String

Yunfan Wang

02-24-2023

Contents

KMP Algorithm

C++:

```cpp
vector<int> prefix_function(string s) {
  int n = (int)s.length();
  vector<int> pi(n);
  for (int i = 1; i < n; i++) {
    int j = pi[i - 1];
    while (j > 0 && s[i] != s[j]) j = pi[j - 1];
    if (s[i] == s[j]) j++;
    pi[i] = j;
  }
  return pi;
}

vector<int> find_occurrences(string text, string pattern) {
  string cur = pattern + '#' + text;
  int sz1 = text.size(), sz2 = pattern.size();
  vector<int> v;
  vector<int> lps = prefix_function(cur);
  for (int i = sz2 + 1; i <= sz1 + sz2; i++) {
    if (lps[i] == sz2)
      v.push_back(i - 2 * sz2);
  }
  return v;
}
```

Python:

```python
def prefix_function(s):
    n = len(s)
    pi = [0] * n
    for i in range(1, n):
```

```
        j = pi[i - 1]
        while j > 0 and s[i] != s[j]:
            j = pi[j - 1]
        if s[i] == s[j]:
            j += 1
        pi[i] = j
    return pi

def find_occurrences(t, s):
    cur = s + '#' + t
    sz1, sz2 = len(t), len(s)
    ret = []
    lps = prefix_function(cur)
    for i in range(sz2 + 1, sz1 + sz2 + 1):
        if lps[i] == sz2:
            ret.append(i - 2 * sz2)
    return ret
```

Count occurrence

C++:

```
vector<int> ans(n + 1);
for (int i = 0; i < n; i++) ans[pi[i]]++;
for (int i = n - 1; i > 0; i--) ans[pi[i - 1]] += ans[i];
for (int i = 0; i <= n; i++) ans[i]++;
```

Python:

```
ans = [0] * (n + 1)
for i in range(0, n):
    ans[pi[i]] += 1
for i in range(n - 1, 0, -1):
    ans[pi[i - 1]] += ans[i]
for i in range(0, n + 1):
    ans[i] += 1
```

Boyer-Moore Algorithm

CPython/Pypy has this implementation for `str.find`, but Java & C++ doesn't guarantee that.

Trie

C++:

```cpp
struct trie {
  int nex[100000][26], cnt;
  bool exist[100000];  // existence of a string ending with this vertex

  void insert(char *s, int l) {  // insert a string
    int p = 0;
    for (int i = 0; i < l; i++) {
      int c = s[i] - 'a';
      if (!nex[p][c]) nex[p][c] = ++cnt;  // add a vertex otherwise
      p = nex[p][c];
    }
    exist[p] = 1;
  }

  bool find(char *s, int l) {  // search for a string
    int p = 0;
    for (int i = 0; i < l; i++) {
      int c = s[i] - 'a';
      if (!nex[p][c]) return 0;
      p = nex[p][c];
    }
    return exist[p];
  }
};
```

Python:

```python
class trie:
    nex = [[0 for i in range(26)] for j in range(100000)]
    cnt = 0
    exist = [False] * 100000  # existence of a string ending with this vertex

    def insert(self, s):  # insert a string
        p = 0
        for i in s:
            c = ord(i) - ord('a')
            if not self.nex[p][c]:
                self.cnt += 1
                self.nex[p][c] = self.cnt  # add a vertex otherwise
            p = self.nex[p][c]
        self.exist[p] = True

    def find(self, s):  # search for a string
        p = 0
        for i in s:
            c = ord(i) - ord('a')
            if not self.nex[p][c]:
                return False
            p = self.nex[p][c]
        return self.exist[p]
```

Min Representation

Minimum (alphabetical order) string T of S such that

$$\exists i, S[i, ..., n] + S[1, ..., i-1] = T$$

```cpp
int k = 0, i = 0, j = 1;
while (k < n && i < n && j < n) {
  if (sec[(i + k) % n] == sec[(j + k) % n]) {
    ++k;
  } else {
    if (sec[(i + k) % n] > sec[(j + k) % n])
      ++i;
    else
      ++j;
    k = 0;
    if (i == j) i++;
  }
}
i = min(i, j);
```

```python
k, i, j = 0, 0, 1
while k < n and i < n and j < n:
    if sec[(i + k) % n] == sec[(j + k) % n]:
        k += 1
    else:
        if sec[(i + k) % n] > sec[(j + k) % n]:
            i += 1
        else:
            j += 1
        k = 0
        if i == j:
            i += 1
i = min(i, j)
```

SAM

```
struct state {
  int len, link;
  std::map<char, int> next;
};

const int MAXLEN = 100000;
state st[MAXLEN * 2]; // a directed acyclic graph
int sz, last;

void sam_init() {
  st[0].len = 0;
  st[0].link = -1;
  sz++;
  last = 0;
}

void sam_extend(char c) {
  int cur = sz++;
  st[cur].len = st[last].len + 1;
  int p = last;
  while (p != -1 && !st[p].next.count(c)) {
    st[p].next[c] = cur;
    p = st[p].link;
  }
  if (p == -1) {
    st[cur].link = 0;
  } else {
    int q = st[p].next[c];
    if (st[p].len + 1 == st[q].len) {
      st[cur].link = q;
    } else {
      int clone = sz++;
      st[clone].len = st[p].len + 1;
      st[clone].next = st[q].next;
      st[clone].link = st[q].link;
      while (p != -1 && st[p].next[c] == q) {
        st[p].next[c] = clone;
        p = st[p].link;
      }
      st[q].link = st[cur].link = clone;
    }
  }
  last = cur;
}
```

Suffix Array

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>

using namespace std;

const int N = 1000010;

char s[N];
// key1[i] = rk[id[i]] (as the first keyword array for radix sorting)
int n, sa[N], rk[N], oldrk[N << 1], id[N], key1[N], cnt[N];

bool cmp(int x, int y, int w) {
  return oldrk[x] == oldrk[y] && oldrk[x + w] == oldrk[y + w];
}

int main() {
  int i, m = 127, p, w;

  scanf("%s", s + 1);
  n = strlen(s + 1);
  for (i = 1; i <= n; ++i) ++cnt[rk[i] = s[i]];
  for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
  for (i = n; i >= 1; --i) sa[cnt[rk[i]]--] = i;

  for (w = 1;; w <<= 1, m = p) {  // m=p is the value set for optimized radix sorting
    for (p = 0, i = n; i > n - w; --i) id[++p] = i;
    for (i = 1; i <= n; ++i)
      if (sa[i] > w) id[++p] = sa[i] - w;

    memset(cnt, 0, sizeof(cnt));
    for (i = 1; i <= n; ++i) ++cnt[key1[i] = rk[id[i]]];

    for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
    for (i = n; i >= 1; --i) sa[cnt[key1[i]]--] = id[i];
    memcpy(oldrk + 1, rk + 1, n * sizeof(int));
    for (p = 0, i = 1; i <= n; ++i)
      rk[sa[i]] = cmp(sa[i], sa[i - 1], w) ? p : ++p;
    if (p == n) {
      for (int i = 1; i <= n; ++i) sa[rk[i]] = i;
      break;
    }
  }

  for (i = 1; i <= n; ++i) printf("%d ", sa[i]);

  return 0;
}
```

Palindromic Tree (EER Tree)

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int mod = 1e9 + 7;
const int maxn = 1000000 + 5;
```

```
inline int add(int x, int y) {
  x += y;
  return x >= mod ? x -= mod : x;
}

namespace pam {
int sz, tot, last;
int ch[maxn][26], len[maxn], fail[maxn];
int cnt[maxn], dep[maxn], dif[maxn], slink[maxn];
char s[maxn];

int node(int l) { // build a new node length l
  sz++;
  memset(ch[sz], 0, sizeof(ch[sz]));
  len[sz] = l;
  fail[sz] = 0;
  cnt[sz] = 0;
  dep[sz] = 0;
  return sz;
}

void clear() {  // init
  sz = -1;
  last = 0;
  s[tot = 0] = '$';
  node(0);
  node(-1);
  fail[0] = 1;
}

int getfail(int x) {  // finding suffix palindrome
  while (s[tot - len[x] - 1] != s[tot]) x = fail[x];
  return x;
}

void insert(char c) {  // build tree
  s[++tot] = c;
  int now = getfail(last);
  if (!ch[now][c - 'a']) {
    int x = node(len[now] + 2);
    fail[x] = ch[getfail(fail[now])][c - 'a'];
    dep[x] = dep[fail[x]] + 1;
    ch[now][c - 'a'] = x;
    dif[x] = len[x] - len[fail[x]];
    if (dif[x] == dif[fail[x]])
      slink[x] = slink[fail[x]];
    else
      slink[x] = fail[x];
  }
  last = ch[now][c - 'a'];
  cnt[last]++;
}
}  // namespace pam

using pam::dif;
using pam::fail;
using pam::len;
using pam::slink;
int n, dp[maxn], g[maxn];
char s[maxn], t[maxn];
```

```cpp
int main() {
  pam::clear();
  scanf("%s", s + 1);
  n = strlen(s + 1);
  for (int i = 1, j = 0; i <= n; i++) t[++j] = s[i], t[++j] = s[n - i + 1];
  dp[0] = 1;
  for (int i = 1; i <= n; i++) {
    pam::insert(t[i]);
    for (int x = pam::last; x > 1; x = slink[x]) {
      g[x] = dp[i - len[slink[x]] - dif[x]];
      if (dif[x] == dif[fail[x]]) g[x] = add(g[x], g[fail[x]]);
      if (i % 2 == 0) dp[i] = add(dp[i], g[x]);  // update DP array at even indices
    }
  }
  printf("%d", dp[n]);
  return 0;
}
```