Based on the types of competitive programming problems you've been working on (like "Tour," "Bit String," and "Lover"), here's a breakdown of key C++ concepts and algorithmic knowledge that would be beneficial to learn:

**I. C++ Fundamentals and Language Features**

* **Basic Syntax and Control Flow:**
  + Variables (integers, floating-point, characters, booleans)
  + Operators (arithmetic, relational, logical, bitwise)
  + Conditional statements (if-else, switch)
  + Loops (for, while, do-while)
  + Functions (declaration, definition, parameters, return types)
* **Data Types and Memory Management:**
  + Understanding different integer types (int, long long, unsigned int) and their ranges to prevent overflow.
  + Arrays (static and dynamic).
  + Pointers and references (basic understanding for passing arguments efficiently).
* **Standard Library (STL) Containers:**
  + std::vector: Dynamic arrays for flexible sizing. Essential for most problems.
  + std::string: For handling text and bit strings.
  + std::map / std::unordered\_map: For efficient key-value pair storage and lookups (e.g., storing pattern weights, memoization tables). unordered\_map is generally faster but map provides sorted keys.
  + std::set / std::unordered\_set: For storing unique elements.
  + std::queue, std::priority\_queue: For BFS and Dijkstra's algorithm, respectively.
  + std::stack: For DFS or specific parsing tasks.
* **Standard Library (STL) Algorithms:**
  + std::sort: For sorting arrays/vectors.
  + std::min, std::max: For finding minimum/maximum values.
  + std::accumulate: For summing elements in a range.
* **Input/Output:**
  + cin, cout for standard input/output.
  + Optimizing I/O with ios\_base::sync\_with\_stdio(false) and cin.tie(NULL) for faster execution in competitive programming.
* **Bitwise Operations:**
  + Understanding and applying bitwise AND (&), OR (|), XOR (^), NOT (~), left shift (<<), and right shift (>>). Crucial for problems like "Bit String" where you manipulate binary representations.

**II. Core Algorithms**

* **Dynamic Programming (DP):**
  + **Basic DP:** Understanding the concept of optimal substructure and overlapping subproblems.
  + **Memoization (Top-Down DP):** Recursive approach with caching results (as seen in "Bit String").
  + **Tabulation (Bottom-Up DP):** Iterative approach filling a DP table (as seen in "Lover").
  + **Prefix Sums:** A technique to calculate sums over ranges efficiently in O(1) time after O(N) preprocessing (as seen in "Lover").
  + **Bitmask DP:** DP where states are represented by bitmasks (as seen in "Bit String").
* **Graph Algorithms:**
  + **Graph Representation:** Adjacency lists (most common for sparse graphs), adjacency matrices.
  + **Traversal Algorithms:**
    - Breadth-First Search (BFS): For shortest path on unweighted graphs, finding all reachable nodes.
    - Depth-First Search (DFS): For general graph traversal, cycle detection, topological sorting.
  + **Shortest Path Algorithms:**
    - Dijkstra's Algorithm: For shortest path on graphs with non-negative edge weights.
    - Bellman-Ford Algorithm: For shortest path on graphs with negative edge weights.
    - Floyd-Warshall Algorithm: For all-pairs shortest path.
  + **Minimum Spanning Tree (MST) Algorithms:** Prim's, Kruskal's.
  + **Topological Sort:** For directed acyclic graphs (DAGs). Implicitly used in some DP problems on DAGs.

**III. Data Structures**

Beyond basic STL containers, understanding the underlying principles and common use cases of:

* **Trees:** Binary Trees, Binary Search Trees (BSTs).
* **Heaps/Priority Queues:** For efficient retrieval of min/max elements.
* **Disjoint Set Union (DSU):** For managing sets of elements and checking connectivity.
* **Segment Trees/Fenwick Trees (BITs):** For efficient range queries and updates (sums, min/max). These are more advanced but highly valuable for O(log N) operations.

**IV. Mathematical Concepts**

* **Number Theory Basics:** Primality testing, GCD, LCM, modular arithmetic (for problems involving large numbers and specific operations).
* **Combinatorics:** Permutations, combinations.

**V. Problem-Solving Techniques**

* **Greedy Algorithms:** Making locally optimal choices hoping for a global optimum.
* **Divide and Conquer:** Breaking problems into smaller subproblems.
* **Backtracking:** For exploring all possible solutions (often recursive).
* **Two Pointers:** For efficiently processing arrays or lists.
* **Binary Search:** For searching in sorted ranges or optimizing answers.

By focusing on these areas, you'll build a strong foundation to approach a wide variety of competitive programming problems using C++.