To effectively solve the types of C++ programming questions found in the provided competition problems, you should focus on mastering the following concepts and techniques:

**I. C++ Fundamentals**

* **Basic Syntax and Data Types:** Understand core C++ syntax, primitive data types (integers, floating-point numbers, characters, booleans), and their appropriate use. Pay attention to long long for problems involving potentially large numbers.
* **Control Flow:** Proficiency in if/else, switch, for, while, and do-while loops for controlling program execution.
* **Functions:** How to define, declare, and use functions, including passing parameters and returning values.
* **Input/Output:** Standard input (cin) and output (cout) operations for reading problem data and printing results.

**II. Standard Template Library (STL)**

The STL is critical for efficient and concise C++ competitive programming.

* **Containers:**
  + std::vector: Dynamic arrays for flexible data storage and manipulation. This is frequently used for lists of items, such as orb weights 3, car costs 4, or stone properties5.
  + std::string: For handling sequences of characters, especially relevant for problems involving text or binary strings6.
  + std::map and std::unordered\_map: For efficient key-value pair storage and retrieval, useful for associating patterns with weights 7777 or managing doll counts in different scenarios.
  + std::set and std::priority\_queue: std::priority\_queue is particularly useful for problems requiring efficient retrieval of the minimum or maximum element, such as sorting or selecting orbs by weight8.
* **Algorithms:**
  + std::sort: Essential for sorting collections of data, such as orb weights9.
  + std::pair: Useful for grouping two related values, like coordinates or connection points for routes10.
  + Other useful algorithms like std::min\_element, std::max\_element, std::lower\_bound, and std::upper\_bound for efficient operations on sorted data.

**III. Algorithms and Data Structures**

These are the backbone of solving complex programming problems.

* **Sorting and Searching:** Beyond std::sort, understand the principles of various sorting algorithms and binary search for efficient data lookup.
* **Graph Theory:** Problems like "Route" 11and "Tour" 12 heavily rely on graph concepts.
  + **Representations:** Adjacency lists (more common) and adjacency matrices for representing connections between entities (e.g., locations, stores).
  + **Traversal Algorithms:** Breadth-First Search (BFS) and Depth-First Search (DFS) for exploring graph structures.
  + **Minimum Spanning Tree (MST):** Algorithms like Kruskal's or Prim's, which are relevant for finding the minimum cost to connect all nodes in a graph. The "Route" problem asks for maximizing the MST cost13.
  + **Disjoint Set Union (DSU):** A data structure to efficiently manage sets of elements partitioned into a number of disjoint (non-overlapping) subsets, often used with MST algorithms for connectivity checks.
* **Dynamic Programming (DP):** Many optimization problems, such as "Sculpture" 14, "Tour" 15, and "Bit String"16, can be efficiently solved using dynamic programming by breaking them down into smaller, overlapping subproblems. Familiarize yourself with memoization and tabulation techniques.
* **Greedy Algorithms:** Understanding when a greedy approach (making locally optimal choices) can lead to a globally optimal solution.
* **Bit Manipulation:** Direct manipulation of bits, which is explicitly relevant for the "Bit String" problem17.

**IV. Problem-Solving Techniques**

* **Problem Understanding:** Carefully read and fully comprehend the problem statement, including all constraints and examples. The problems provided often have detailed descriptions and examples.
* **Problem Decomposition:** Break down large, complex problems into smaller, more manageable sub-problems.
* **Algorithm and Data Structure Selection:** Choose the most appropriate algorithms and data structures based on the problem's requirements and constraints.
* **Complexity Analysis:** Evaluate the time and space complexity of your solutions to ensure they meet performance requirements (e.g., "processing time" and "maximum memory usage" are specified in the problem statements 19191919).
* **Edge Cases and Constraints:** Always consider edge cases and boundary conditions to ensure your solution is robust.
* **Debugging:** Develop strong debugging skills to identify and fix logical and runtime errors.

By focusing on these areas, you will build a strong foundation for tackling competitive programming questions in C++.