

Hash

\\ Pre Code //

```
1  const int N = 1e5 + 5, P1 = 31, P2 = 37, M = 1e9 + 7;
2  int pw1[N], pw2[N], inv1[N], inv2[N];
3  int mul(int a, int b) {
4      a = ((a % M) + M) % M;
5      b = ((b % M) + M) % M;
6      return (a * 1LL * b) % M;
7  }
8  int add(int a, int b) {
9      a = ((a % M) + M) % M;
10     b = ((b % M) + M) % M;
11     return (a + b) % M;
12 }
13 int fastPower(int base, int power) {
14     if (!power) return 1;
15     int ret = fastPower(base, power >> 1);
16     ret = mul(ret, ret);
17     if (power % 2) ret = mul(ret, base);
18     return ret;
19 }
20 int pre = []() {
21     pw1[0] = inv1[0] = pw2[0] = inv2[0] = 1;
22     int mulInv1 = fastPower(P1, M - 2);
23     int mulInv2 = fastPower(P2, M - 2);
24     for (int i = 1; i < N; i++) {
25         pw1[i] = mul(pw1[i - 1], P1);
26         pw2[i] = mul(pw2[i - 1], P2);
27         inv1[i] = mul(inv1[i - 1], mulInv1);
28         inv2[i] = mul(inv2[i - 1], mulInv2);
29     }
30     return 0;
31 }();
```

\\ Base Code //

```
1  struct Hash {
2      vector<pair<int, int>> prefixHash;
3      Hash(string s) {
4          prefixHash = vector<pair<int, int>> (s.size(), {0, 0});
5          for (int i = 0; i < s.size(); i++) {
6              prefixHash[i].F = mul(s[i] - 'a' + 1, pw1[i]);
7              prefixHash[i].S = mul(s[i] - 'a' + 1, pw2[i]);
8              if (i) prefixHash[i] = {
9                  add(prefixHash[i].F, prefixHash[i - 1].F),
10                  add(prefixHash[i].S, prefixHash[i - 1].S)
11              };
12          }
13      }
14      pair<int, int> getHashVal() {
15          return prefixHash.back();
16      }
17      pair<int, int> getRangeHashVal(int l, int r) { // 0-based
18          if (r < l) return {0, 0};
19
20          return {
21              mul(add(prefixHash[r].F, -(l ? prefixHash[l - 1].F : 0)), inv1[l]),
22              mul(add(prefixHash[r].S, -(l ? prefixHash[l - 1].S : 0)), inv2[l])
23          };
24      }
25      pair<int, int> getHashValWithoutRange(int l, int r) { // 0-based
26          if (r < l) return getHashVal();
27
28          auto rem = getRangeHashVal(l, r);
29          auto hash = getHashVal();
30
31          return {
32              add(hash.F, -mul(rem.F, fastPower(P1, l))),
33              add(hash.S, -mul(rem.S, fastPower(P2, l)))
34          };
35      }
36      pair<int, int> addRangeFromMeToAnotherHash(pair<int, int> secondHash, int l, int r) { // 0-based
37          if (r < l) return secondHash;
38
39          auto over = getRangeHashVal(l, r);
40
41          return {
42              add(secondHash.F, mul(over.F, fastPower(P1, l))),
43              add(secondHash.S, mul(over.S, fastPower(P2, l)))
44          };
45      }
46  };
```

Suffix Array

// Start //

```
1 class SuffixArray {
2     // code
3 };
```

// Base Code(private) //

```
1 private:
2     string s;
3     int n;
4     vector<int> p; // Suffix array (sorted suffixes)
5     vector<int> c; // Equivalence classes
6     vector<int> lcp; // Longest Common Prefix array
7
8     // Counting sort for suffix array construction
9     void countSort(vector<int> &p, vector<int> &c) {
10         vector<int> cnt(n, 0);
11         for (auto x : c) cnt[x]++;
12
13         vector<int> p_new(n);
14         vector<int> pos(n);
15         pos[0] = 0;
16         for (int i = 1; i < n; i++)
17             pos[i] = pos[i-1] + cnt[i-1];
18
19         for (auto x : p) {
20             int i = c[x];
21             p_new[pos[i]] = x;
22             pos[i]++;
23         }
24         p = p_new;
25     }
26
27     // Kasai's algorithm for LCP array
28     void buildLCP() {
29         lcp.resize(n);
30         int k = 0;
31         for (int i = 0; i < n-1; i++) {
32             int pi = c[i];
33             int j = p[pi-1];
34
35             while (s[i+k] == s[j+k]) k++;
36             lcp[pi] = k;
37             k = max<ll>(k-1, 0);
38         }
39     }
40 }
```

// Base Code(public) //

```
1 public:
2     explicit SuffixArray(string &str, char terminal = '$') : s(str) {
3         s += terminal;
4         n = s.size();
5         p.resize(n);
6         c.resize(n);
7
8         // Initial sorting (k=1)
9         {
10             vector<pair<char, int>> a(n);
11             for (int i = 0; i < n; i++)
12                 a[i] = {s[i], i};
13
14             sort(a.begin(), a.end());
15             for (int i = 0; i < n; i++)
16                 p[i] = a[i].second;
17
18             c[p[0]] = 0;
19             for (int i = 1; i < n; i++)
20                 c[p[i]] = (a[i].first == a[i-1].first) ? c[p[i-1]] : c[p[i-1]]+1;
21         }
22
23         // Iterative doubling (k=2,4,8, ...)
24         int k = 0;
25         while ((1 << k) < n) {
26             // Shift positions
27             for (int i = 0; i < n; i++)
28                 p[i] = (p[i] - (1 << k) + n) % n;
29
30             countSort(p, c);
31
32             vector<int> c_new(n);
33             c_new[p[0]] = 0;
34             for (int i = 1; i < n; i++) {
35                 pair<int, int> prev = {c[p[i-1]], c[(p[i-1] + (1 << k)) % n]};
36                 pair<int, int> curr = {c[p[i]], c[(p[i] + (1 << k)) % n]};
37                 c_new[p[i]] = (curr == prev) ? c_new[p[i-1]] : c_new[p[i-1]]+1;
38             }
39             c = c_new;
40             k++;
41         }
42
43         buildLCP();
44     }
```

\\ Accessors //

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

\\ Utility function //

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

```
1 void printSuffixes() {
2     for (int i = 0; i < n; i++)
3         cout << p[i] << ": " << s.substr(p[i]) << endl;
4 }
5
6 void printLCP() {
7     for (int i = 1; i < n; i++)
8         cout << lcp[i] << " ";
9     cout << endl;
10 }
11 void printOrder() {
12     for (int i = 0; i < n; ++i) {
13         cout << p[i] << " \n"[i == n - 1];
14     }
15 }
16
17 void printStrings() {
18     for (int i = 0; i < n; ++i) {
19         cout << p[i] << ' ' << s.substr(p[i]) << '\n';
20     }
21 }
22
23 void printC(){
24     for(auto &x: this->c){
25         cout << x << ' ';
26     }
27     cout << endl;
28 }
```

\\ Applications //

Count Distinct Substrings, Longest Common Substring

```
1 // Applications
2     int countDistinctSubstrings() {
3         int total = 0;
4         for (int i = 1; i < n; i++)
5             total += (n - p[i] - 1) - lcp[i];
6         return total;
7     }
8
9     string longestCommonSubstring(string &s1, string &s2) {
10         string combined = s1 + "$" + s2;
11         SuffixArray sa(combined, '#');
12         const vector<int>& sa_vec = sa.getSuffixArray();
13         const vector<int>& lcp_vec = sa.getLCP();
14
15         int max_len = 0, pos = -1;
16         int s1_len = s1.size();
17
18         for (int i = 1; i < sa_vec.size(); i++) {
19             if ((sa_vec[i-1] < s1_len && sa_vec[i] > s1_len) ||
20                 (sa_vec[i] < s1_len && sa_vec[i-1] > s1_len)) {
21                 if (lcp_vec[i] > max_len) {
22                     max_len = lcp_vec[i];
23                     pos = min(sa_vec[i-1], sa_vec[i]);
24                 }
25             }
26         }
27
28         return (max_len > 0) ? combined.substr(pos, max_len) : "";
29     }
```

Z Algorithm

```
1 vector<int> z_algo(string s) {
2     vector<int> z(s.size());
3     for(int i = 1, l = 0, r = 0; i < s.size(); i++) {
4         if(i < r) z[i] = min(r - i, z[i - l]);
5         while(i + z[i] < s.size() && s[z[i]] == s[z[i] + i]) z[i]++;
6         if(i + z[i] > r) r = i + z[i], l = i;
7     }
8     return z;
9 }
```


Suffix Automaton

\\ Base code //

```
1 struct suffix_automaton {
2     struct state : map<int, int> {
3         int fail = -1, len{}, cnt = 1;
4         bool ed = false;
5     };
6     int lst, n{};
7     vector<state> tr;
8     explicit suffix_automaton(const string& s = ""s) : tr(1), lst(0) {
9         tr.reserve(s.size() * 2);
10        for(auto i : s) add(i);
11    }
12    void add(int c) {
13        int x = (int)(tr.size());
14        tr.emplace_back(), n++;
15        tr[x].len = tr[lst].len + 1;
16        int p = lst, q;
17        while(~p && (q = tr[p].emplace(c, x).first->second) == x) p = tr[p].fail;
18        if(p == -1) tr[x].fail = 0;
19        else {
20            if(tr[p].len + 1 == tr[q].len) tr[x].fail = q;
21            else {
22                int y = (int)(tr.size()); tr.emplace_back(tr[q]);
23                tr[y].cnt = 0, tr[y].len = tr[p].len + 1;
24                while(~p && tr[p][c] == q) tr[p][c] = y, p = tr[p].fail;
25                tr[x].fail = tr[q].fail = y;
26            }
27        }
28        lst = x;
29    }
30    void init() { // to build cnt, end
31        for(int p = lst; ~p; p = tr[p].fail)
32            vector b(n + 1, vector(0, 0));
33        for(int i = 0; i < tr.size(); ++i) b[tr[i].len].push_back(i);
34        for(int l = n; l >= 1; --l)
35            for(int u : b[l])
36                tr[tr[u].fail].cnt += tr[u].cnt;
37    }
38
39    long long distinct_substrings() {
40        long long res = 0;
41        for(int i = 1; i < tr.size(); ++i)
42            res += tr[i].len - tr[tr[i].fail].len;
43        return res;
44    }
45 };
```

\\ Applications //

Longest Common Substring

```
1 string longest_common_substring(const string& s, const string& t) {
2     suffix_automaton sa(s);
3     int max_len = 0, pos = 0;
4     int current_len = 0, current_state = 0;
5     for (int i = 0; i < t.size(); ++i) {
6         while (current_state != 0 && sa.tr[current_state].find(t[i]) == sa.tr[current_state].end()) {
7             current_state = sa.tr[current_state].fail;
8             current_len = sa.tr[current_state].len;
9         }
10        if (sa.tr[current_state].find(t[i]) != sa.tr[current_state].end()) {
11            current_state = sa.tr[current_state][t[i]];
12            current_len++;
13            if (current_len > max_len) {
14                max_len = current_len;
15                pos = i - max_len + 1;
16            }
17        }
18    }
19    return t.substr(pos, max_len);
20 }
```

Longest Palindromic Substring

```
1 string longest_palindromic_substring(const string& s) {
2     string t = s + '#' + string(s.rbegin(), s.rend());
3     suffix_automaton sa(t);
4     return longest_common_substring(s, string(s.rbegin(), s.rend()));
5 }
```

Find Occurrences

```
1 vector<int> find_occurrences(const string& s, const string& p) {
2     suffix_automaton sa(s);
3     int state = 0;
4     for (char c : p) {
5         if (sa.tr[state].find(c) == sa.tr[state].end())
6             return {};
7         state = sa.tr[state][c];
8     }
9     vector<int> occurrences;
10    queue<int> q;
11    q.push(state);
12    while (!q.empty()) {
13        int u = q.front(); q.pop();
14        if (sa.tr[u].ed) occurrences.push_back(sa.tr[u].len - p.size());
15        for (auto [c, v] : sa.tr[u]) q.push(v);
16    }
17    return occurrences;
18 }
```

Lex Smallest Substring

```
1 string lex_smallest_substring(const string& s, int k) {
2     suffix_automaton sa(s);
3     string res;
4     int state = 0;
5     for (int i = 0; i < k; ++i) {
6         for (auto [c, v] : sa.tr[state]) {
7             res += c;
8             state = v;
9             break;
10        }
11    }
12    return res;
13 }
```

Longest Repeated Substring

```
1 string longest_repeated_substring(const string& s) {
2     suffix_automaton sa(s);
3     sa.init();
4     int max_len = 0, state = 0;
5     for (int i = 1; i < sa.tr.size(); ++i) {
6         if (sa.tr[i].cnt > 1 && sa.tr[i].len > max_len) {
7             max_len = sa.tr[i].len;
8             state = i;
9         }
10    }
11    string res;
12    while (state != 0) {
13        for (auto [c, v] : sa.tr[state]) {
14            res += c;
15            state = sa.tr[state].fail;
16            break;
17        }
18    }
19    reverse(res.begin(), res.end());
20    return res;
21 }
```

Largest Lex Substring

```
1 vector<int> z_algo(string s) {
2     vector<int> z(s.size());
3     for(int i = 1, l = 0, r = 0; i < s.size(); i++) {
4         if(i < r) z[i] = min(r - i, z[i - l]);
5         while(i + z[i] < s.size() && s[z[i]] == s[z[i] + i]) z[i]++;
6         if(i + z[i] > r) r = i + z[i], l = i;
7     }
8     return z;
9 }
```

Manacher

```
1 auto manacher(const string &t) {
2     string s = "%#";
3     s.reserve(t.size() * 2 + 3);
4     for(char c : t) s += c + "#s";
5     s += '$';
6     // t = aabaacaabaa → s = %a#a#b#a#a#c#a#a#b#a#a#$
7
8     vector<int> res(s.size());
9     for(int i = 1, l = 1, r = 1; i < s.size(); i++) {
10        res[i] = max(0, min(r - i, res[l + r - i]));
11        while(s[i + res[i]] == s[i - res[i]]) res[i]++;
12        if(i + res[i] > r) {
13            l = i - res[i];
14            r = i + res[i];
15        }
16    }
17    for(auto &i : res) i--;
18    return vector(res.begin() + 2, res.end() - 2); // a#a#b#a#a#c#a#a#b#a#a
19    // get max odd len = res[2 * i]; aba → i = b
20    // get max even len = res[2 * i + 1]; abba → i = first b
21 }
```

Aho Corasick

```
1 struct corasick {
2     struct node {
3         array<int, 26> nxt{}, go{};
4         vector<int> idx; // all string's indexes have any suffix
5         int p, link;
6         char ch;
7         int cnt = 0;
8         explicit node(int p = -1, char ch = '?') : p(p), ch(ch), link(-1) {
9             nxt.fill(-1);
10            go.fill(-1);
11        }
12    };
13    vector<node> tr;
14    explicit corasick(vector<string> &v) : tr(1) {
15        for(int i = 0; i < v.size(); i++) {
16            int x = 0;
17            for(char c : v[i]) {
18                if(tr[x].nxt[c - 'a'] == -1) {
19                    tr[x].nxt[c - 'a'] = (int)(tr.size());
20                    tr.emplace_back(x, c);
21                }
22                x = tr[x].nxt[c - 'a'];
23            }
24            tr[x].idx.push_back(i);
25        }
26        for(int i = 0; i < tr.size(); i++) {
27            mxSuffix(i);
28        }
29    }
30    int plus(int x, char c) {
31        if(tr[x].go[c - 'a'] == -1) {
32            if(tr[x].nxt[c - 'a'] != -1)
33                tr[x].go[c - 'a'] = tr[x].nxt[c - 'a'];
34            else
35                tr[x].go[c - 'a'] = x == 0 ? 0 : plus(mxSuffix(x), c);
36        }
37        return tr[x].go[c - 'a'];
38    }
39    int mxSuffix(int x) {
40        if(tr[x].link == -1) {
41            if(!x || !tr[x].p)
42                tr[x].link = 0;
43            else
44                tr[x].link = plus(mxSuffix(tr[x].p), tr[x].ch);
45        }
46        mxSuffix(tr[x].link);
47        tr[x].idx.reserve(tr[x].idx.size() + tr[tr[x].link].idx.size());
48        for(int y : tr[tr[x].link].idx)
49            tr[x].idx.push_back(y);
50    }
51    return tr[x].link;
52    }
53    vector<int> match(const string &text, int n) {
54        vector<int> pattern_count(n);
55        int state = 0;
56        for(char c : text) {
57            state = plus(state, c);
58            tr[state].cnt++;
59        }
60
61        vector<int> order(tr.size());
62        iota(order.begin(), order.end(), 0);
63        sort(order.begin(), order.end(), [&](int a, int b) {
64            return depth(a) > depth(b);
65        });
66
67        for(int i : order) {
68            if(tr[i].link != -1)
69                tr[tr[i].link].cnt += tr[i].cnt;
70        }
71
72        for(int i = 0; i < tr.size(); i++) {
73            for(int pat : tr[i].idx) {
74                pattern_count[pat] += tr[i].cnt;
75            }
76        }
77        return pattern_count;
78    }
79
80    int depth(int x) {
81        int d = 0;
82        while(x != 0) {
83            x = tr[x].p;
84            ++d;
85        }
86        return d;
87    }
88    };
```