

KMP 算法概述

基本概念

KMP (Knuth-Morris-Pratt) 算法：高效的字符串匹配算法

核心思想：利用已匹配的信息，避免重复比较

关键：

- **next 数组**：记录模式串的自我匹配信息
- **失配指针**：匹配失败时的跳转位置

时间复杂度：

- 预处理： $O(m)$ ，其中 m 为模式串长度
- 匹配： $O(n)$ ，其中 n 为文本串长度
- 总复杂度： $O(n + m)$

KMP 算法原理

next 数组定义

$\text{next}[i]$: 模式串 $P[1 \dots i]$ 的最长相等真前缀和真后缀的长度

示例: 模式串 "ababc"

i	子串	next[i]	说明
1	a	0	没有真前缀
2	ab	0	前缀 "a" \neq 后缀 "b"
3	aba	1	前缀 "a" = 后缀 "a"
4	abab	2	前缀 "ab" = 后缀 "ab"
5	ababc	0	没有相等前后缀

KMP 核心思想

暴力匹配的问题：每次失配时从头开始匹配

KMP 优化：利用 next 数组跳过不必要的比较

匹配过程：

文本串： a b a b a b c a

模式串： a b a b c

✓ ✓ ✓ ✓ x

next[4]=2: a b a b c

✓ ✓ x

next[2]=0: a b a b c

next 数组计算

```
const int N = 1000010;

char P[N];    // 模式串 (从1开始)
int nxt[N];   // next数组
int n;        // 模式串长度

void computeNext() {
    nxt[1] = 0; // 第一个字符的next值为0

    for (int i = 2, j = 0; i <= n; i++) {
        // j > 0 且不匹配时, 回退到 nxt[j]
        while (j > 0 && P[i] != P[j + 1])
            j = nxt[j];

        // 匹配成功, j前进
        if (P[i] == P[j + 1])
            j++;

        nxt[i] = j;
    }
}
```

KMP 匹配过程

```
char S[N];    // 文本串 (从1开始)
int m;        // 文本串长度

void kmpMatch() {
    for (int i = 1, j = 0; i <= m; i++) {
        // 不匹配时, 利用next数组跳转
        while (j > 0 && S[i] != P[j + 1])
            j = nxt[j];

        // 匹配成功, j前进
        if (S[i] == P[j + 1])
            j++;

        // 完全匹配
        if (j == n) {
            printf("%d\n", i - n + 1); // 输出匹配起始位置
            j = nxt[j]; // 继续寻找下一个匹配
        }
    }
}
```

KMP 完整模板

```
/* S 模式串与 T 匹配串，一对多 */
string s, p;
int nxt[N];
int main() {
    cin >> s >> p;

    int n = s.size(), m = p.size();
    s = " " + s, p = " " + p;
    /* 构造回跳数组 nxt[j] */
    nxt[1] = 0;
    for (int i = 2, j = 0; i <= m; i++) {
        while (j && p[i] != p[j + 1])
            j = nxt[j];
        if (p[i] == p[j + 1])
            ++j;
        nxt[i] = j;
    }

    /* 匹配过程，进可攻退可守，退而求其次! */
    for (int i = 1, j = 0; i <= n; i++) {
        while (j && s[i] != p[j + 1])
            j = nxt[j];
        if (s[i] == p[j + 1])
            ++j;
        if (j == m)
            cout << i - m + 1 << endl;
    }

    /* 输出匹配串的 border */
    for (int i = 1; i <= m; i++)
        cout << nxt[i] << " ";
}
```

字符串 Hash

基本概念

字符串 Hash：将字符串映射为整数的算法

应用场景：

- 快速判断字符串相等
- 快速计算子串 Hash
- 字符串匹配的替代方案

优点:

- 预处理后 $O(1)$ 比较子串
- 实现简单, 运行效率高

缺点:

- 存在哈希冲突的可能性

多项式 Hash 方法

Hash 公式：

$$H(s) = \sum_{i=1}^n s[i] \times base^{n-i} \bmod mod$$

子串 Hash：对于子串 $s[l \dots r]$ ：

$$H(l, r) = H(r) - H(l - 1) \times base^{r-l+1} \bmod mod$$

双 Hash：使用两个不同的模数减少冲突

字符串 Hash 实现

```
typedef unsigned long long ULL;

const int N = 100010;
const int P = 131; // 质数基数

char str[N];
ULL h[N], p[N]; // h[i]前缀Hash, p[i] P的i次方

// 初始化Hash
void initHash(int n) {
    p[0] = 1;
    for (int i = 1; i <= n; i++) {
        p[i] = p[i - 1] * P;
        h[i] = h[i - 1] * P + str[i];
    }
}

// 获取子串Hash值
ULL getHash(int l, int r) {
    return h[r] - h[l - 1] * p[r - l + 1];
}

// 判断两个子串是否相等
bool isEqual(int l1, int r1, int l2, int r2) {
    return getHash(l1, r1) == getHash(l2, r2);
}
```

字符串 Hash 应用示例

```
int main() {  
    int n, m;  
    cin >> n >> m;  
    cin >> str + 1;  
  
    initHash(n);  
  
    while (m--) {  
        int l1, r1, l2, r2;  
        cin >> l1 >> r1 >> l2 >> r2;  
  
        if (isEqual(l1, r1, l2, r2))  
            cout << "Yes" << endl;  
        else  
            cout << "No" << endl;  
    }  
  
    return 0;  
}
```

双 Hash 实现

```
typedef unsigned long long ULL;

const int N = 100010;
const int P1 = 131, P2 = 13331; // 两个不同的质数
const ULL MOD1 = 1e9 + 7, MOD2 = 1e9 + 9;

char str[N];
ULL h1[N], h2[N], p1[N], p2[N];

// 初始化双Hash
void initDoubleHash(int n) {
    p1[0] = p2[0] = 1;
    for (int i = 1; i <= n; i++) {
        p1[i] = p1[i - 1] * P1 % MOD1;
        p2[i] = p2[i - 1] * P2 % MOD2;

        h1[i] = (h1[i - 1] * P1 + str[i]) % MOD1;
        h2[i] = (h2[i - 1] * P2 + str[i]) % MOD2;
    }
}

// 获取子串的双Hash值
pair<ULL, ULL> getDoubleHash(int l, int r) {
    ULL hash1 = (h1[r] - h1[l - 1] * p1[r - l + 1] % MOD1 + MOD1) % MOD1;
    ULL hash2 = (h2[r] - h2[l - 1] * p2[r - l + 1] % MOD2 + MOD2) % MOD2;
    return {hash1, hash2};
}

// 比较两个子串
bool isDoubleEqual(int l1, int r1, int l2, int r2) {
    auto hash1 = getDoubleHash(l1, r1);
    auto hash2 = getDoubleHash(l2, r2);
    return hash1 == hash2;
}
```

KMP 与 Hash 对比

算法特性比较

特性	KMP	字符串 Hash
时间复杂度	$O(n + m)$	$O(n + m)$
空间复杂度	$O(m)$	$O(n)$
准确性	100% 准确	可能有哈希冲突
实现难度	较复杂	较简单
适用场景	精确匹配	快速比较、最长回文等

选择建议

使用 KMP 的情况：

- 需要 100% 准确的匹配结果
- 需要找到所有匹配位置
- 需要 next 数组的其他应用

使用 Hash 的情况：

- 需要快速比较多个子串
- 处理回文串相关问题
- 对准确性要求不是极高

综合应用示例

循环节问题

问题：求字符串的最小循环节长度

KMP 解法：

```
int minCycleLength(char s[], int n) {  
    computeNext(s, n);  
    int len = n - nxt[n];  
    return (n % len == 0) ? len : n;  
}
```

原理：如果 $n \% (n - next[n]) == 0$ ，则存在循环节

循环节问题

样例与解析

样例 1: 字符串 "abccabccabc"

字符串: a b c a b c a b c

长度 $n = 9$

next数组: 0 0 0 1 2 3 4 5 6

计算: $\text{len} = n - \text{next}[n] = 9 - 6 = 3$

检查: $9 \% 3 = 0 \checkmark$

结果: 最小循环节长度为 3 ("abc")

样例 2：字符串 "abcbcab"

字符串：a b c a b c a b

长度 $n = 8$

next数组：0 0 0 1 2 3 4 5

计算： $\text{len} = n - \text{next}[n] = 8 - 5 = 3$

检查： $8 \% 3 \neq 0$ ✗

结果：最小循环节长度为 8（没有完整循环节）

完整代码实现：

```
const int N = 100010;
char s[N];
int nxt[N];

void computeNext(int n) {
    nxt[1] = 0;
    for (int i = 2, j = 0; i <= n; i++) {
        while (j && s[i] != s[j + 1]) j = nxt[j];
        if (s[i] == s[j + 1]) j++;
        nxt[i] = j;
    }
}

int minCycleLength(int n) {
    computeNext(n);
    int len = n - nxt[n];
    return (n % len == 0) ? len : n;
}
```

```
int main() {  
    // 测试样例1  
    strcpy(s + 1, "abcabcabc");  
    int n1 = strlen(s + 1);  
    cout << "样例1: " << s + 1 << endl;  
    cout << "最小循环节长度: " << minCycleLength(n1) << endl;  
  
    // 测试样例2  
    strcpy(s + 1, "abcabcab");  
    int n2 = strlen(s + 1);  
    cout << "样例2: " << s + 1 << endl;  
    cout << "最小循环节长度: " << minCycleLength(n2) << endl;  
  
    return 0;  
}
```

输出结果：

样例1: `abccabccab`

最小循环节长度: 3

样例2: `abccabccab`

最小循环节长度: 8

最长回文子串

样例与解析

样例：字符串 "babad"

字符串：b a b a d

长度 $n = 5$

奇长度回文分析：

- 中心在位置 1 ('b')：半径 0，回文 "b"，长度 1
- 中心在位置 2 ('a')：半径 1，回文 "aba"，长度 3
- 中心在位置 3 ('b')：半径 1，回文 "bab"，长度 3
- 中心在位置 4 ('a')：半径 0，回文 "a"，长度 1
- 中心在位置 5 ('d')：半径 0，回文 "d"，长度 1

偶长度回文分析：

- 中心在位置 1-2 ("ba")：不是回文
- 中心在位置 2-3 ("ab")：不是回文
- 中心在位置 3-4 ("ba")：不是回文
- 中心在位置 4-5 ("ad")：不是回文

最长回文子串："bab" 或 "aba"，长度 3

完整代码实现：

```
typedef unsigned long long ULL;

const int N = 100010;
const int P = 131;

char str[N];
ULL h[N], hr[N], p[N]; // h: 正序Hash, hr: 逆序Hash

// 初始化正序和逆序Hash
void initHash(int n) {
    p[0] = 1;
    for (int i = 1; i <= n; i++) {
        p[i] = p[i - 1] * P;
        h[i] = h[i - 1] * P + str[i];
        hr[i] = hr[i - 1] * P + str[n - i + 1];
    }
}

// 获取正序子串Hash
ULL getHash(int l, int r) {
    return h[r] - h[l - 1] * p[r - l + 1];
}

// 获取逆序子串Hash
ULL getReverseHash(int l, int r) {
    int n = strlen(str + 1);
    return hr[n - l + 1] - hr[n - r] * p[r - l + 1];
}

// 判断子串是否是回文
bool isPalindrome(int l, int r) {
    return getHash(l, r) == getReverseHash(l, r);
}
```



```
// 求最长回文子串长度
int longestPalindrome(int n) {
    initHash(n);
    int ans = 1;

    // 奇长度回文
    for (int i = 1; i <= n; i++) {
        int left_radius = i - 1;
        int right_radius = n - i;
        int max_radius = min(left_radius, right_radius);

        int l = 0, r = max_radius;
        while (l < r) {
            int mid = (l + r + 1) >> 1;
            if (isPalindrome(i - mid, i + mid)) {
                l = mid;
            } else {
                r = mid - 1;
            }
        }
        ans = max(ans, 2 * l + 1);
    }
    ...
}
```

```
// 偶长度回文
for (int i = 1; i < n; i++) {
    if (str[i] != str[i + 1]) continue;

    int left_radius = i - 1;
    int right_radius = n - i - 1;
    int max_radius = min(left_radius, right_radius);

    int l = 0, r = max_radius;
    while (l < r) {
        int mid = (l + r + 1) >> 1;
        if (isPalindrome(i - mid, i + 1 + mid)) {
            l = mid;
        } else {
            r = mid - 1;
        }
    }
    ans = max(ans, 2 * l + 2);
}

return ans;
}
```

```
int main() {
    strcpy(str + 1, "babad");
    int n = strlen(str + 1);

    cout << "字符串: " << str + 1 << endl;
    cout << "最长回文子串长度: " << longestPalindrome(n) << endl;

    // 测试其他样例
    strcpy(str + 1, "cbbd");
    n = strlen(str + 1);
    cout << "字符串: " << str + 1 << endl;
    cout << "最长回文子串长度: " << longestPalindrome(n) << endl;

    return 0;
}
```

输出结果：

字符串：babad

最长回文子串长度：3

字符串：cbbd

最长回文子串长度：2

算法解析

循环节问题：

- 原理：如果字符串由循环节构成，那么 $n - \text{next}[n]$ 就是最小循环节长度
- 验证：需要检查 $n \% \text{len} == 0$ 确保完整循环
- 时间复杂度： $O(n)$

最长回文子串：

- 原理：对每个可能的中心点，使用二分查找确定最大回文半径
- Hash 优化：通过正序和逆序 Hash 在 $O(1)$ 时间内判断回文
- 时间复杂度： $O(n \log n)$

注意事项

KMP 注意事项

1. 数组下标：通常从 1 开始，方便处理
2. next 数组：注意不要使用 C++ 关键字 `next`
3. 边界处理：确保数组大小足够
4. 跳转逻辑：理解 while 循环的跳转原理

Hash 注意事项

1. 基数选择：使用质数作为基数
2. 模数选择：使用大质数减少冲突
3. 无符号类型：使用 `unsigned long long` 自动取模
4. 双 Hash：重要场合使用双 Hash 提高准确性