Standard Data Structures and Libraries

- Data structures are integral in solving many problems in computer science.
- Sometimes the problem can be solved simply by using the correct data structures.
- Sometimes data structures are needed for other algorithms.
- It is important to be familiar with standard libraries so we do not reinvent the wheel.

Common useful STL Algorithms

- `fill`, `max_element`, `min_element`, `find`, `sort`
- `find_first_of`, `count`, `copy`, `replace`, `reverse`
- `lower_bound`, `upper_bound`, `equal_range`
- `next_permutation`, `prev_permutation`
- `accumulate`, `partial_sum`, `adjacent_difference`

STL pair

- Quick and dirty way of representing a pair of two things.
- Use `make_pair` to construct new pair when needed.
- Lexicographic comparison is automatically defined.

STL sequences

- Include `vector`, `deque`, and `list`.
- Last two are seldomly used in contest problems.
- `list` is useful if we need to insert/remove anywhere, but access can be much slower than `vector` because of "locality problems".

Special STL containers

- Useful in many algorithms: `stack`, `queue`, `priority_queue`.
- Stack is LIFO. Can be useful in parsing (e.g. bracket matching, infix evaluation). Also to avoid recursion to prevent stack overflow. $O(1)$ each operation.
- Queue is FIFO. Often useful in simulation (also BFS for graph algorithms). $O(1)$ each operation.
- Priority Queue: always remove the largest one. $O(\log n)$ each operation.

Priority Queue

- To have a priority queue to select smallest one, you can negate all elements (if they are numeric), but it doesn't always work. Why not?
- Alternatively, declare

```
priority_queue<int, vector<int>, greater<int> > pq;
```

- Can use multiple priority queues to maintain the k-th element in a sorted list dynamically (see 10107, 501).

Map and Set

- `map` and `set` can be used if you want to access elements by a sortable "key".
- Use `set` if there are no associated values.
- Implemented as a balanced binary search tree: $O(\log n)$ each operation.
- An iterator can be used to go through all keys in smallest to largest order. Note that the elements in a map are pairs (key, value).

Bitset

- To keep track of a set of bits (or a subset of items), you can use `bitset`.
- You can set, clear, flip a particular bit or all bits.
- For only 32 or 64 bits, you can use an unsigned integer.
- Set: `S |= (1ULL << i)`
- Test: `S & (1ULL << i)`
- Clear: `S &= ~(1ULL << i)`
- Flip: `S ^= (1ULL << i)`
- Turn on all n bits: `S = (1 << n) - 1`

Union-Find/Disjoint Sets

- A data structures for us to keep track of **dynamic** equivalence relations.
- A set of n items are initially in their own sets.
- We can merge two elements x and y (and all elements equivalent to them).
- We can find the label of the set containing any element.
- `find(x) == find(y)` iff x and y are equivalent.
- You can only merge, not split: consider the operations backward (ECNA 2001, Galatic Breakup)

Union-Find/Disjoint Sets

- Amortized analysis: we consider the complexity of a group of M merge and find operations.
- Textbook version: $O(M \log n)$.
- My version: $O(M\alpha(n))$

Union-Find/Disjoint Sets Applications

- Graph connectivity and connected components (793, 10583, 11503).
- Kruskal algorithm for minimum spanning tree (later)

Fenwick Trees

- Special type of binary trees.
- Keeps track of a cumulative sum of an "array" of n non-negative integers.
- Initialization to 0: $O(n)$
- Initialization from arbitrary array: $O(n \log n)$
- Cumulative sum at index k: $O(\log k)$
- Read the original item at index k: $O(\log n)$
- Increment/decrement one entry (by any amount): $O(\log n)$
- Find an index with a given cumulative sum: $O(\log n)$.

Fenwick Trees

- Can be used to keep track of ranks of elements in a sorted list as items are inserted or deleted.