1 Strings

1.1 KMP

```cpp

vi prefix(string &S)
{
    vector<int> p(S.size());
    p[0] = 0;
    for (int i = 1; i < S.size(); ++i)
    {
        p[i] = p[i - 1];
        while (p[i] > 0 && S[p[i]] != S[i])
            p[i] = p[p[i] - 1];
        if (S[p[i]] == S[i])
            p[i]++;
    }
    return p;
}

vi KMP(string &P, string &S)
{
    vector<int> pi = prefix(P);
    vi matches;
    int n = S.length(), m = P.length();
    int j = 0, ans = 0;
    for (int i = 0; i < n; ++i)
    {
        while (j > 0 && S[i] != P[j])
            j = pi[j - 1];
        if (S[i] == P[j])
            ++j;

        if (j == P.length())
        {
            /* This is where KMP found a match
             * we can calculate its position on S by using i - m + 1
             * or we can simply count it
             */
            ans += 1; // count the number of matches
            matches.eb(i - m + 1); // store the position of those matches
            // return; we can return on the first match if needed
            // this must stay the same
            j = pi[j - 1];
        }
    }
    return matches; // can be modified to return number of matches or location
}
```

1.2 Rolling Hashing

```cpp

const int MAXLEN = 1e6;

class rollingHashing
{
    static const ull base = 127;
    static const vector<ull> primes;
    static vector<vector<ull>> POW;

    static ull add(ull x, ull y, int a) { return (x + y) % primes[a]; }
    static ull mul(ull x, ull y, int a) { return (x * y) % primes[a]; }

    static void init(int a)
    {
        if (POW.size() <= a + 1)
        {
            POW.eb(MAXLEN, 1);
        }
        repx(i, 1, MAXLEN) POW[a][i] = mul(POW[a][i], base, a);
    }

    static void init()
    {
        rep(i, primes.size()) init(i);
    }

    vector<vector<ull>> h;
    int len;
    rollingHashing(string &s)
    {
        len = s.size();
        h.assign(primes.size(), vector<ull>(len, 0));
        rep(a, primes.size())
        {
            h[a][0] = s[0] - 'a'; //Assuming alphabetic alphabet
            repx(i, 1, len) h[a][i] = add(s[i] - 'a', mul(h[a][i - 1], base, a), a);
        }
    }

    ull hash(int i, int j, int a) //Inclusive-Exclusive [i,i)?
    {
        if (i == 0)
            return h[a][j - 1];
        return add(h[a][j - 1], primes[a] - mul(h[a][i - 1], POW[a][j - i], a), a);
    }

    ull hash(int i, int j)//Supports at most two primes
    {
        return hash(i, j, 1) << 32 | hash(i, j, 0);//Using that 1e18< __LONG_LONG_MAX__
    }

    ull hash() { return hash(0, len); }//Also supports at most two primes
};

const vector<ull> rollingHashing ::primes({(ull)1e9 + 7, (ull)1e9 + 9}); //Add more if needed
```

1.3 Trie

```cpp

/* Implementation from: https://pastebin.com/fyqsH65k */
struct TrieNode
{
    int leaf; // number of words that end on a TrieNode (allows for duplicate words)
    int height; // height of a TrieNode, root starts at height = 1, can be changed with the default value of constructor
    // number of words that pass through this node,
    // ask root node for this count to find the number of entries on the whole Trie
    // all nodes have 1 as they count the words than end on themselves (ie leaf nodes count themselves)
    int count;
    TrieNode *parent; // pointer to parent TrieNode, used on erasing entries
    map<char, TrieNode *> child;
    TrieNode(TrieNode *parent = NULL, int height = 1):
        parent(parent),
        leaf(0),
        height(height),
        count(0), // change to -1 if leaf nodes are to have count 0 insead of 1
        child()
    {}
};

/**
 * Complexity: O(|key| * log(k))
 */
TrieNode *trie_find(TrieNode *root, const string &str)
{
    TrieNode *pNode = root;
    for (string::const_iterator key = str.begin(); key != str.end(); key++)
    {
        if (pNode->child.find(*key) == pNode->child.end())
            return NULL;
        pNode = pNode->child[*key];
    }
    return (pNode->leaf) ? pNode : NULL; // returns only whole word
    // return pNode; // allows to search for a suffix
}

/**
 * Complexity: O(|key| * log(k))
 */
void trie_insert(TrieNode *root, const string &str)
{
    TrieNode *pNode = root;
    root -> count += 1;
    for (string::const_iterator key = str.begin(); key != str.end(); key++)
    {
        if (pNode->child.find(*key) == pNode->child.end())
            pNode->child[*key] = new TrieNode(pNode, pNode->height + 1);
        pNode = pNode->child[*key];
        pNode -> count += 1;
    }
    pNode->leaf += 1;
}

/**
 * Complexity: O(|key| * log(k))
 */
void trie_erase(TrieNode *root, const string &str)
{
    TrieNode *pNode = root;
    string::const_iterator key = str.begin();
    for (; key != str.end(); key++)
    {
        if (pNode->child.find(*key) == pNode->child.end())
            return;
        pNode = pNode->child[*key];
    }
    pNode->leaf -= 1;
    pNode->count -= 1;
    while (pNode->parent != NULL)
    {
        if (pNode->child.size() > 0 || pNode->leaf)
            break;
        pNode = pNode->parent, key--;
        pNode->child.erase(*key);
        pNode->count -= 1;
    }
}
```

1.4 Suffix Tree

```cpp
using namespace std;

#define rep(i, n) for (int i = 0; i < n; ++i)
#define repx(i, x, n) for (int i = x; i < n; ++i)

typedef vector<int> vi;
typedef long long ll;

#define eb emplace_back

struct Node
{
    //map<int,int> children;
    vector<int> children;
    int suffix_link;
    int start;
    int end;
    Node(int start, int end) : start(start), end(end)
    {
        children.resize(27, -1);
        suffix_link = 0;
    }
    inline bool has_child(int i)
    {
        //return children.find(i) != children.end();
        return children[i] != -1;
    }
};

struct SuffixTree
{
    int size;
    int i;
    vector<int> suffix_array;
    vector<Node> tree;
    inline int length(int index)
    {
        if (tree[index].end == -1)
            return i - tree[index].start + 1;
        return tree[index].end - tree[index].start + 1;
    }
    //se puede usar string& s
    SuffixTree(vector<int> &s)
    {
        size = s.size();
        tree.emplace_back(-1, -1);
        int remaining_suffix = 0;
        int active_node = 0;
        int active_edge = -1;
        int active_length = 0;
        for (i = 0; i < size; ++i)
        {
            int last_new = -1;
            remaining_suffix++;
            while (remaining_suffix > 0)
            {
                if (active_length == 0)
                    active_edge = i;
                if (!tree[active_node].has_child(s[active_edge]))
                {
                    tree[active_node].children[s[active_edge]] = tree.size();
                    tree.emplace_back(i, -1);
                    if (last_new != -1)
                    {
                        tree[last_new].suffix_link = active_node;
                        last_new = -1;
                    }
                }
                else
                {
                    int next = tree[active_node].children[s[active_edge]];
                    if (active_length >= length(next))
                    {
                        active_edge += length(next);
                        active_length -= length(next);
                        active_node = next;
                        continue;
                    }
                    if (s[tree[next].start + active_length] == s[i])
                    {
                        if (last_new != -1 and active_node != 0)
                        {
                            tree[last_new].suffix_link = active_node;
                        }
                        active_length++;
                        break;
                    }
                    int split_end = tree[next].start + active_length - 1;
                    int split = tree.size();
                    tree.emplace_back(tree[next].start, split_end);
                    tree[active_node].children[s[active_edge]] = split;
                    int new_leaf = tree.size();
                    tree.emplace_back(i, -1);
                    tree[split].children[s[i]] = new_leaf;
                    tree[next].start += active_length;
                    tree[split].children[s[tree[next].start]] = next;
                    if (last_new != -1)
                    {
                        tree[last_new].suffix_link = split;
                    }
                    last_new = split;
                }
                remaining_suffix--;
                if (active_node == 0 and active_length > 0)
                {
                    active_length--;
                    active_edge = i - remaining_suffix + 1;
                }
                else if (active_node != 0)
                {
                    active_node = tree[active_node].suffix_link;
                }
            }
        }
        i = size - 1;
    }
    vector<int> lcp;
    //last for lcp
    void dfs(int node, int &index, int depth, int min_depth)
    {
        if (tree[node].end == -1 and node != 0)
        {
            suffix_array[index] = size - depth;
            if (index != 0)
            {
                lcp[index - 1] = min_depth;
            }
            index++;
        }
        for (auto it : tree[node].children)
        {
            //if(i.second != -1){
            //    dfs(i.second,index,depth + length(i.second));
            //    min_depth = depth;
            //}
            if (it != -1)
            {
                dfs(it, index, depth + length(it), min_depth);
                min_depth = depth;
            }
        }
    }
    void build_suffix_array()
    {
        suffix_array.resize(size, 0);
        lcp.resize(size, 0);
        int index = 0;
        int depth = 0;
        dfs(0, index, 0, 0);
    }
};
```