# String Manipulation

Tong Wei

Soochow University

Jan. 31$^{\text{th}}$, 2016

# KMP Algorithm

# Sample 1

### Sample 1

Given two strings T and P, find the first position where string P occurs in string T.

# A sample

Sample Input:

T = ”abcdefg”

P = ”cde”

Sample Output:

2

# Another sample

Sample Input:

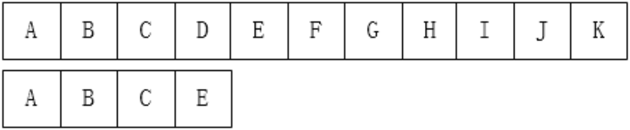
T = "abcdefghijk"

P = "abce"

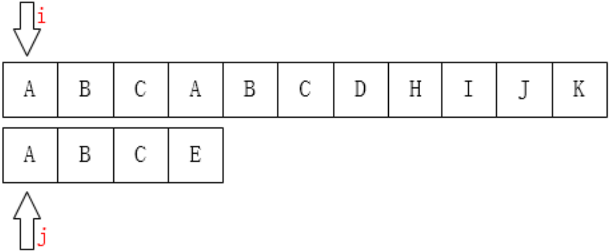
Sample Output:

-1

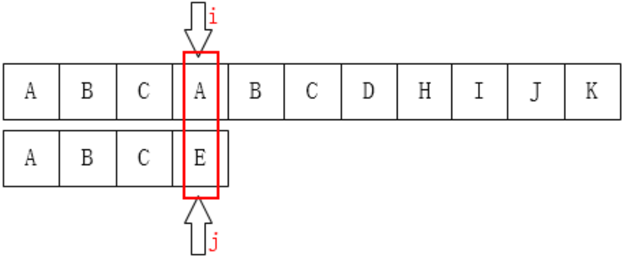
# Naive Solution



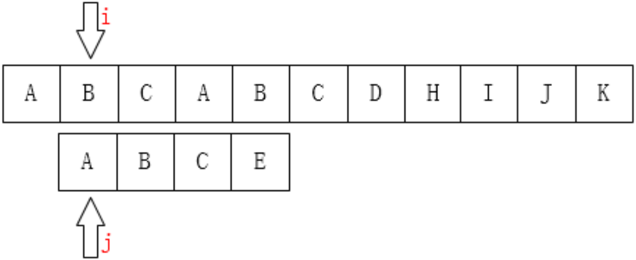
# Naive Solution



# Naive Solution



# Naive Solution



Code 1: Naive Solution

```python
def func(T, P):
  for i in xrange(len(T) - len(P) + 1):
    if T[i:i + len(P)] == P:
      return i
  return -1
```

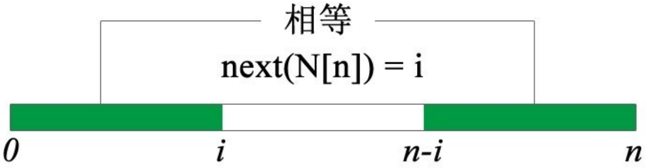
## Naive Solution

Time complexsity is $O(n \times m)$

What if $n$ and $m$ are very large? for example $n = 100000$ and $m = 100000$

# How to improve?

# 寻找最长首尾匹配位置



# Sample

考虑下面几个字符串的最长首尾匹配长度:

S1 = ”abcdefgabc” ==> 3

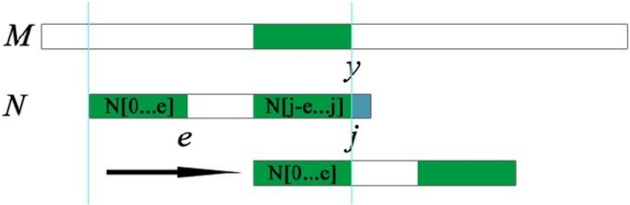
S2 = ”abcdefgab” ==> 2

S3 = ”aaaaa” ==> 4

S4 = ”abcd” ==> 0

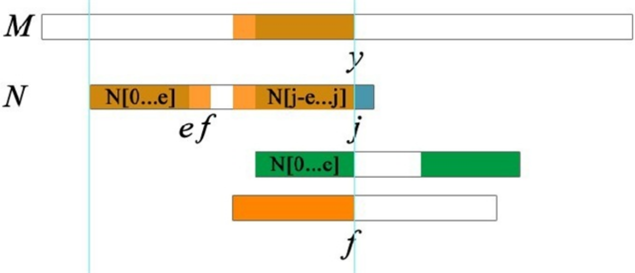
# 证明匹配

如果位置 $j+1$ 不匹配，将 N 后移至首尾相等位置，仍然可以满足匹配，接下来只需查看 N[e+1] 与 M[y+1] 是否相等即可



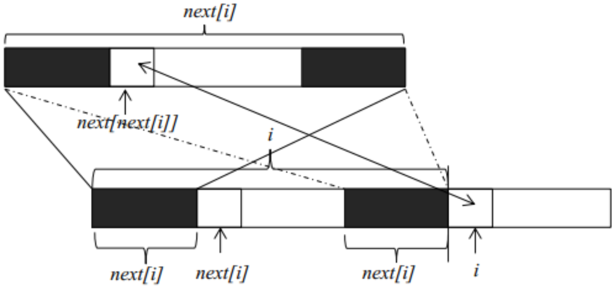
# 证明最优

用反证法，假设存在 f，f>e，满足 N[0...f] = M[y-f...y]，即其匹配位置出现在更早的位置，由于 M[y-j...y] = N[0...j]，M[y-f...y] = N[j-f...j]，则满足 N[j-f...j] = N[0...f]，则 e 就不是最长的首尾匹配点，与原假设不符



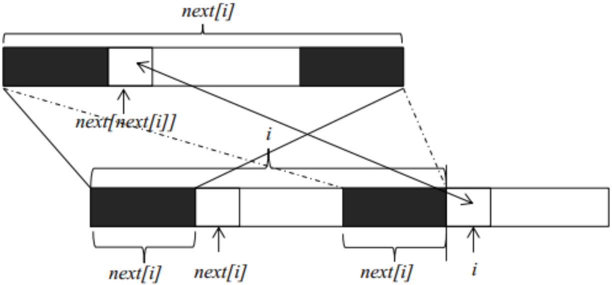
# next 数组计算

假设我们现在已经求得 next[1]、next[2]、⋯⋯next[i]，分别表示长度为 1 到 i 的字符串的前缀和后缀最大公共长度，现在要求 next[i+1]



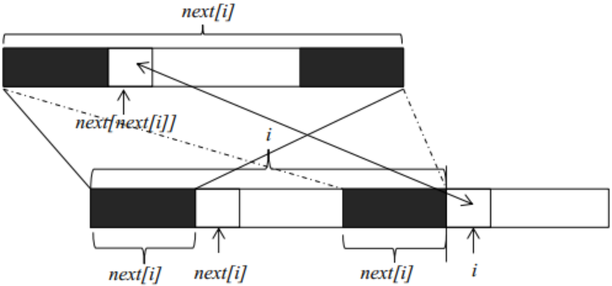
# next 数组计算

如果位置 i 和位置 next[i] 处的两个字符相同，则 next[i+1] 等于 next[i] 加 1。



# next 数组计算

如果两个位置的字符不相同，我们可以将长度为 next[i] 的字符串继续分割，获得其最大公共长度 next[next[i]]，然后再和位置 i 的字符比较。直到字符串长度为 0 为止



# next 数组计算

S = "ABRACADABRA"

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P[i] | A | B | R | A | C | A | D | A | B | R | A | 无 |
| next[i] | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 2 | 3 | 4 |

Table : example of calculating next array

## Code 2: Golden Solution

```
void getNext(const string &P) {
  int m = P.length();
  next[0] = next[1] = 0;
  int j = 0;
  for (int i = 1; i < m; ++i) {
    while (j && P[j] != P[i]) j = next[j];
    if (P]j] == P[i]) ++j;
    next[i + 1] = j;
  }
}
```

## Code 3: Golden Solution

```
int find(const string &T, const string &P) {
  int n = T.length(), m = P.length();
  getNext(P);
  int j = 0;
  for (int i = 0; i < n; ++i) {
    while (j && P[j] != T[i]) j = next[j];
    if (P]j] == T[i]) ++j;
    if (j == m) return i - m + 1;
  }
  return -1;
}
```

# Golden Solution

Time complexsity is $O(n+m)$ （提示：使用摊还分析）

# Hash (字符串哈希)

## Rabin-Karp Algorithm

1. $S = s_1 s_2 s_3 \dots s_m$
2. $H(S) = (s_1 b^{m-1} + s_2 b^{m-2} + s_3 b^{m-3} + \dots + s_m b^0) \mod M$ （可以将 S 看作 b 进制的数，另外实际使用中常用自然溢出来代替取模）
3. $H(S[k+1..k+m]) = (H(S[k..k+m-1]) \times b - s_k b^m + s_{k+m}) \mod M$

# Sample

S = “abcde“, 求 $S$ 的所有后缀的 Hash 值。

$$\begin{cases} H(4) & = & s[4] \\ H(3) & = & s[4]x & + s[3] \\ H(2) & = & s[4]x^2 & + s[3]x & + s[2] \\ H(1) & = & s[4]x^3 & + s[3]x^2 & + s[1]x & + s[0] \\ H(0) & = & s[4]x^4 & + s[3]x^3 & + s[2]x^2 & + s[1]x & + s[0] \end{cases}$$

### Sample 2

Given two strings T and P, find the first position where string P occurs in string T.

## Code 4: Hash Function

```
typedef unsigned long long ll;  const ull B = 131;
int contain(const string &a, const string &b) {
  int n = S.length(), m = b.length();
  if (n > m) return false;
  ull  t = 1;
  for (int i = 0; i < n; ++i) t *= B;
  ull ah = 0, bh = 0;
  for (int i = 0; i < n; ++i) ah = ah * B + a[i];
  for (int i = 0; i < n; ++i) bh = bh * B + b[i];
  for (int i = 0; i + n <= m; ++i) {
    if (ah == bh) return i;
    if (i + n < m) bh = bh * B + b[i + n] - b[i] * t;
  }
  return -1;
}
```

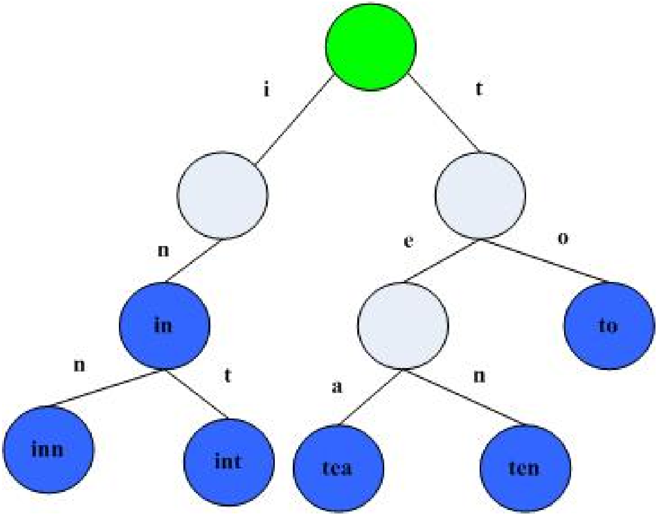
# Trie (字典树, 前缀树)

# How to read?

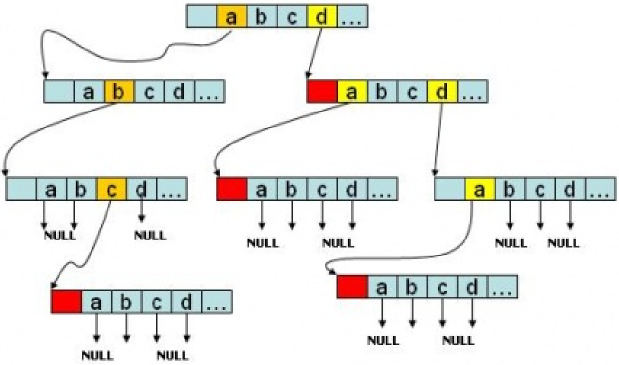
Trie: /tra /

Its pronunciation is same with word “try”.

{tea, ten, to, in, inn, int}



# Another picture



# Function of Trie

1. 字符串检索：检索/查询功能是 Trie 树最原始的功能。思路就是从根节点开始一个一个字符进行比较
2. 词频统计：Trie 树常被搜索引擎系统用于文本词频统计。
3. 前缀匹配：trie 树前缀匹配常用于搜索提示。如当输入一个网址，可以自动搜索出可能的选择。当没有完全匹配的搜索结果，可以返回前缀最相似的可能。

## Code 5: Trie

```
struct Trie {
  int ch[maxNode][sigma_size], val[maxNode], sz;
  Trie() { sz = 1; memset(ch[0], 0, sizeof(ch[0])); }
  void insert(char *s) {
    int u = 0, n = strlen(s);
    for (int i = 0; i < n; ++i, u = ch[u][c]) {
      int c = s[i] - 'a';
      if (!ch[u][c]) {
        memset(ch[sz], 0, sizeof(ch[sz]));
        val[sz] = 0; ch[u][c] = sz++;
      }
    }
    val[u] = 1;
  }
};
```

# Others (其他)

# Several more string algorithms

1. Manacher Algorithm
2. Suffix Array
3. Extended KMP Algorithm
4. Aho-Corasick Automachine
5. Suffix Automachine

# The end
# Thank you!