Contents

1 Basic

1.1 Pragma

```cpp
#pragma GCC optimize("Ofast,no-stack-protector")
#pragma GCC optimize("no-math-errno,unroll-loops")
#pragma GCC target("sse,sse2,sse3,ssse3,sse4")
#pragma GCC target("popcnt,abm,mmx,avx,tune=native")
```

2 Data Structure

2.1 Black Magic

```cpp
template<typename T>
using pbds_tree = tree<T, null_type, less<T>,
   rb_tree_tag, tree_order_statistics_node_update>;
// find_by_order: like array accessing, order_of_key
```

2.2 Linear Basis

```cpp
template<int BITS>
struct linear_basis {
  array<uint64_t, BITS> basis;
  linear_basis() { basis.fill(0); }
  void add(uint64_t x) {
    for(int i = BITS - 1; i >= 0; i--) if((x >> i) & 1)
     {
      if(basis[i] == 0) {
        basis[i] = x;
        continue;
      }
      x ^= basis[i];
    }
  }
  bool valid(uint64_t x) {
    for(int i = BITS - 1; i >= 0; i--)
      if((x >> i) & 1) x ^= basis[i];
    return x == 0;
  }
  // max xor sum: xor sum of all basis
  // min xor sum: zero(if possible) or min_element
}; // not tested
```

3 Graph

3.1 Bridge CC

```cpp
namespace bridge_cc {
  vector<int> tim, low;
  stack<int, vector<int>> st;
  int t, bcc_id;
  void dfs(int u, int p, const vector<vector<pair<int,
    int>>> &edge, vector<int> &pa) {
    tim[u] = low[u] = t++;
    st.push(u);
    for(const auto &[v, id] : edge[u]) {
      if(id == p)
        continue;
      if(tim[v])
        low[u] = min(low[u], tim[v]);
      else {
        dfs(v, id, edge, pa);
        if(low[v] > tim[u]) {
          int x;
          do {
            pa[x = st.top()] = bcc_id;
            st.pop();
          } while(x != v);
          bcc_id++;
        }
        else
          low[u] = min(low[u], low[v]);
      }
    }
  }
  vector<int> solve(const vector<vector<pair<int, int
    >>> &edge) { // (to, id)
    int n = edge.size();
    tim.resize(n);
    low.resize(n);
    t = bcc_id = 1;
    vector<int> pa(n);

    for(int i = 0; i < n; i++) {
      if(!tim[i]) {
        dfs(i, -1, edge, pa);
        while(!st.empty()) {
          pa[st.top()] = bcc_id;
          st.pop();
        }
        bcc_id++;
      }
    }
    return pa;
  } // return bcc id(start from 1)
};
```