第三章 串

概念：

串，空串，串的长度

子字符串：任一连续子序列

操作

参考 C++ STL 的string模板：

```
class String{
    public:
        String();
        bool empty();
        String substr(int pos);
        String operator+(String& a);
        int find(char c);
        void insert(int pos, char c);
        void replace(int pos, String& a);
};
```

最主要的操作：

子串；

查找子串；

存储方式：

- 数组静态
- 块状存储

串匹配算法(重点)

1. 暴力算法(BF):

每次对Text串中可能的位置进行枚举, 一位一位的与模式串比较, 直到完全匹配。

平均时间复杂度$O(N^2)$, 滑动窗口为1的比较方式

最大比较次数(每次都比到最后一个失败) $(n-m+1)m$次, n, m为文本串，模式串的长度

2. Karp-Rabin指纹比较法：

哈希策略

哈希值的计算在移动过程中容易修改, 耗时$O(1)$

而只有当哈希值完全匹配，才会进行真正的逐字符比对;

最坏情况依然是$O(N^2)$， 但通常效果很好

3. KMP (考试最爱考的next表)

文本串指针不回头!

部分匹配表 PMT

$next[i]$的意义：

●代表的是$P[0:i-1]$这一子串中最长公共前后缀的长度

就是 前缀完全等于后缀, 这样的字符串最长的那个的长度;

而有了这样的信息, 模式串指针在匹配失败后就可以尽可能地避免比较完全相同的前缀部分, 从而尽可能地减少回退!!!

例子:

字符串	A	B	A	A	B	A
idx	0	1	2	3	4	5
next[]	-1	0	0	1	1	2

比如这样的一个字符串比较

```
    A B A A B X A A A A A X X X     Text
    A B A A B A                     Pattern
              ^ = 5
          A B A A B A
              ^ = 2                 减少回退
```

一个套路代码：

```
int* get_next(string P){
    int i = 0, j = -1;
    int m = P.size();
    int* next = new int[m];

    while(i < m - 1){
        if(j < 0 || P[i] == P[j]){
            ++i;
            ++j;
            next[i] = j; // 整体失配或者相同的情况合并
        }
        else{
            j = next[j]; // 不断回退寻找合适的长度
        }
    }
    return next;
}
```

KMP最大比较次数： $2n-1$

时间复杂度：$O(m+n)$

改进优化：相同字符可以继续向前扩展

$next[i] = (P[i] != P[j])?j : next[j]$

4. BM算法

good-suffix & bad-charcter 策略

好后缀，坏字符

坏字符：对应失配字符出现的最后位置

好后缀：对应最长匹配的前后缀长度

最坏情况：$3n$次比较

时间复杂度 $O(n)$

最好$O(n/m)$