问题简介

- **给定字符串S和T，在主串S中寻找子串T。称为模式匹配。其中，T称为模式串。**

问题分类：

- **单串匹配：给定一个模式串，找出前者在后者中的所有位置。**
- **多串匹配：给定多个模式串和一个待匹配串，找出这些模式串在后者中的所有位置。**
- **其他类型：匹配一个串的任意后缀，匹配多个串的任意后缀等等。（这个笔记，没有看太懂。）**

问题方法：

- **暴力算法**$(Brute \quad Force)$
- **字符串哈希。**
- **KMP算法。**

字符串哈希算法。$BKDR \quad hash$

根本思想：

- 构造一个字符串到正整数的一个单射函数。通过对函数值得查询，判读判定字符串是否相等。

具体解决问题：

- 构造多项式哈希函数：
- 优化准确率，时间复杂度。

构造多项式哈希函数：

- 形式：
  $f(x) = \sum_{i=1}^{1} S_{|i|} \times b^{l-i} \, (mod \quad M)$
- **原理：**
- **前面部分为什么是这样的多项式？**
  - 进制系统之间是一个单射关系。这里可以把字符串的意义转变为一个b进制数值表达式。由此，它们的哈希函数值，就是它们十进制上的表示。这样就保证了，随着字符串的不同，它们的哈希函数值不同。查询过程中不会产生多义性。

- **为什么要取一个模?**
    - 可以看到，字符串的长度一旦超过10或者20.哈希函数值就会非常的大，导致出现溢出问题。这样就无法得到理想的哈希函数值，可能导致导致发生哈希碰撞。借助求模运算，可以辅助完成多个角度判断理想哈希值是否相等相等、减少哈希碰撞发生概率。
- **优化角度:**
    - 减小冲突概率:
        - 数的选取上x取133331类型的数。根据统计学可以让冲突的概率最小。
        - 双哈希。
        - 布隆过滤。
        - mod数取较大素数。$专门挑一些特殊的数字来卡。
            - 1000000007最小的十位素数。
            - 最大的十位素数,9999999967
    - 优化时间
        - 不取mod数，如果溢出相当于对2^{64}取模。而且效率比取模运算快。
        - 利用匹配串的前缀哈希值。预处理$O(n)$，查询判断$O(1)$。

代码实例：基于查询字符串任意子串的问题。

```
#include <iostream>
#include <set>
#include <vector>
using namespace std;
typedef unsigned long long ULL;
string s;
const ULL X = 13331; //构建问题情形。查询字符串中任意段落的哈希值。
vector<ULL> h;         //这个相当于前缀为0的哈希值统计；
vector<ULL> p;         //这个是对x的倍数的保存。用这个实现，快速运用原有的前缀的结果来得到目标任意段的哈希值。
void BKDR_hash()            //初始化,这里没有使用双哈希。
{
    h[0] = s[0];
    p[0] = 1;
    for (int i = 1,size=s.size(); i < size; i++)
    {
        h[i] = h[i - 1] * X + s[i];
        p[i] = p[i - 1] * X;
    }
}
ULL get_hash(int l,int r)
{
    return l ? h[r] - h[l-1] * p[r - l + 1] : h[r];
}
int main()
{
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);
    cin >> s;
```

```
    h.resize(s.size());
    p.resize(s.size());
}
```

- **更新**
  - 单哈希不成功时，角度。
    - 用双哈希减少冲突的概率。
    - 另取一个模数时（不是选择自然溢出的方案。）就要注意，数据溢出。将使哈希函数值不可控。，起码在不同的子串中由于$get-hash$对同一个子串，可能用到了不同的X^{l-r-1}等等。即使是同一个子串，也会得到不同的哈希函数值。
    - 例子如下：
      problem相关

KMP (knuth ,Morris, Pratt) $O(n+m)$

- **概述：**
  - **在基于brute force的算法上进行改进。主要角度有，双指针管理一一匹配时，j指针不用动。利用前一次的信息，减少check目标字串和模式串前缀匹配的工作。归结起来就是，根据某些现象规律确认下一个检查的匹配子串子串。减少check模式串和待匹配串前缀的工作。**
  - **如**

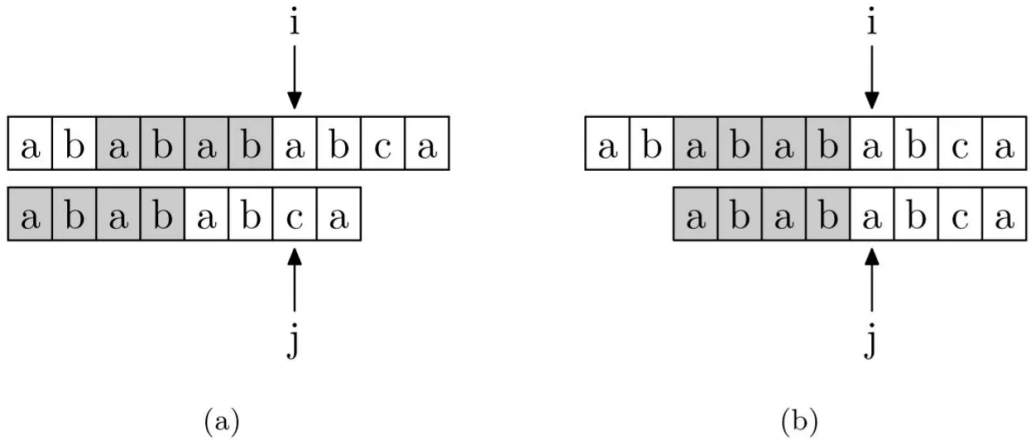

图 1.12: KMP算法示意图

  - $question \quad first: \quad j$ **为什么对齐头匹配失败后，下一步就立刻移到当前图示位置?**

- 这样移动的根据：相关自定义名称：S（检查过程中遇到到第一个匹配的符号时，已经匹配成功的子段。即对于(a)模式串，j指针指向的'c'以前的的子串）S_1，S_2。 S_1 是 S 的子串尾部为 S_{first} 。S_2 为 S 的子串，首部对应了 S_{first} 。其中满足， $S_1 == S_2$ 同时是满足该条件的最大长度子串,且 S_1 长度小于 S
- 根据算法匹配失败后下一步的目标匹配串的前缀就是 S_2.
  - 如果往前挪一点，显然目标匹配串不可能会和模式串相等。 反证法证明如下：
    - 如果匹配成功，说明前缀相同，原j位置之前，的前缀满足上 S_2 相似的定义前提。但是该前缀串长度大于 S_2 矛盾。故前挪没有意义。
  - 如果往后挪，显然可能会漏解。综上所述。该位置为最接近的可能位置。

实现该算法的几个关键。

1. match函数： （对应next数组，最初论文版本是failer函数）。

$$math\,(j) = \begin{cases} \max\,(S_1)\&\&S_1 = S_2 \\ -1\,(not \quad found) \quad (i < j) \end{cases}$$

.

- 说明：S_2 以 P_j 为结尾, P为模板串。

2. 计算得到match数组（next数组）。

- 暴力枚举就是$O(m^3)$.
- 计算$match(j)$，利用好$match(j-1)$. 如下：
  - 1. 令$i = j - 1$.
  - 2. 令$i = match(i)$
  - 3. 则$match(j) <= i + 1$
  - 4. 若$P_j == P_{i+1}$ ， $match(j) = i + 1$;
  - 5. 否则回到过程2直到结束，或者 i 小于0.

示例

```
int m;  //代表模式串的长度。
int match[m];
void build_match( )
{
 match[0] = -1;
    for (int j = 1, i; j < m; j++)
    {
        i = match[j - 1];
```

```
        while (i >= 0 && par[i + 1] != par[j])
            i = match[i];
        if (par[i + 1] == par[j])
            match[j] = i + 1;
        else
            match[j] = -1;
    }
}
```

对上面一些结论证明：

- $point\quad first:$ ———$match(j) <= match(j-1)+1$
  - 若是$match(j) > match(j-1)+1$成立。那么说明$P_1P_2.....P_{match(j-1)}P_{match(j-1)+1}$在字符串$P_{0\sim(j-1)}$。同时作为前缀和后缀。可以推出$match(j-1)$应该更大。矛盾。
- $point\quad second:$———$match(j) < j$
  - 溯源分析这个数组的构造过程。假设P足够大。$match(0) = -1$.即不存在满足题意得条件。下一次构造$match$假设$match_{j_1}$由$match_{j_2}$得到。$match_{j_1} <= match_{j_2}+1$.由于$j_1 > j_2$.所以若$match(j_2)! = j_2$则$match(j_1)! = j_1$.因为match(0)<0.由数学归纳法得$match(j) < j$.
- $point\quad third$: ———当发现$match(j)! = match(j-1)+1$后。为什么下一步关注的是$match(i)?(i = match(j-1))$
  - 令$a = match(j-1), b = match(a), b < c <= a$.证明不可能出现$match(j) = c+1$.反证法如下：假设出现了该情况：r若$match(j) = c+1$.那么$P_{0\sim c}$是 $P_{0\sim a}$的一段前缀。是$P_{j-a\sim j-1}$的一段前缀。由于

    $$P_{0\sim a} = P_{j-a\sim j-1}$$

    所以$P_{0\sim c}$是$P_{0\sim a}$的前后缀。推出$match(a) = c$矛盾。因此，下一个可能的匹配就在 $b+1$ 处.也就是$match(a)$处。

---

3. 使用match的数组。

```
int kmp(string &str, string &par)
{
    const int n = str.length();
    const int m = par.length();
    int *match = new int(m);
    build_match(match, par);
```

```
    int s = 0, p = 0; // s->str , p->par。
    while (p < m && s < n)
    {
        if (str[s] == par[p])
            s++, p++;
        else if (p > 0)
            p = match[p - 1] + 1;
        else
            s++;
    }
    return p == m ? s - m : -1; //跳出循环的时候，m的大小成功匹配子串的末索引加一。
}
```

-

示例：寻找第一个匹配的子串.不面向竞赛式的代码。

```
#include <iostream>
using namespace std;
void build_match(int *match, string &par)
{
    match[0] = -1;
    int m = sizeof(match);
    for (int j = 1, i; j < m; j++)
    {
        i = match[j - 1];
        while (i >= 0 && par[i + 1] != par[j])
            i = match[i];
        if (par[i + 1] == par[j])
            match[j] = i + 1;
        else
            match[j] = -1;
    }
}
int kmp(string &str, string &par)
{
    const int n = str.length();
    const int m = par.length();
    int *match = new int(m);
    build_match(match, par);
    int s = 0, p = 0; // s->str , p->par。
    while (p < m && s < n)
    {
        if (str[s] == par[p])
            s++, p++;
        else if (p > 0)
            p = match[p - 1] + 1;
        else
            s++;
    }
    return p == m ? s - m : -1; //跳出循环的时候，m的大小成功匹配子串的末索引加一。
}
int main()
```

```
{
    string s, p;
    cin >> s >> p;
    int f = kmp(s, p);
    cout << f;
}
```

关于复杂度的证明：

- 显然构造match数组的过程中，一次循环过去，主要时间都是再花在确认$match(j)$上。由于求取它的过程中while循环都是依托于迁移到其它的$match(j)$上。于是最多迭代次数为$\sum match(j)$，其中由于对于每一个$match(j)$最多变化一次。显然求和结果上界是$O(m)$所以最终算法复杂度是$O(n+m)$.

一般竞赛问题提取出来的问题不会仅仅寻找找一个子串。而是寻找所有子串的信息。

问题示例:

---

kmp字符串匹配模板题

题目描述

给出两个字符串 s_1 和 s_2，若 s_1 的区间 $[l,r]$ 子串与 s_2 完全相同，则称 s_2 在 s_1 中出现了，其出现位置为 l。
现在请你求出 s_2 在 s_1 中所有出现的位置。

定义一个字符串 s 的 border 为 s 的一个**非 s 本身**的子串 t，满足 t 既是 s 的前缀，又是 s 的后缀。
对于 s_2，你还需要求出对于其每个前缀 s' 的最长 border t' 的长度。

输入格式

第一行为一个字符串，即为 s_1。
第二行为一个字符串，即为 s_2。

输出格式

首先输出若干行，每行一个整数，**按从小到大的顺序**输出 s_2 在 s_1 中出现的位置。
最后一行输出 $|s_2|$ 个整数，第 i 个整数表示 s_2 的长度为 i 的前缀的最长 border 长度。

个人题解：

```
#include <iostream>
#include <vector>
using namespace std; //记录有匹配串中有多少个子串和模式串匹配。
const int maxn = 1e5 + 10;
int match[maxn];
int n, m;
string str, par;
vector<int> ans;
```

```cpp
void kmp()
{
    ios::sync_with_stdio(false);
    cin.tie(nullptr), cout.tie(nullptr);
    cin >> str >> par;
    n = str.length();
    m = par.length();
    match[0] = -1;
    //初始化match数组。
    for (int j = 1, i; j < m; j++)
    {
        i = match[j - 1];
        while (i >= 0 && par[i + 1] != par[j])
            i = match[i];
        if (par[i + 1] == par[j])
            match[j] = i + 1;
        else
            match[j] = -1; //如果不设置为-1.会漏解?
    }
    //进行有多少位。
    int s = 0, p = 0;
    while (s < n)
    {
        if (p >= m) //如果相同那么会怎么样?
        {
            ans.push_back(s - m);
            p = match[m - 1] + 1; //这里出现了一些困惑;//按照这个板子这里解决不了。
        }
        if (str[s] == par[p])
            s++, p++;
        else if (p > 0)
            p = match[p - 1] + 1;
        else
            s++;
    }
    if (p >= m)
        ans.push_back(s - m);
}
int main()
{
    kmp();
    for (int i = 0; i < ans.size(); i++)
        cout << ans[i] + 1 << '\n';
    for (int i = 0; i < m; i++)
        cout << match[i] + 1 << ' ';
    cout << '\n';
}
```