giá của `k` sản phẩm trong `n` ngày. Mỗi ngày chỉ mua được tối đa 1 sản phẩm. Hỏi cần bao nhiêu tiền để mua hết `k` sản phẩm?

* **Hướng giải:**
  * Sử dụng dynamic programming để tính tổng giá tối thiểu khi mua tập sản phẩm `S` trong `d` ngày:
    * `total(S,d)`: Tổng giá tối thiểu.
    * `total(S,d) = min(total(S,d-1), total(S\{x\},d-1) + price[x][d])`,
      với `x` là sản phẩm được mua vào ngày `d`.
* **Code:**

```cpp
#include <iostream>
#include <vector>
#include <limits>

using namespace std;

int main() {
    int n, k;
    cin >> n >> k;
    vector<vector<int>> price(k, vector<int>(n));
    for (int i = 0; i < k; i++) {
        for (int j = 0; j < n; j++) {
            cin >> price[i][j];
        }
    }

    vector<vector<int>> dp(1 << k, vector<int>(n + 1, numeric_limits<int>::max()));
    dp[0][0] = 0; // Không mua sản phẩm nào trong ngày 0

    for (int d = 1; d <= n; d++) {
        for (int s = 0; s < (1 << k); s++) {
            dp[s][d] = dp[s][d - 1]; // Không mua sản phẩm nào trong ngày d
            for (int x = 0; x < k; x++) {
                if ((s & (1 << x)) == 0) { // Nếu sản phẩm x chưa được mua
```

```
                dp[s | (1 << x)][d] = min(dp[s | (1 << x)][d], dp[s][d - 1] + price[x][d - 1]); // Mua sản phẩm x vào ngày d
            }
        }
    }
}

cout << dp[(1 << k) - 1][n] << endl; // Tổng giá tối thiểu để mua hết k sản phẩm
return 0;
}
```

6. Bài toán "Từ hoán vị đến tập con"

* **Bài toán:** Cho `n` người, mỗi người có khối lượng khác nhau muốn qua sông cùng lúc. Mỗi thuyền chỉ chở tối đa `X` trọng lượng, hỏi cần ít nhất bao nhiêu thuyền?
* **Hướng giải:**
    * **Cách 1:** Tạo tất cả các hoán vị rồi kiểm tra từng hoán vị. ĐPT là `O(n!n)`, không khả thi khi `n` lớn.
    * **Cách 2:** Chuyển về bài toán tập con.
        * `last(S)`: Khối lượng còn trống của thuyền cuối cùng khi đưa tập `S` lên thuyền.
        * Kiểm tra xem có thể đưa người `p` lên thuyền:
            * Nếu `last(S\{p}) + w[p] <= x` thì có thể đưa người `p` lên thuyền.
            * Ngược lại, phải thêm thuyền mới.
* **Code:**

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main() {
    int n, X;
```

```
cin >> n >> X;

vector<int> w(n);
for (int i = 0; i < n; i++) {
   cin >> w[i];
}

int boats = 0;
int currentWeight = 0;

for (int i = 0; i < (1 << n); i++) {
   int currentSet = 0;
   for (int j = 0; j < n; j++) {
      if ((i & (1 << j)) != 0) { // Người j thuộc tập con hiện tại
         currentSet |= (1 << j); // Thêm người j vào tập con
         currentWeight += w[j];
      }
   }

   if (currentWeight > X) { // Cần thêm thuyền mới
      boats++;
      currentWeight = 0;
   }
}

if (currentWeight > 0) { // Kiểm tra thuyền cuối
   boats++;
}

cout << boats << endl; // Số thuyền tối thiểu
return 0;
```

}

```

**Lưu ý:**

* Việc sử dụng bit giúp tối ưu hóa một số thuật toán, đặc biệt là khi xử lý các tập hợp con.

* Dynamic programming trên bit là một kỹ thuật hiệu quả để giải quyết các bài toán đệ quy với không gian trạng thái lớn.

* Hãy thử áp dụng các kỹ thuật bit manipulation để giải quyết các bài toán trong lập trình thi đấu.
```

Bài 28: Segment trees revisited

1. Sơ lược lại Segment trees

* **Cấu trúc:** Cây phân đoạn (Segment Tree) là một cấu trúc dữ liệu cây nhị phân hiệu quả cho việc truy vấn đoạn và cập nhật phần tử trong một mảng. Mỗi node trong cây lưu trữ thông tin về một đoạn con của mảng.

* **Phương pháp:** Có hai phương pháp xây dựng Segment Tree:

  * **Bottom-to-top:** Xây dựng từ nút lá lên nút gốc.

  * **Top-to-bottom:** Xây dựng từ nút gốc xuống nút lá.

* **Độ phức tạp:** Độ phức tạp của mỗi truy vấn là O(logn), trong đó n là số lượng phần tử trong mảng.

* **Ưu điểm:**

  * Hỗ trợ 2 loại truy vấn chính:

    * Cập nhật một phần tử.

    * Tính tổng/min/max/xor/gcd của đoạn [a,b].

* **Nhược điểm:**

  * Không thể hỗ trợ cùng lúc 2 truy vấn:

    * Cập nhật đoạn [a,b].

    * Tính tổng/min/max/xor/gcd của đoạn [a,b].

2. Lan truyền lười (Lazy propagation)

* **Kỹ thuật:** Lazy propagation giúp hỗ trợ cả cập nhật đoạn và tính toán trên đoạn [a,b] với độ phức tạp O(logn).

* **Ý tưởng:** Khi cập nhật đoạn [a,b], ta lưu thông tin cập nhật tại node hiện tại và chỉ cập nhật thực sự khi cần thiết.

* **Ví dụ:**

  * Bài toán tăng đoạn [a,b] lên x đơn vị:

    * Lưu thông tin `lazy` tại mỗi node để biểu diễn việc tăng giá trị.

    * Chỉ cập nhật giá trị tại node khi cần tính toán, bằng cách "push down" thông tin `lazy` xuống các node con.

3. Cập nhật theo đa thức

* **Cập nhật đa thức:** Khi cập nhật đoạn [a,b] theo đa thức `p(u) = tkuk + ... + t1u1 + t0`, ta lưu trữ thêm `k+1` giá trị `zk`, `zk-1`, ..., `z0` tại mỗi node.

* **Công thức tính:** Tổng các phần tử trong đoạn [x,y] được tính theo công thức:

```
s + z1(y-x)(y-x+1)/2 + z0(y-x+1)
```

* **Lưu trữ:** Thông tin `s` lưu trong mảng `tree` và `k+1` giá trị `zk` lưu trong mảng `lazy`.

4. Dynamic trees

* **Cấu trúc:** Dynamic trees cho phép tạo ra các node mới khi cần thiết, thay vì phải tạo sẵn tất cả các node như Segment Tree thông thường.

* **Ưu điểm:** Tiết kiệm bộ nhớ khi xử lý các mảng thưa thớt hoặc có nhiều node rỗng.

* **Code:**

```cpp
struct Node {
    int val;
    int lazy;
    Node *left, *right;

    Node(int v) : val(v), lazy(0), left(nullptr), right(nullptr) {}
};

// Hàm tạo node mới
Node *newNode(int v) {
    return new Node(v);
}

// Hàm xây dựng cây phân đoạn
Node *build(int a, int b) {
    if (a == b) {
```

```
        return newNode(arr[a]);
    }
    int mid = (a + b) / 2;
    return newNode(0)->left = build(a, mid), newNode(0)->right = build(mid + 1, b);
}

// Hàm cập nhật giá trị
void update(Node *node, int a, int b, int l, int r, int val) {
    // Cập nhật giá trị tại node hiện tại
    node->val += val * (r - l + 1);
    node->lazy += val;

    // Xử lý các node con
    if (a <= l && r <= b) return;
    int mid = (l + r) / 2;
    if (a <= mid) update(node->left, a, b, l, mid, val);
    if (mid + 1 <= b) update(node->right, a, b, mid + 1, r, val);
}

// Hàm tính tổng
int query(Node *node, int a, int b, int l, int r) {
    // Cập nhật giá trị tại node hiện tại
    if (node->lazy != 0) {
        node->val += node->lazy * (r - l + 1);
        if (l != r) {
            node->left->lazy += node->lazy;
            node->right->lazy += node->lazy;
        }
        node->lazy = 0;
    }
```

```
    // Xử lý các node con
    if (a <= l && r <= b) return node->val;
    int mid = (l + r) / 2;
    int sum = 0;
    if (a <= mid) sum += query(node->left, a, b, l, mid);
    if (mid + 1 <= b) sum += query(node->right, a, b, mid + 1, r);
    return sum;
}
```

5. Persistent segment trees

* **Kỹ thuật:** Persistent segment trees cho phép lưu trữ lịch sử cập nhật của cây phân đoạn.

* **Lưu trữ:** Mỗi lần cập nhật, ta tạo ra một cây phân đoạn mới dựa trên cây cũ, nhưng chỉ thay đổi các node cần cập nhật.

* **Ưu điểm:** Cho phép truy vấn giá trị của mảng ở các thời điểm khác nhau trong quá khứ.

6. Data structures

* **Kết hợp với cấu trúc dữ liệu khác:** Segment Tree có thể kết hợp với các cấu trúc dữ liệu khác như `map`, `set` để giải quyết các bài toán phức tạp hơn.

    * **Ví dụ:** Cho mảng `arr` và các truy vấn: Đếm xem trong đoạn [a,b] trong mảng có bao nhiêu lần xuất hiện `x`.

        * Sử dụng `map` để lưu trữ số lần xuất hiện của mỗi phần tử trong mỗi node của Segment Tree.

7. Two-dimensionality

* **Cây phân đoạn 2 chiều:** Sử dụng để hỗ trợ việc truy vấn hình chữ nhật con trong một mảng 2 chiều.

* **Cấu trúc:** Tạo một cây lớn tương ứng với các hàng của mảng, mỗi node của cây lớn chứa một cây con tương ứng với các phần tử của cột.

Lưu ý:

* Việc lựa chọn kỹ thuật phù hợp phụ thuộc vào yêu cầu cụ thể của bài toán.

* Hãy thử áp dụng các kỹ thuật Segment Tree, Lazy propagation, Dynamic trees và Persistent Segment Trees để giải quyết các bài toán trong lập trình thi đấu.