Hash

\\ Pre Code //

```cpp
const int N = 1e5 + 5, P1 = 31, P2 = 37, M = 1e9 + 7;
int pw1[N], pw2[N], inv1[N], inv2[N];
int mul(int a, int b) {
    a = ((a % M) + M) % M;
    b = ((b % M) + M) % M;
    return (a * 1LL * b) % M;
}
int add(int a, int b) {
    a = ((a % M) + M) % M;
    b = ((b % M) + M) % M;
    return (a + b) % M;
}
int fastPower(int base, int power) {
    if (!power) return 1;
    int ret = fastPower(base, power >> 1);
    ret = mul(ret, ret);
    if (power % 2) ret = mul(ret, base);
    return ret;
}
int pre = []() {
    pw1[0] = inv1[0] = pw2[0] = inv2[0] = 1;
    int mulInv1 = fastPower(P1, M - 2);
    int mulInv2 = fastPower(P2, M - 2);
    for (int i = 1; i < N; i++) {
        pw1[i] = mul(pw1[i - 1], P1);
        pw2[i] = mul(pw2[i - 1], P2);
        inv1[i] = mul(inv1[i - 1], mulInv1);
        inv2[i] = mul(inv2[i - 1], mulInv2);
    }
    return 0;
}();
```

\\ Base Code //

```cpp
struct Hash {
    vector<pair<int, int>> prefixHash;
    Hash(string s) {
        prefixHash = vector<pair<int, int>> (s.size(), {0, 0});
        for (int i = 0; i < s.size(); i++) {
            prefixHash[i].F = mul(s[i] - 'a' + 1, pw1[i]);
            prefixHash[i].S = mul(s[i] - 'a' + 1, pw2[i]);
            if (i) prefixHash[i] = {
                        add(prefixHash[i].F, prefixHash[i - 1].F),
                        add(prefixHash[i].S, prefixHash[i - 1].S)
                };
        }
    }
    pair<int, int> getHashVal() {
        return prefixHash.back();
    }
    pair<int, int> getRangeHashVal(int l, int r) { // 0-based
        if(r < l) return {0, 0};

        return {
                mul(add(prefixHash[r].F, -(l ? prefixHash[l - 1].F : 0)), inv1[l]),
                mul(add(prefixHash[r].S, -(l ? prefixHash[l - 1].S : 0)), inv2[l])
        };
    }
    pair<int, int> getHashValWithoutRange(int l, int r){ // 0-based
        if(r < l) return getHashVal();

        auto rem = getRangeHashVal(l, r);
        auto hash = getHashVal();

        return{
            add(hash.F, -mul(rem.F, fastPower(P1, l))),
            add(hash.S, -mul(rem.S, fastPower(P2, l)))
        };
    }
    pair<int, int> addRangeFromMeToAnotherHash(pair<int, int> secondHash, int l, int r){ // 0-based
        if(r < l) return secondHash;

        auto over = getRangeHashVal(l, r);

        return{
            add(secondHash.F, mul(over.F, fastPower(P1, l))),
            add(secondHash.S, mul(over.S, fastPower(P2, l)))
        };
    }
};
```

Suffix Array

\\ Start //

```
class SuffixArray {
    // code
};
```

\\ Base Code(private) //

```
private:
    string s;
    int n;
    vector<int> p;  // Suffix array (sorted suffixes)
    vector<int> c;  // Equivalence classes
    vector<int> lcp; // Longest Common Prefix array

    // Counting sort for suffix array construction
    void countSort(vector<int> &p, vector<int> &c) {
        vector<int> cnt(n, 0);
        for (auto x : c) cnt[x]++;

        vector<int> p_new(n);
        vector<int> pos(n);
        pos[0] = 0;
        for (int i = 1; i < n; i++)
            pos[i] = pos[i-1] + cnt[i-1];

        for (auto x : p) {
            int i = c[x];
            p_new[pos[i]] = x;
            pos[i]++;
        }
        p = p_new;
    }

    // Kasai's algorithm for LCP array
    void buildLCP() {
        lcp.resize(n);
        int k = 0;
        for (int i = 0; i < n-1; i++) {
            int pi = c[i];
            int j = p[pi-1];

            while (s[i+k] == s[j+k]) k++;
            lcp[pi] = k;
            k = max<ll>(k-1, 0);
        }
    }

```

\\ Base Code(public) //

```
public:
    explicit SuffixArray(string &str, char terminal = '$') : s(str) {
        s += terminal;
        n = s.size();
        p.resize(n);
        c.resize(n);

        // Initial sorting (k=1)
        {
            vector<pair<char, int>> a(n);
            for (int i = 0; i < n; i++)
                a[i] = {s[i], i};

            sort(a.begin(), a.end());
            for (int i = 0; i < n; i++)
                p[i] = a[i].second;

            c[p[0]] = 0;
            for (int i = 1; i < n; i++)
                c[p[i]] = (a[i].first == a[i-1].first) ? c[p[i-1]] : c[p[i-1]]+1;
        }

        // Iterative doubling (k=2,4,8,...)
        int k = 0;
        while ((1 << k) < n) {
            // Shift positions
            for (int i = 0; i < n; i++)
                p[i] = (p[i] - (1 << k) + n) % n;

            countSort(p, c);

            vector<int> c_new(n);
            c_new[p[0]] = 0;
            for (int i = 1; i < n; i++) {
                pair<int, int> prev = {c[p[i-1]], c[(p[i-1] + (1 << k)) % n]};
                pair<int, int> curr = {c[p[i]], c[(p[i] + (1 << k)) % n]};
                c_new[p[i]] = (curr == prev) ? c_new[p[i-1]] : c_new[p[i-1]]+1;
            }
            c = c_new;
            k++;
        }

        buildLCP();
    }
```

\\ Accessors //

```cpp
// Accessors
    const vector<int>& getSuffixArray() const { return p; }
    const vector<int>& getLCP() const { return lcp; }
```

\\ Utility function //

Print(Suffixes, LCP, Order, Strings, C)

```cpp
void printSuffixes() {
    for (int i = 0; i < n; i++)
        cout << p[i] << ": " << s.substr(p[i]) << endl;
}

void printLCP() {
    for (int i = 1; i < n; i++)
        cout << lcp[i] << " ";
    cout << endl;
}
void printOrder() {
    for (int i = 0; i < n; ++i) {
        cout << p[i] << " \n"[i == n - 1];
    }
}

void printStrings() {
    for (int i = 0; i < n; ++i) {
        cout << p[i] << ' ' << s.substr(p[i]) << '\n';
    }
}

void printC(){
    for(auto &x: this->c){
        cout << x << ' ';
    }
    cout << endl;
}
```

\\ Applications //

Count Distinct Substrings, Longest Common Substring

```cpp
// Applications
    int countDistinctSubstrings() {
        int total = 0;
        for (int i = 1; i < n; i++)
            total += (n - p[i] - 1) - lcp[i];
        return total;
    }

    string longestCommonSubstring(string &s1, string &s2) {
        string combined = s1 + "$" + s2;
        SuffixArray sa(combined, '#');
        const vector<int>& sa_vec = sa.getSuffixArray();
        const vector<int>& lcp_vec = sa.getLCP();

        int max_len = 0, pos = -1;
        int s1_len = s1.size();

        for (int i = 1; i < sa_vec.size(); i++) {
            if ((sa_vec[i-1] < s1_len && sa_vec[i] > s1_len) ||
                (sa_vec[i] < s1_len && sa_vec[i-1] > s1_len)) {
                if (lcp_vec[i] > max_len) {
                    max_len = lcp_vec[i];
                    pos = min(sa_vec[i-1], sa_vec[i]);
                }
            }
        }

        return (max_len > 0) ? combined.substr(pos, max_len) : "";
    }
```

Z Algorithm

```cpp
vector<int> z_algo(string s) {
    vector<int> z(s.size());
    for(int i = 1, l = 0, r = 0; i < s.size(); i++) {
        if(i < r) z[i] = min(r - i, z[i - l]);
        while(i + z[i] < s.size() && s[z[i]] == s[z[i] + i]) z[i]++;
        if(i + z[i] > r) r = i + z[i], l = i;
    }
    return z;
}
```

Suffix Automaton

\\ Base code //

```cpp
struct suffix_automaton {
    struct state : map<int, int> {
        int fail = -1, len{}, cnt = 1;
        bool ed = false;
    };
    int lst, n{};
    vector<state> tr;
    explicit suffix_automaton(const string& s = ""s) : tr(1), lst(0) {
        tr.reserve(s.size() * 2);
        for(auto i : s) add(i);
    }
    void add(int c) {
        int x = (int)(tr.size());
        tr.emplace_back(), n++;
        tr[x].len = tr[lst].len + 1;
        int p = lst, q;
        while(~p && (q = tr[p].emplace(c, x).first->second) == x) p = tr[p].fail;
        if(p == -1) tr[x].fail = 0;
        else {
            if(tr[p].len + 1 == tr[q].len) tr[x].fail = q;
            else {
                int y = (int)(tr.size()); tr.emplace_back(tr[q]);
                tr[y].cnt = 0, tr[y].len = tr[p].len + 1;
                while(~p && tr[p][c] == q) tr[p][c] = y, p = tr[p].fail;
                tr[x].fail = tr[q].fail = y;
            }
        }
        lst = x;
    }
    void init() { // to build cnt, end
        for(int p = lst; ~p; tr[p].ed = true, p = tr[p].fail);
        vector b(n + 1, vector(0, 0));
        for(int i = 0; i < tr.size(); ++i) b[tr[i].len].push_back(i);
        for(int l = n; l >= 1; --l)
            for(int u : b[l])
                tr[tr[u].fail].cnt += tr[u].cnt;
    }

    long long distinct_substrings() {
        long long res = 0;
        for(int i = 1; i < tr.size(); ++i)
            res += tr[i].len - tr[tr[i].fail].len;
        return res;
    }
};
```

\\ Applications //

Longest Common Substring

```cpp
string longest_common_substring(const string& s, const string& t) {
    suffix_automaton sa(s);
    int max_len = 0, pos = 0;
    int current_len = 0, current_state = 0;
    for (int i = 0; i < t.size(); ++i) {
        while (current_state != 0 && sa.tr[current_state].find(t[i]) == sa.tr[current_state].end()) {
            current_state = sa.tr[current_state].fail;
            current_len = sa.tr[current_state].len;
        }
        if (sa.tr[current_state].find(t[i]) != sa.tr[current_state].end()) {
            current_state = sa.tr[current_state][t[i]];
            current_len++;
            if (current_len > max_len) {
                max_len = current_len;
                pos = i - max_len + 1;
            }
        }
    }
    return t.substr(pos, max_len);
}
```

Longest Palindromic Substring

```cpp
string longest_palindromic_substring(const string& s) {
    string t = s + '#' + string(s.rbegin(), s.rend());
    suffix_automaton sa(t);
    return longest_common_substring(s, string(s.rbegin(), s.rend()));
}
```

Find Occurrences

```cpp
vector<int> find_occurrences(const string& s, const string& p) {
    suffix_automaton sa(s);
    int state = 0;
    for (char c : p) {
        if (sa.tr[state].find(c) == sa.tr[state].end())
            return {};
        state = sa.tr[state][c];
    }
    vector<int> occurrences;
    queue<int> q;
    q.push(state);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        if (sa.tr[u].ed) occurrences.push_back(sa.tr[u].len - p.size());
        for (auto [c, v] : sa.tr[u]) q.push(v);
    }
    return occurrences;
}
```

Lex Smallest Substring

```cpp
string lex_smallest_substring(const string& s, int k) {
    suffix_automaton sa(s);
    string res;
    int state = 0;
    for (int i = 0; i < k; ++i) {
        for (auto [c, v] : sa.tr[state]) {
            res += c;
            state = v;
            break;
        }
    }
    return res;
}
```

Longest Repeated Substring

```cpp
string longest_repeated_substring(const string& s) {
    suffix_automaton sa(s);
    sa.init();
    int max_len = 0, state = 0;
    for (int i = 1; i < sa.tr.size(); ++i) {
        if (sa.tr[i].cnt > 1 && sa.tr[i].len > max_len) {
            max_len = sa.tr[i].len;
            state = i;
        }
    }
    string res;
    while (state != 0) {
        for (auto [c, v] : sa.tr[state]) {
            res += c;
            state = sa.tr[state].fail;
            break;
        }
    }
    reverse(res.begin(), res.end());
    return res;
}
```

Largest Lex Substring

```cpp
vector<int> z_algo(string s) {
    vector<int> z(s.size());
    for(int i = 1, l = 0, r = 0; i < s.size(); i++) {
        if(i < r) z[i] = min(r - i, z[i - l]);
        while(i + z[i] < s.size() && s[z[i]] == s[z[i] + i]) z[i]++;
        if(i + z[i] > r) r = i + z[i], l = i;
    }
    return z;
}
```

Manacher

```cpp
auto manacher(const string &t) {
    string s = "%#";
    s.reserve(t.size() * 2 + 3);
    for(char c : t) s += c + "#"s;
    s += '$';
    // t = aabaacaabaa → s = %#a#a#b#a#a#c#a#a#b#a#a#$

    vector<int> res(s.size());
    for(int i = 1, l = 1, r = 1; i < s.size(); i++) {
        res[i] = max(0, min(r - i, res[l + r - i]));
        while(s[i + res[i]] == s[i - res[i]]) res[i]++;
        if(i + res[i] > r) {
            l = i - res[i];
            r = i + res[i];
        }
    }
    for(auto &i : res) i--;
    return vector(res.begin() + 2, res.end() - 2); // a#a#b#a#a#c#a#a#b#a#a
    // get max odd len = res[2 * i]; aba → i = b
    // get max even len = res[2 * i + 1]; abba → i = first b
}
```

Aho Corasick

```cpp
struct corasick {
    struct node {
        array<int, 26> nxt{}, go{};
        vector<int> idx; // all string's indexes have any suffix
        int p, link;
        char ch;
        int cnt = 0;
        explicit node(int p = -1, char ch = '?') : p(p), ch(ch), link(-1) {
            nxt.fill(-1);
            go.fill(-1);
        }
    };
    vector<node> tr;
    explicit corasick(vector<string> &v) : tr(1) {
        for(int i = 0; i < v.size(); i++) {
            int x = 0;
            for(char c : v[i]) {
                if(tr[x].nxt[c - 'a'] == -1) {
                    tr[x].nxt[c - 'a'] = (int)(tr.size());
                    tr.emplace_back(x, c);
                }
                x = tr[x].nxt[c - 'a'];
            }
            tr[x].idx.push_back(i);
        }
        for(int i = 0; i < tr.size(); i++) {
            mxSuffix(i);
        }
    }
    int plus(int x, char c) {
        if(tr[x].go[c - 'a'] == -1) {
            if(tr[x].nxt[c - 'a'] != -1)
                tr[x].go[c - 'a'] = tr[x].nxt[c - 'a'];
            else
                tr[x].go[c - 'a'] = x == 0? 0: plus(mxSuffix(x), c);
        }
        return tr[x].go[c - 'a'];
    }
    int mxSuffix(int x) {
        if(tr[x].link == -1) {
            if(!x || !tr[x].p)
                tr[x].link = 0;
            else
                tr[x].link = plus(mxSuffix(tr[x].p), tr[x].ch);

            mxSuffix(tr[x].link);
            tr[x].idx.reserve(tr[x].idx.size() + tr[tr[x].link].idx.size());
            for(int y : tr[tr[x].link].idx)
                tr[x].idx.push_back(y);
        }
        return tr[x].link;
    }
    vector<int> match(const string &text, int n) {
        vector<int> pattern_count(n);
        int state = 0;
        for(char c : text) {
            state = plus(state, c);
            tr[state].cnt++;
        }

        vector<int> order(tr.size());
        iota(order.begin(), order.end(), 0);
        sort(order.begin(), order.end(), [&](int a, int b) {
            return depth(a) > depth(b);
        });

        for(int i : order) {
            if(tr[i].link != -1)
                tr[tr[i].link].cnt += tr[i].cnt;
        }

        for(int i = 0; i < tr.size(); i++) {
            for(int pat : tr[i].idx) {
                pattern_count[pat] += tr[i].cnt;
            }
        }
        return pattern_count;
    }

    int depth(int x) {
        int d = 0;
        while(x != 0) {
            x = tr[x].p;
            ++d;
        }
        return d;
    }
};
```