C++ STL and Algorithms Guide for Competitive Programming

1.1 std::vector

A vector is a dynamic array that provides random access and efficient insertion at the end.
Create: vector<int> v;
Insert: v.push_back(x); // O(1) amortized
Access: v[i] or v.at(i); // O(1)
Update: v[0] = 5;
Delete: v.pop_back(); or v.erase(it); // O(1) or O(n)
Traverse: for (int x : v) cout << x;

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    vector<int> v = {3, 1, 4};
    v.push_back(2);
    v[1] = 5;
    v.pop_back();
    for(int x : v) cout << x << " ";
    return 0;
}
```

1.2 std::string

std::string is a sequence of characters.
Concatenate: s += "abc"; or s.append("xyz");
Length: s.length();
Substring: s.substr(pos, len);
Erase: s.erase(pos, len);
Find: s.find("abc");

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    string s = "Hello";
    s += " World";
    cout << s << endl;
    s.replace(6, 5, "C++");
    cout << s << endl;
    return 0;
}
```

1.3 std::set & std::unordered_set

Set stores unique elements in sorted order.
unordered_set is hash-based and has average O(1) operations.
Insert: s.insert(x);
Erase: s.erase(x);
Search: s.find(x);

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    set<int> s = {3, 1, 4};
    unordered_set<int> us = {3, 1, 4};
    s.insert(2);
    us.insert(2);
    s.erase(1);
    for(int x : s) cout << x << " ";
    return 0;
}
```

1.4 std::map & std::unordered_map

Map stores key-value pairs sorted by key.
unordered_map uses hashing for average O(1) time.
Access/Insert: mp[key] = value;
Erase: mp.erase(key);
Find: mp.find(key);

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    map<int, string> mp;
    mp[2] = "two";
    mp[5] = "five";
    mp[2] = "dos";
    for(auto &[k,v] : mp) cout << k << ": " << v << endl;
    return 0;
}
```

1.5 std::stack

LIFO structure. Top: s.top(); Push: s.push(x); Pop: s.pop();

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    stack<int> st;
    st.push(10);
    st.push(20);
    cout << st.top() << endl;
    st.pop();
    cout << st.top() << endl;
    return 0;
}
```

1.6 std::queue

FIFO structure. Front: q.front(); Back: q.back(); Push: q.push(x); Pop: q.pop();

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    queue<int> q;
    q.push(1); q.push(2); q.push(3);
    q.pop();
    cout << q.front() << endl;
    return 0;
}
```

1.7 std::priority_queue

A heap by default (max-heap). For min-heap, use greater<int> as comparator.

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    priority_queue<int> maxpq;
    maxpq.push(5); maxpq.push(1); maxpq.push(10);
    cout << maxpq.top() << endl; maxpq.pop();

    priority_queue<int, vector<int>, greater<int>> minpq;
    minpq.push(5); minpq.push(1); minpq.push(10);
    cout << minpq.top() << endl;
    return 0;
}
```

2.1 sort, binary_search, lower_bound

sort(v.begin(), v.end()); // O(n log n)
binary_search(v.begin(), v.end(), x); // O(log n)
lower_bound(v.begin(), v.end(), x); // first >= x
upper_bound(v.begin(), v.end(), x); // first > x

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    vector<int> v = {3, 1, 4, 2};
    sort(v.begin(), v.end());
    cout << binary_search(v.begin(), v.end(), 3) << endl;
    auto it = lower_bound(v.begin(), v.end(), 2);
    cout << *it << endl;
    return 0;
}
```

3.1 Graph using vector<vector<int>>

Adjacency list representation of graph.
Each node has a list of connected nodes.
Add edge: adj[u].push_back(v);

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int V = 4;
    vector<vector<int>> adj(V);
    adj[0].push_back(1);
    adj[1].push_back(2);
    for (int i = 0; i < V; i++) {
        cout << i << ": ";
        for (int j : adj[i]) cout << j << " ";
        cout << endl;
    }
    return 0;
}
```