String

1 KMP

1.1 Fail

```cpp
// input: 0-based, output: 1-based
auto getFail (const std::string& s) {
  int n = s.size();
  std::vector<int> fail(n + 1);
  for (int i = 1, j = 0; i < n; ++i) {
    while (j && s[i] != s[j]) j = fail[j];
    fail[i + 1] = j += (s[i] == s[j]);
  }
  return fail;
}
```

1.2 Trans

```cpp
// input: 0-based, output: 1-based
auto getTrans(const std::string& s) {
  int n = s.size();
  auto fail = getFail(s);

  std::vector<int> trans(n + 1);
  for (int i = 1, j = 0; i < n; ++i) {
    while (j && s[i] != s[j]) j = fail[j];
    j += s[i] == s[j];
    while (2 * j > i + 1) j = fail[j];
    trans[i + 1] = j;
  }
  return trans;
}
```

1.3 occurrence

```cpp
// input: 0-based, output: 0-based
auto occurrence(const std::string& s, const std::string& t) {
  int n = s.size(), m = t.size();
  auto fail = getFail(t);

  std::vector<int> occur;
  for (int i = 0, j = 0; i < n; ++i) {
    while (j && s[i] != t[j]) j = fail[j];
    j += s[i] == t[j];
    if (j == m) {
      occur.push_back(i - m + 1);
      j = fail[j];
    }
  }
  return occur;
}
```

2 Manacher

2.1

```cpp
// input: 0-based, idx(s[i]) = 2i
std::vector<int> Manacher(const std::string& t) {
  std::string s = "#";
  for (auto& ch : t) {
    s += '$', s += ch;
  } s += '$';

  int n = s.size() - 1;
  std::vector<int> d(n + 1);
  for (int i = 1, j = 1; i <= n; ++i) {
    d[i] = i < j + d[j] ? std::min(d[2 * j - i], j + d[j] - i) : 1;
    while (i + d[i] <= n && i - d[i] >= 1 && s[i - d[i]] == s[i + d[i]]) ++d[i];
    if (i + d[i] > j + d[j]) j = i;
  }

  return d;
}
```

2.2

题意：找最长的子串满足存在一个字符串 S 使得该子串可以被表示成 S+$\overline{\text{S}}$+S+$\overline{\text{S}}$. 显然一个字符串只有 $O(n)$ 个本质不同的回文串，暴力 check 即可

```cpp
// input: 0-based, idx(s[i]) = 2i
int Manacher(const std::string& t) {
  std::string s = "#";
  for (auto& ch : t) {
    s += '$', s += ch;
  } s += '$';

  int n = s.size() - 1, ans = 0;
  std::vector<int> d(n + 1);
  for (int i = 1, j = 1; i <= n; ++i) {
    d[i] = i < j + d[j] ? std::min(d[2 * j - i], j + d[j] - i) : 1;
    while (i + d[i] <= n && i - d[i] >= 1 && s[i - d[i]] == s[i + d[i]]) ++d[i];
    if (i + d[i] > j + d[j]) {
      if (s[i] == '$') {
        for (int k = j + d[j]; k < i + d[i]; ++k) {
          if (s[k] == '$') continue;
          int l = (2 * i - k) >> 1, r = k >> 1;
          if ((r - l + 1) % 4 != 0) continue;
          int x = i >> 1;
          if (d[l + x] - 1 >= (r - l + 1) / 2) {
            ans = std::max(ans, r - l + 1);
          }
        }
      }
      j = i;
    }
  }

  return ans;
}
```

3 Z Function

```cpp
// input: 0-based, output: 0-based
auto ZFunction(const std::string& s) {
  int n = s.size();
  std::vector<int> z(n + 1);
  z[0] = n;
  for (int i = 1, j = 1; i < n; ++i) {
    z[i] = std::max(0, std::min(j + z[j] - i, z[i - j]));
    while (i + z[i] < n && s[i + z[i]] == s[z[i]]) z[i]++;
    if (i + z[i] > j + z[j]) j = i;
  }
  return z;
}
```

4 Suffix Array

要学会使用 sa,rk,height 数组，多观察性质，转化成 sa,rk,height 能做的

```cpp
// input: 0-based, output: 1-based
auto SuffixArray(const std::string& s) {
  int n = s.size();
  std::vector<int> sa(n + 1), rk(n + 1);
  std::iota(sa.begin() + 1, sa.end(), 1);
  std::sort(sa.begin() + 1, sa.end(), [&](int x, int y) {
    return s[x - 1] < s[y - 1];
  });

  rk[sa[1]] = 1;
  for (int i = 1; i < n; ++i) {
    rk[sa[i + 1]] = rk[sa[i]] + (s[sa[i + 1] - 1] != s[sa[i] - 1]);
  }

  std::vector<int> tmp(n + 1), cnt(n + 1);
  for (int k = 1; rk[sa[n]] != n; k <<= 1) {
    for (int i = n - k + 1, j = 1; i <= n; ++i, ++j) {
      tmp[j] = i;
    }
    for (int i = 1, j = k; i <= n; ++i) {
      if (sa[i] <= k) continue;
      tmp[++j] = sa[i] - k;
    }

    for (int i = 1; i <= n; ++i) {
      cnt[rk[i]]++;
    }
    for (int i = 1; i < rk[sa[n]]; ++i) {
      cnt[i + 1] += cnt[i];
    }
    for (int i = n; i >= 1; --i) {
      sa[cnt[rk[tmp[i]]]--] = tmp[i];
    }

    std::swap(rk, tmp);
    rk[sa[1]] = 1, cnt[tmp[sa[n]]] = 0;
    for (int i = 1; i < n; ++i) {
      cnt[tmp[sa[i]]] = 0;
```

```
      rk[sa[i + 1]] = rk[sa[i]] + (
        tmp[sa[i + 1]] != tmp[sa[i]] ||
        sa[i] + k - 1 == n ||
        tmp[sa[i + 1] + k] != tmp[sa[i] + k]
      );
    }
  }

  std::vector<int> height(n + 1);
  for (int i = 1, lcp = 0; i <= n; ++i) {
    if (rk[i] == 1) continue;
    if (lcp != 0) lcp--;
    while (
      i + lcp <= n &&
      sa[rk[i] - 1] + lcp <= n &&
      s[i + lcp - 1] == s[sa[rk[i] - 1] + lcp - 1]
    ) ++lcp;
    height[rk[i]] = lcp;
  }

  return std::tuple {
    std::move(sa),
    std::move(rk),
    std::move(height)
  };
}
```

4.1 求 Longest Common Substring

```
int n = s.size(), m = t.size();
auto [sa, rk, height] = SuffixArray(s + '$' + t);

std::array<int, 3> ans { 0, 0, 0 };
for (int i = 1; i <= n + m; ++i) {
  int x = sa[i], y = sa[i + 1];
  int len = height[i + 1];
  if (len <= ans[0]) continue;
  if (x <= n && y >= n + 2) {
    ans = { len, x - 1, y - n - 2 };
  }
  if (y <= n && x >= n + 2) {
    ans = { len, y - 1, x - n - 2 };
  }
}
```

5 Suffix Automaton

```
// Node: 1-based "" 为 1 号节点
struct SAM {
  static constexpr int N = 26;
  struct Node {
    int len;
    int link;
    std::array<int, N> next;
    Node() : len(), link(), next() {}
  };
```

```
i64 substr;
std::vector<Node> t;

SAM (int n = 0) {
  t.reserve(n);
  t.assign(2, Node());
  t[0].next.fill(1);
  t[0].len = -1;
  substr = 0;
}

int newNode() {
  t.push_back();
  return t.size() - 1;
}

int extend(int p, int c) {
  int cur = newNode();
  t[cur].len = t[p].len + 1;

  while (t[p].next[c] == 0) {
    t[p].next[c] = cur;
    p = t[p].link;
  }

  int q = t[p].next[c];
  if (t[q].len == t[p].len + 1) {
    t[cur].link = q;
  } else {
    int r = newNode();
    t[r].len = t[p].len + 1;
    t[r].link = t[q].link;
    t[r].next = t[q].next;
    t[q].link = r;
    while (t[p].next[c] == q) {
      t[p].next[c] = r;
      p = t[p].link;
    }
    t[cur].link = r;
  }

  substr += t[cur].len - t[t[cur].link].len;
  return cur;
}

int len(int p)          const {return t[p].len; }
int link(int p)         const {return t[p].link; }
int next(int p, int x) const {return t[p].next[x]; }
int size()              const {return t.size(); }
i64 count ()            const {return substr; }
```

```
  // [ SAM 节点的个数（不含空节点），后缀树 ]
  auto getTree() {
    int n = t.size();
    std::vector<std::vector<int>> adj(n);
    for (int i = 2; i < n; ++i) {
      adj[t[i].link].push_back(i);
    }
    return std::pair { n - 1, std::move(adj) };
  }
};
```

5.1 弦论

计算 kth 子串，`t == 1` 时多次出现需要多次计算

```
int n = s.size();
SAM sam(n);
vector<int> p(n + 1);
p[0] = 1;
for (int i = 0; i < n; ++i) {
  p[i + 1] = sam.extend(p[i], s[i] - 'a');
}

auto [m, adj] = sam.getTree();
vector<i64> siz(m + 1);
if (t == 1) { // 考虑 endpos.size()
  for (int i = 1; i <= n; ++i) {
    siz[p[i]]++;
  }
  auto dfs = [&](auto &&dfs, int u) -> void {
    for (auto v : adj[u]) {
      dfs(dfs, v);
      siz[u] += siz[v];
    }
  }; dfs(dfs, 1);
} else { // 否则一个集合只算一次
  for (int i = 1; i <= m; ++i) {
    siz[i] = 1;
  }
}
siz[1] = 0;

vector<int> deg(m + 1);
adj.assign(m + 1, {});
for (int u = 1; u <= m; ++u) {
  for (int ch = 0; ch < 26; ++ch) {
    int v = sam.next(u, ch);
    if (v == 0) continue;
    adj[v].push_back(u);
    deg[u]++;
  }
}

i64 substr = 0;
vector<i64> dp = siz;
```

```cpp
for (int u = 2; u <= m; ++u) {
  substr += (sam.len(u) - sam.len(sam.link(u))) * siz[u];
}
if (substr < k) {
  cout << "-1\n";
  return;
}

queue<int> que;
for (int u = 1; u <= m; ++u) {
  if (deg[u] == 0) {
    que.push(u);
  }
}
while (!que.empty()) {
  int u = que.front();
  que.pop();
  for (auto v : adj[u]) {
    dp[v] += dp[u];
    if (--deg[v] == 0) {
      que.push(v);
    }
  }
}

int u = 1;
string ans;
while (k > siz[u]) {
  ans.push_back('$');
  k -= siz[u];
  for (int ch = 0; ch < 26; ++ch) {
    int v = sam.next(u, ch);
    if (v == 0) continue;
    ans.back() = 'a' + ch;
    if (k > dp[v]) {
      k -= dp[v];
    } else break;
  }
  u = sam.next(u, ans.back() - 'a');
}
cout << ans << "\n";
```

6 Palindromic Automaton

```cpp
//  odd root: 1
// even root: 0
struct PAM {
  static constexpr int N = 26;
  struct Node {
    int len, fail;
    std::array<int, N> next;
    Node() : len(0), fail(0), next{} {}
  };

  int cur;
  std::vector<int> s;
```

```
  std::vector<Node> t;
  PAM(int n = 0) {
    s.reserve(n);
    t.reserve(n + 2);
    t.assign(2, Node());
    t[t[0].fail = 1].len = -1;
  }

  int newNode() {
    t.emplace_back();
    return t.size() - 1;
  }

  int append(int p, int ch) {
    int n = s.size();
    s.push_back(ch);

    auto get = [&](int p) {
      while (n - t[p].len - 1 < 0 || ch != s[n - t[p].len - 1]) {
        p = t[p].fail;
      }
      return p;
    };

    p = get(p);
    if (t[p].next[ch] == 0) {
      int cur = newNode();
      t[cur].len = t[p].len + 2;
      t[p].next[ch] = cur;
      if (t[cur].len != 1) {
        t[cur].fail = t[get(t[p].fail)].next[ch];
      }
    }

    return t[p].next[ch];
  }

  int len(int p)         const { return t[p].len; }
  int fail(int p)        const { return t[p].fail; }
  int next(int p, int x) const { return t[p].next[x]; }
  int size()             const { return t.size(); }
};
```

7 Aho Corasick Automaton

```
// 记得 work
// 树根 "" 为 1 号节点
// adj 为失配树的子节点
struct ACAM {
  static constexpr int N = 26;
  struct Node {
    int len, fail;
    std::vector<int> adj;
    std::array<int, N> next;
    Node() : len(0), fail(0), adj{}, next{} {}
  };
```

```
  std::vector<Node> t;

  ACAM (int n = 0) {
    t.reserve(n);
    t.assign(2, Node());
    t[0].next.fill(1);
    t[0].len = -1;
    t[0].adj.push_back(1);
  }

  int newNode() {
    t.emplace_back();
    return t.size() - 1;
  }

  int insert(const std::string& s) {
    int p = 1;
    for (auto c : s) {
      int x = c - 'a';
      if (t[p].next[x] == 0) {
        t[p].next[x] = newNode();
        t[t[p].next[x]].len = t[p].len + 1;
      }
      p = t[p].next[x];
    }
    return p;
  }

  void work() {
    std::queue<int> q;
    q.push(1);

    while (!q.empty()) {
      int u = q.front();
      q.pop();

      for (int i = 0; i < N; i++) {
        if (t[u].next[i] == 0) {
          t[u].next[i] = t[t[u].fail].next[i];
        } else {
          t[t[u].next[i]].fail = t[t[u].fail].next[i];
          t[t[t[u].fail].next[i]].adj.push_back(t[u].next[i]);
          q.push(t[u].next[i]);
        }
      }
    }
  }

  int len(int p)                    const { return t[p].len; }
  int fail(int p)                   const { return t[p].fail; }
  const std::vector<int>& adj(int p) const { return t[p].adj; }
  int next(int p, int x)            const { return t[p].next[x]; }
  int size()                        const { return t.size(); }
};
```