ACM Template

Fibonacci's Rabbit

Last build at May 10, 2019

Contents

1 字符串

1.1 字符串最小最大表示

```cpp
#include <algorithm>
using namespace std;

// T = sec[k..n-1]+sec[0..k-1]
// k为返回值,n为sec的大小,T为sec的最小表示法
int get_min(const char* sec, int n) {
    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 {
            sec[(i + k) % n] > sec[(j + k) % n] ? i = i + k + 1 : j = j + k + 1;
            if (i == j) i++;
            k = 0;
        }
    }
    i = min(i, j);
    return i;
}

int get_max(const char* sec, int n) {
    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 {
            sec[(i + k) % n] < sec[(j + k) % n] ? i = i + k + 1 : j = j + k + 1;
            if (i == j) i++;
            k = 0;
        }
    }
    i = min(i, j);
    return i;
}
```

1.2 kmp 算法

以 i 结尾的最小循环节: $i - f[i]$

```cpp
#include <bits/stdc++.h>

using namespace std;

const int maxn = 10000 + 5;

int f[maxn];

void get_next(const char *P, int n) {
    f[0] = 0;
    f[1] = 0;  // 递推边界初值
    for (int i = 1; i < n; i++) {
        int j = f[i];
        while (j && P[i] != P[j]) j = f[j];
        f[i + 1] = (P[i] == P[j] ? j + 1 : 0);
    }
}
```

1.3 z 函数

```cpp
#include <bits/stdc++.h>

using namespace std;

const int maxn = 1000000 + 5;

int z[maxn];

// s 为待匹配的字符串指针
// n 为字符串长度
// z[i]是s和s+i的最大公共前缀长度。
void z_function(const char* s, int n) {
    fill_n(z, n, 0);
    for (int i = 1, l = 0, r = 0; i < n; ++i) {
```

```
        if (i <= r) z[i] = min(r - i + 1, z[i - l]);
        while (i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];
        if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
    }
}
```

1.4 manacher

回文匹配算法 (可用后缀数组代替, 但是比后缀数组简洁得多)

```
#include <bits/stdc++.h>

using namespace std;

const int maxn = 1000000;
int d1[maxn], d2[maxn];

// s 为字符串,也可以是const string&
// n 是字符串长度,即为s.length()
// d1为奇数回文长度(算上起点),总长度为d1[.]*2-1
// d2为偶数回文长度(算上起点),总长度为d2[.]*2
void Manacher(const char* s, int n) {
    for (int i = 0, l = 0, r = -1; i < n; i++) {
        int k = (i > r) ? 1 : min(d1[l + r - i], r - i);
        while (0 <= i - k && i + k < n && s[i - k] == s[i + k]) {
            k++;
        }
        d1[i] = k--;
        if (i + k > r) {
            l = i - k;
            r = i + k;
        }
    }

    for (int i = 0, l = 0, r = -1; i < n; i++) {
        int k = (i > r) ? 0 : min(d2[l + r - i + 1], r - i + 1);
        while (0 <= i - k - 1 && i + k < n && s[i - k - 1] == s[i + k]) {
            k++;
        }
        d2[i] = k--;
        if (i + k > r) {
            l = i - k - 1;
            r = i + k;
        }
    }
}

// 判断[l,r)是否回文
bool is_palindrome(int l, int r) {
    if (l == r) return true;
    if ((r - l) & 1) {
        return d1[l + (r - l) / 2] >= (r - l + 1) / 2;
    } else {
        return d2[l + (r - l) / 2] >= (r - l) / 2;
    }
}
```

1.5 字典树

```
#include <cstring>
#include <vector>
using namespace std;

const int wordnum = 100;
const int wordlen = 4000;
const int maxnode = wordnum * wordlen + 10;
const int sigma_size = 26;

// 字母表为全体小写字母的Trie
struct Trie {
    int ch[maxnode][sigma_size];
    int val[maxnode];
    int sz;  // 结点总数
    void clear() {
        sz = 1;
        memset(ch[0], 0, sizeof(ch[0]));
    }                                   // 初始时只有一个根结点
    int idx(char c) { return c - 'a'; }  // 字符c的编号
```

```
20
21      // 插入字符串s，附加信息为v。注意v必须非0，因为0代表“本结点不是单词结点”
22      void insert(const char *s, int v) {
23          int u = 0, n = strlen(s);
24          for (int i = 0; i < n; i++) {
25              int c = idx(s[i]);
26              if (!ch[u][c]) {  // 结点不存在
27                  memset(ch[sz], 0, sizeof(ch[sz]));
28                  val[sz] = 0;      // 中间结点的附加信息为0
29                  ch[u][c] = sz++;  // 新建结点
30              }
31              u = ch[u][c];  // 往下走
32          }
33          val[u] = v;  // 字符串的最后一个字符的附加信息为v
34      }
35  };
```

1.6 ac 自动机

```
1   #include <cstring>
2   #include <queue>
3
4   using namespace std;
5
6   const int SIGMA_SIZE = 128;
7   const int WORD_SIZE = 55;
8   const int WORD_NUM = 1005;
9   const int MAXNODE = WORD_SIZE * WORD_NUM + 10;
10
11  struct AhoCorasickAutomata {
12      int ch[MAXNODE][SIGMA_SIZE];
13      int f[MAXNODE];     // fail函数
14      int val[MAXNODE];   // 每个字符串的结尾结点都有一个非0的val
15      int last[MAXNODE];  // 输出链表的下一个结点
16      bool vis[MAXNODE];
17      int cnt[WORD_NUM];
18      int sz;
19
20      void init() {
21          sz = 1;
22          memset(ch[0], 0, sizeof(ch[0]));
23          memset(vis, 0, sizeof(vis));
24          memset(cnt, 0, sizeof(cnt));
25      }
26
27      // 字符c的编号
28      int idx(char c) const { return c; }
29
30      // 插入字符串。v必须非0
31      void insert(char* s, int v) {
32          int u = 0, n = strlen(s);
33          for (int i = 0; i < n; i++) {
34              int c = idx(s[i]);
35              if (!ch[u][c]) {
36                  memset(ch[sz], 0, sizeof(ch[sz]));
37                  val[sz] = 0;
38                  ch[u][c] = sz++;
39              }
40              u = ch[u][c];
41          }
42          val[u] = v;
43      }
44
45      // 递归打印以结点j结尾的所有字符串
46      void print(int j) {
47          int ret = 0;
48          if (j) {
49              cnt[val[j]]++;
50              print(last[j]);
51          }
52      }
53
54      // 在T中找模板
55      void find(const char* T) {
56          int n = strlen(T);
57          int j = 0;  // 当前结点编号，初始为根结点
58          for (int i = 0; i < n; i++) {  // 文本串当前指针
59              int c = idx(T[i]);
60              while (j && !ch[j][c]) j = f[j];  // 顺着细边走，直到可以匹配
```

```
            j = ch[j][c];
            if (val[j])
                print(j);
            else if (last[j])
                print(last[j]);  // 找到了!
        }
    }

    // 计算fail函数
    void getFail() {
        queue<int> q;
        f[0] = 0;
        // 初始化队列
        for (int c = 0; c < SIGMA_SIZE; c++) {
            int u = ch[0][c];
            if (u) {
                f[u] = 0;
                q.push(u);
                last[u] = 0;
            }
        }
        // 按BFS顺序计算fail
        while (!q.empty()) {
            int r = q.front();
            q.pop();
            for (int c = 0; c < SIGMA_SIZE; c++) {
                int u = ch[r][c];
                if (!u) continue;
                q.push(u);
                int v = f[r];
                while (v && !ch[v][c]) v = f[v];
                f[u] = ch[v][c];
                last[u] = val[f[u]] ? f[u] : last[f[u]];
            }
        }
    }
};  // namespace AhoCorasickAutomata

AhoCorasickAutomata ac;
```

1.7 后缀数组

全字符串找循环节　要有长度为 i 的循环节，就要满足以下条件：

$$rank[0]-rank[i] = 1$$
$$height[rank[0]] = len-i$$
$$len\%i == 0$$

找字串循环节最大重复次数　枚举长度 len，枚举起点 j，求 $lcp(j, j+len)$

$$ans = lcp/len + 1$$
$$k = j - (len - ans\%len)$$
$$if(k > 0 \&\& lcp(k, k+len) >= len)\{ans++;\}$$

求 ans 最大值

复杂度

* 后缀数组倍增法（时间 O(nlongn)，空间 O(4n)）
* 后缀数组 dc3 法（时间 O(n)，空间 O(10n)）
* 后置数组快排（适用于最大值很大的情况，除了 sa 数组，其他暂未测试）

1.7.1 后缀数组-倍增法

```
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;

namespace SuffixArray {
    using std::printf;

    const int maxn = 1e7 + 5;  // max(字符串长度, 最大字符值加1)
```

```
10
11      int s[maxn];                    // 原始字符数组（最后一个字符应必须是0，而前面的字符必须非0）
12      int sa[maxn];                   // 后缀数组
13      int rank[maxn];                 // 名次数组. rank[0]一定是n-1，即最后一个字符
14      int height[maxn];               // height数组
15      int t[maxn], t2[maxn], c[maxn]; // 辅助数组
16      int n;                          // 字符个数（包括最后一个0字符）
17
18      void init() { n = 0; }
19
20      // m为最大字符值加1。调用之前需设置好s和n
21      void build_sa(int m) {
22          int i, *x = t, *y = t2;
23          for (i = 0; i < m; i++) c[i] = 0;
24          for (i = 0; i < n; i++) c[x[i] = s[i]]++;
25          for (i = 1; i < m; i++) c[i] += c[i - 1];
26          for (i = n - 1; i >= 0; i--) sa[--c[x[i]]] = i;
27          for (int k = 1; k <= n; k <<= 1) {
28              int p = 0;
29              for (i = n - k; i < n; i++) y[p++] = i;
30              for (i = 0; i < n; i++)
31                  if (sa[i] >= k) y[p++] = sa[i] - k;
32              for (i = 0; i < m; i++) c[i] = 0;
33              for (i = 0; i < n; i++) c[x[y[i]]]++;
34              for (i = 0; i < m; i++) c[i] += c[i - 1];
35              for (i = n - 1; i >= 0; i--) sa[--c[x[y[i]]]] = y[i];
36              swap(x, y);
37              p = 1;
38              x[sa[0]] = 0;
39              for (i = 1; i < n; i++) x[sa[i]] = y[sa[i - 1]] == y[sa[i]] && y[sa[i - 1] + k] == y[sa[i] + k
                    ] ? p - 1 : p++;
40              if (p >= n) break;
41              m = p;
42          }
43      }
44
45      void build_height() {
46          int i, k = 0;
47          for (i = 0; i < n; i++) rank[sa[i]] = i;
48          for (i = 0; i < n; i++) {
49              if (k) k--;
50              int j = sa[rank[i] - 1];
51              while (s[i + k] == s[j + k]) k++;
52              height[rank[i]] = k;
53          }
54      }
55  }  // namespace SuffixArray
56
57  // 编号辅助
58  namespace SuffixArray {
59      int idx[maxn];
60
61      // 给字符串加上一个字符，属于字符串i
62      void add(int ch, int i) {
63          idx[n] = i;
64          s[n++] = ch;
65      }
66  }  // namespace SuffixArray
67
68  // LCP 模板
69  namespace SuffixArray {
70      using std::min;
71      int dp[maxn][20];
72      void initRMQ(int n) {
73          for (int i = 1; i <= n; i++) dp[i][0] = height[i];
74          for (int j = 1; (1 << j) <= n; j++)
75              for (int i = 1; i + (1 << j) - 1 <= n; i++) dp[i][j] = min(dp[i][j - 1], dp[i + (1 << (j - 1))
                    ][j - 1]);
76          return;
77      }
78
79      void initRMQ() { initRMQ(n - 1); }
80
81      int lcp(int a, int b) {
82          int ra = rank[a], rb = rank[b];
83          if (ra > rb) swap(ra, rb);
84          int k = 0;
85          while ((1 << (k + 1)) <= rb - ra) k++;
86          return min(dp[ra + 1][k], dp[rb - (1 << k) + 1][k]);
87      }
```

```
}  // namespace SuffixArray

// 调试信息
namespace SuffixArray {
    using std::printf;
    void debug() {
        printf("n:%d\n", n);

        printf("%8s", "");
        for (int i = 0; i < n; i++) {
            printf("%4d", i);
        }
        printf("\n");

        printf("%8s", "s:");
        for (int i = 0; i < n; i++) {
            printf("%4d", s[i]);
        }
        printf("\n");

        printf("%8s", "sa:");
        for (int i = 0; i < n; i++) {
            printf("%4d", sa[i]);
        }
        printf("\n");

        printf("%8s", "rank:");
        for (int i = 0; i < n; i++) {
            printf("%4d", rank[i]);
        }
        printf("\n");

        printf("%8s", "height:");
        for (int i = 0; i < n; i++) {
            printf("%4d", height[i]);
        }
        printf("\n");
    }
}  // namespace SuffixArray
```

1.7.2 后缀数组-dc3

```
#include <algorithm>

using namespace std;

/*
注意:
1.maxn开n的十倍大小;
2.dc3(r,sa,n+1,Max+1);r为待后缀处理的数组,sa为存储排名位置的数组,n+1和Max+1都和倍增一样
3.calheight(r,sa,n);和倍增一样
*/
// DC3 算法
namespace SuffixArray {
#define F(x) ((x) / 3 + ((x) % 3 == 1 ? 0 : tb))
#define G(x) ((x) < tb ? (x)*3 + 1 : ((x)-tb) * 3 + 2)

    const int maxn = 1e7 + 5;

    int wa[maxn], wb[maxn], wv[maxn], ws[maxn];
    int s[maxn], sa[maxn];
    int rank[maxn], height[maxn];
    int n;

    void init() { n = 0; }

    int c0(int *r, int a, int b) { return r[a] == r[b] && r[a + 1] == r[b + 1] && r[a + 2] == r[b + 2]; }

    int c12(int k, int *r, int a, int b) {
        if (k == 2)
            return r[a] < r[b] || (r[a] == r[b] && c12(1, r, a + 1, b + 1));
        else
            return r[a] < r[b] || (r[a] == r[b] && wv[a + 1] < wv[b + 1]);
    }

    void sort(int *r, int *a, int *b, int n, int m) {
        int i;
        for (i = 0; i < n; i++) wv[i] = r[a[i]];
        for (i = 0; i < m; i++) ws[i] = 0;
```

```
38          for (i = 0; i < n; i++) ws[wv[i]]++;
39          for (i = 1; i < m; i++) ws[i] += ws[i - 1];
40          for (i = n - 1; i >= 0; i--) b[--ws[wv[i]]] = a[i];
41          return;
42      }
43
44      void dc3(int *r, int *sa, int n, int m) {
45          int i, j, *rn = r + n, *san = sa + n, ta = 0, tb = (n + 1) / 3, tbc = 0, p;
46          r[n] = r[n + 1] = 0;
47          for (i = 0; i < n; i++)
48              if (i % 3 != 0) wa[tbc++] = i;
49          sort(r + 2, wa, wb, tbc, m);
50          sort(r + 1, wb, wa, tbc, m);
51          sort(r, wa, wb, tbc, m);
52          for (p = 1, rn[F(wb[0])] = 0, i = 1; i < tbc; i++) rn[F(wb[i])] = c0(r, wb[i - 1], wb[i]) ? p - 1
                : p++;
53          if (p < tbc)
54              dc3(rn, san, tbc, p);
55          else
56              for (i = 0; i < tbc; i++) san[rn[i]] = i;
57          for (i = 0; i < tbc; i++)
58              if (san[i] < tb) wb[ta++] = san[i] * 3;
59          if (n % 3 == 1) wb[ta++] = n - 1;
60          sort(r, wb, wa, ta, m);
61          for (i = 0; i < tbc; i++) wv[wb[i] = G(san[i])] = i;
62          for (i = 0, j = 0, p = 0; i < ta && j < tbc; p++) sa[p] = c12(wb[j] % 3, r, wa[i], wb[j]) ? wa[i
                ++] : wb[j++];
63          for (; i < ta; p++) sa[p] = wa[i++];
64          for (; j < tbc; p++) sa[p] = wb[j++];
65          return;
66      }
67
68      void build_height(int n) {
69          int i, j, k = 0;
70          for (i = 1; i <= n; i++) rank[sa[i]] = i;
71          for (i = 0; i < n; height[rank[i++]] = k)
72              for (k ? k-- : 0, j = sa[rank[i] - 1]; s[i + k] == s[j + k]; k++)
73                  ;
74          return;
75      }
76
77      void build_height() { build_height(n - 1); }
78
79      void build_sa(int m) { dc3(s, sa, n, m); }
80
81  }  // namespace SuffixArray
```

1.7.3 后缀数组-快排

```
1   #include <cstdio>
2   #include <algorithm>
3   #include <cstring>
4   using namespace std;
5
6   namespace SuffixArray {
7       using std::printf;
8
9       const int maxn = 1e7 + 5;  // max(字符串长度, 最大字符值加1)
10
11      int s[maxn];                        // 原始字符数组（最后一个字符应必须是0，而前面的字符必须非0）
12      int sa[maxn];                       // 后缀数组
13      int t[maxn], rank[maxn], c[maxn];   // 辅助数组
14      int n;                              // 字符个数（包括最后一个0字符）
15
16      void init() { n = 0; }
17
18      int k;
19      bool compare_sa(int i, int j) {
20          if (rank[i] != rank[j]) {
21              return rank[i] < rank[j];
22          } else {
23              int ri = i + k < n ? rank[i + k] : -1;
24              int rj = j + k < n ? rank[j + k] : -1;
25              return ri < rj;
26          }
27      }
28
29      void build_sa(int _) {
30          for (int i = 0; i < n; i++) {
```

```
            sa[i] = i;
            rank[i] = i < n ? s[i] : -1;
        }
        for (k = 1; k < n; k <<= 1) {
            sort(sa, sa + n, compare_sa);
            t[sa[0]] = 0;
            for (int i = 1; i < n; i++) {
                t[sa[i]] = t[sa[i - 1]] + (compare_sa(sa[i - 1], sa[i]) ? 1 : 0);
            }
            for (int i = 0; i < n; i++) {
                rank[i] = t[i];
            }
        }
    }

    int height[maxn];  // height数组
    void build_height() {
        int i, k = 0;
        for (i = 0; i < n; i++) {
            if (k) k--;
            int j = sa[rank[i] - 1];
            while (s[i + k] == s[j + k]) k++;
            height[rank[i]] = k;
        }
    }
}  // namespace SuffixArray
```

1.8 字符串哈希

```
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;

const int maxn = 40000 + 10; // 字符串长度

// 字符串哈希（概率算法）
struct StringHash {
    const int x;  // 随便取
    unsigned long long H[maxn], xp[maxn];
    int n;
    StringHash() : x(123) {}
    // n为字符串长度
    void init(const char* s, int n) {
        this->n = n;
        H[n] = 0;
        for (int i = n - 1; i >= 0; i--) H[i] = H[i + 1] * x + (s[i] - 'a');
        xp[0] = 1;
        for (int i = 1; i <= n; i++) xp[i] = xp[i - 1] * x;
    }
    // 从i开始，长度为L的字串的hash
    unsigned long long getHash(int i, int L) const {
        return H[i] - H[i + L] * xp[L];
    }
};
```

1.9 字符串分割

1.9.1 按字符分割

```
#include <iostream>
#include <cstring>
#include <vector>
using namespace std;

// 字符串分割，分隔符为字符，可为多字符，前后不留空字符串
// *a,b*c,d, 按,*分割 -> {"a","b","c","d"}
// 注意：源字符串s将会被改变，请勿使用string.c_str()
// s源字符串 t传出结果 sep分隔符字符串(分隔符为每个单字符)
void split(char *s, vector<string> &v,const char *sep) {
    char *p = strtok(s, sep);
    while (p) {
        v.push_back(string(p));
        p = strtok(NULL, sep);
    }
}
```

1.9.2 按字符串分割

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

// 字符串分割，分隔符为字符串，前后留空字符串
// cabcacac 按c分割 -> {"","ab","a","a",""}
// s源字符串 v传出结果 c分隔符字符串
void split(const string& s, vector<string>& v, const string& c) {
    string::size_type pos1, pos2;
    pos2 = s.find(c);
    pos1 = 0;
    while (string::npos != pos2) {
        v.push_back(s.substr(pos1, pos2 - pos1));

        pos1 = pos2 + c.size();
        pos2 = s.find(c, pos1);
    }
    if (pos1 <= s.length()) v.push_back(s.substr(pos1));
    // 如果要去除最后空串,用下方语句替代上一条
    // if (pos1 != s.length()) v.push_back(s.substr(pos1));
}
```

1.9.3 按字符分割 (STL)

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

// 字符串分割，分隔符为字符，可为多字符，前后不留空字符串
// **a,b*c,d, 按,*分割 -> {"a","b","c","d"}
// strtok 的 实现
// s源字符串 t传出结果 sep分隔符字符串(分隔符为每个单字符)
void split(const string &s, vector<string> &v, const string &sep) {
    typedef string::size_type string_size;
    string_size i = 0;
    while (i != s.size()) {
        //找到字符串中首个不等于分隔符的字母;
        int flag = 0;
        while (i != s.size() && flag == 0) {
            flag = 1;
            for (string_size x = 0; x < sep.size(); ++x) {
                if (s[i] == sep[x]) {
                    ++i;
                    flag = 0;
                    break;
                }
            }
        }

        //找到又一个分隔符，将两个分隔符之间的字符串取出;
        flag = 0;
        string_size j = i;
        while (j != s.size() && flag == 0) {
            for (string_size x = 0; x < sep.size(); ++x) {
                if (s[j] == sep[x]) {
                    flag = 1;
                    break;
                }
            }
            if (flag == 0) ++j;
        }
        if (i != j) {
            v.push_back(s.substr(i, j - i));
            i = j;
        }
    }
}
```