

1 Strings

1.1 KMP

```

1 vi prefix(string &S)
2 {
3     vector<int> p(S.size());
4     p[0] = 0;
5     for (int i = 1; i < S.size(); ++i)
6     {
7         p[i] = p[i - 1];
8         while (p[i] > 0 && S[p[i]] != S[i])
9             p[i] = p[p[i] - 1];
10        if (S[p[i]] == S[i])
11            p[i]++;
12    }
13    return p;
14 }
15
16 vi KMP(string &P, string &S)
17 {
18     vector<int> pi = prefix(P);
19     vi matches;
20     int n = S.length(), m = P.length();
21     int j = 0, ans = 0;
22     for (int i = 0; i < n; ++i)
23     {
24         while (j > 0 && S[i] != P[j])
25             j = pi[j - 1];
26         if (S[i] == P[j])
27             ++j;
28
29         if (j == P.length())
30         {
31             /* This is where KMP found a match
32              * we can calculate its position on S by using i - m + 1
33              * or we can simply count it
34              */
35             ans += 1; // count the number of matches
36             matches.pb(i - m + 1); // store the position of those
37                                 matches
38             // return; we can return on the first match if needed
39             // this must stay the same
40             j = pi[j - 1];
41         }
42     }
43     return matches; // can be modified to return number of matches or
44                     location
45 }

```

1.2 Rolling Hashing

```

2 const int MAXLEN = 1e6;
3
4 class rollingHashing {
5     static const ull base = 127;
6     static const vector<ull> primes;
7     static vector<vector<ull>> POW;
8
9     static ull add(ull x, ull y, int a) { return (x + y) % primes[a]; }
10    static ull mul(ull x, ull y, int a) { return (x * y) % primes[a]; }
11
12    static void init(int a) {
13        if (POW.size() <= a + 1) {
14            POW.pb(MAXLEN, 1);
15        }
16        repx(i, 1, MAXLEN) POW[a][i] = mul(POW[a][i], base, a);
17    }
18
19    static void init() { rep(i, primes.size()) init(i); }
20
21    vector<vector<ull>> h;
22    int len;
23    rollingHashing(string &s) {
24        len = s.size();
25        h.assign(primes.size(), vector<ull>(len, 0));
26        rep(a, primes.size()) {
27            h[a][0] = s[0] - 'a'; // Assuming alphabetic alphabet
28            repx(i, 1, len) h[a][i] = add(s[i] - 'a', mul(h[a][i - 1], base, a), a);
29        }
30    }
31
32    ull hash(int i, int j, int a) // Inclusive-Exclusive [i,i)?
33    {
34        if (i == 0)
35            return h[a][j - 1];
36        return add(h[a][j - 1], primes[a] - mul(h[a][i - 1], POW[a][j - i], a), a);
37    }
38
39    ull hash(int i, int j) // Supports at most two primes
40    {
41        return hash(i, j, 1) << 32 |
42            hash(i, j, 0); // Using that 1e18 < __LONG_LONG_MAX__
43    }
44
45    ull hash() { return hash(0, len); } // Also supports at most two
46    primes
47 };
48
49 const vector<ull> rollingHashing ::primes({(ull)1e9 + 7,
50                                             (ull)1e9 + 9}); // Add more
51                                             if needed

```

1.3 Trie

```

1  /* Implementation from: https://pastebin.com/fyqsH65k */
2
3  struct TrieNode {
4      int leaf; // number of words that end on a TrieNode (allows for
5                duplicate
6                // words)
7      int height; // height of a TrieNode, root starts at height = 1, can be
8                  changed
9                  // with the default value of constructor
10     // number of words that pass through this node,
11     // ask root node for this count to find the number of entries on the
12     whole
13     // Trie all nodes have 1 as they count the words than end on
14     themselves (ie
15     // leaf nodes count themselves)
16     int count;
17     TrieNode *parent; // pointer to parent TrieNode, used on erasing
18     entries
19     map<char, TrieNode *> child;
20     TrieNode(TrieNode *parent = NULL, int height = 1)
21         : parent(parent), leaf(0), height(height),
22         count(0), // change to -1 if leaf nodes are to have count 0
23         instead of 1
24         child() {}
25 };
26
27 /**
28  * Complexity: O(|key| * log(k))
29  */
30 TrieNode *trie_find(TrieNode *root, const string &str) {
31     TrieNode *pNode = root;
32     for (string::const_iterator key = str.begin(); key != str.end(); key
33         ++){
34         if (pNode->child.find(*key) == pNode->child.end())
35             return NULL;
36         pNode = pNode->child[*key];
37     }
38     return (pNode->leaf) ? pNode : NULL; // returns only whole word
39     // return pNode; // allows to search for a suffix
40 }
41
42 /**
43  * Complexity: O(|key| * log(k))
44  */
45 void trie_insert(TrieNode *root, const string &str) {
46     TrieNode *pNode = root;
47     root->count += 1;
48     for (string::const_iterator key = str.begin(); key != str.end(); key
49         ++){
50         if (pNode->child.find(*key) == pNode->child.end())
51             pNode->child[*key] = new TrieNode(pNode, pNode->height + 1);
52         pNode = pNode->child[*key];
53         pNode->count += 1;
54     }
55     pNode->leaf += 1;

```

```

48 }
49
50 /**
51  * Complexity: O(|key| * log(k))
52  */
53 void trie_erase(TrieNode *root, const string &str) {
54     TrieNode *pNode = root;
55     string::const_iterator key = str.begin();
56     for (; key != str.end(); key++) {
57         if (pNode->child.find(*key) == pNode->child.end())
58             return;
59         pNode = pNode->child[*key];
60     }
61     pNode->leaf -= 1;
62     pNode->count -= 1;
63     while (pNode->parent != NULL) {
64         if (pNode->child.size() > 0 || pNode->leaf)
65             break;
66         pNode = pNode->parent, key--;
67         pNode->child.erase(*key);
68         pNode->count -= 1;
69     }
70 }

```

1.4 Suffix Tree

```

1
2  struct Node {
3      // map<int,int> children;
4      vector<int> children;
5      int suffix_link;
6      int start;
7      int end;
8
9      Node(int start, int end) : start(start), end(end) {
10         children.resize(27, -1);
11         suffix_link = 0;
12     }
13     inline bool has_child(int i) {
14         // return children.find(i) != children.end();
15         return children[i] != -1;
16     }
17 };
18
19 struct SuffixTree {
20     int size;
21     int i;
22     vector<int> suffix_array;
23     vector<Node> tree;
24     inline int length(int index) {
25         if (tree[index].end == -1)
26             return i - tree[index].start + 1;
27         return tree[index].end - tree[index].start + 1;
28     }
29     // se puede usar string& s

```

```

30 SuffixTree(vector<int> &s) {
31     size = s.size();
32     tree.emplace_back(-1, -1);
33     int remaining_suffix = 0;
34     int active_node = 0;
35     int active_edge = -1;
36     int active_length = 0;
37     for (i = 0; i < size; ++i) {
38         int last_new = -1;
39         remaining_suffix++;
40         while (remaining_suffix > 0) {
41             if (active_length == 0)
42                 active_edge = i;
43             if (!tree[active_node].has_child(s[active_edge])) {
44                 tree[active_node].children[s[active_edge]] = tree.size();
45                 tree.emplace_back(i, -1);
46                 if (last_new != -1) {
47                     tree[last_new].suffix_link = active_node;
48                     last_new = -1;
49                 }
50             } else {
51                 int next = tree[active_node].children[s[active_edge]];
52                 if (active_length >= length(next)) {
53                     active_edge += length(next);
54                     active_length -= length(next);
55                     active_node = next;
56                     continue;
57                 }
58                 if (s[tree[next].start + active_length] == s[i]) {
59                     if (last_new != -1 and active_node != 0) {
60                         tree[last_new].suffix_link = active_node;
61                     }
62                     active_length++;
63                     break;
64                 }
65                 int split_end = tree[next].start + active_length - 1;
66                 int split = tree.size();
67                 tree.emplace_back(tree[next].start, split_end);
68                 tree[active_node].children[s[active_edge]] = split;
69                 int new_leaf = tree.size();
70                 tree.emplace_back(i, -1);
71                 tree[split].children[s[i]] = new_leaf;
72                 tree[next].start += active_length;
73                 tree[split].children[s[tree[next].start]] = next;
74                 if (last_new != -1) {
75                     tree[last_new].suffix_link = split;
76                 }
77                 last_new = split;
78             }
79             remaining_suffix--;
80             if (active_node == 0 and active_length > 0) {
81                 active_length--;
82                 active_edge = i - remaining_suffix + 1;
83             } else if (active_node != 0) {
84                 active_node = tree[active_node].suffix_link;

```

```

85     }
86 }
87 }
88 i = size - 1;
89 }
90 vector<int> lcp;
91 // last for lcp
92 void dfs(int node, int &index, int depth, int min_depth) {
93     if (tree[node].end == -1 and node != 0) {
94         suffix_array[index] = size - depth;
95         if (index != 0) {
96             lcp[index - 1] = min_depth;
97         }
98         index++;
99     }
100     for (auto it : tree[node].children) {
101         // if(i.second != -1){
102         //     dfs(i.second, index, depth + length(i.second));
103         //     min_depth = depth;
104         // }
105         if (it != -1) {
106             dfs(it, index, depth + length(it), min_depth);
107             min_depth = depth;
108         }
109     }
110 }
111 void build_suffix_array() {
112     suffix_array.resize(size, 0);
113     lcp.resize(size, 0);
114     int index = 0;
115     int depth = 0;
116     dfs(0, index, 0, 0);
117 }
118
119 // pensado para map<int,int>, pero puede modificarse para vector<int>
120 bool match(string &a, string &base) {
121     int active_node = 0;
122     int active_length = 0;
123     int active_char = -1;
124     for (int i = 0; i < a.size(); i++) {
125         if (active_length == 0) {
126             if (!tree[active_node].has_child(a[i]))
127                 return false;
128             active_char = a[i];
129             active_length++;
130             i++;
131             continue;
132         }
133         int next = tree[active_node].children[active_char];
134         if (active_length == length(next)) {
135             active_node = next;
136             active_length = 0;
137             active_char = -1;
138             continue;
139         }

```

```
140     if ((base)[tree[next].start + active_length] != a[i])
141         return false;
142     active_length++;
143     i++;
144 }
145 return true;
146 }
147 };
```