

# 1 垃圾邮件过滤

## Requirement and notification

1. 解释代码逻辑和布隆过滤器中哈希函数的选取。
2. 分析问题中两种算法的复杂度。

实现一个简易的垃圾邮件过滤系统

第一问的黑名单为blacklist\_1.txt，待检测名单为underfilt\_1.txt，最终输出为output\_1.txt.

第二问的黑名单为blacklist\_2.txt，待检测名单为underfilt\_2.txt，最终输出为output\_2.txt.提交的文件无需包含这些文件，但代码中文件的读写请使用同样的文件名（测试时文件会与代码处于同一目录下），例如

```
1 FILE* pf1 = fopen("blacklist_1.txt", "r");
2 FILE* pf2 = fopen("underfilt_1.txt", "r");
3 FILE* pf3 = fopen("output_1.txt", "w");
```

最终会使用另外的测试样例，但数据规模一致。

如果使用Visual Studio开发，请在提交代码时删除

或将scanf\_s()等全部改回scanf()等形式。

```
#define _CRT_SECURE_NO_WARNINGS
```

### 1. 使用KMP算法进行过滤

一份邮件地址黑名单，里面包含了若干个已被识别为垃圾邮件发送者的邮件地址。

但是这份名单遭到了污染，**邮件地址之间存在若干无效字符，且没有空格隔开。**

给定一个邮件名单，请**使用KMP算法**识别名单中的邮件地址是否在黑名单中，

并输出在黑名单中的地址，

将其写入txt文件中。

fopen 文件打开方式详解

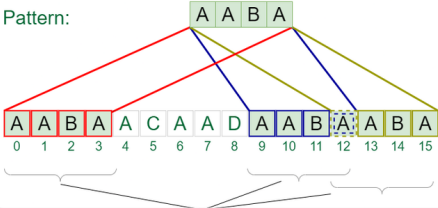
<https://blog.csdn.net/shulianghan/article/details/117307116>

fgets()函数的详解-使用技巧-C语言基础

[https://blog.csdn.net/Devour\\_/article/details/110955333](https://blog.csdn.net/Devour_/article/details/110955333)

参考下面的代码写出部分匹配表函数

```
void computeLPSArray(const char *pat, int M, int *lps) 和 int  
KMPSearch(const char *pat, const char *txt)
```



<https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/>

### KMP Algorithm for Pattern Searching - GeeksforGeeks

A Computer Science portal for geeks. It contains well written, well thought and well explained computer science and programming articles, quizzes and...

main函数中先打开blacklist\_1.txt并做异常处理

## 代码逻辑

然后处理黑名单文件

### 1. 确定文件大小

**定位到文件末尾：** `fseek(blacklist_file, 0, SEEK_END);`

将文件指针移动到文件的末尾，从而可以使用 `ftell` 函数获取文件的大小。

**获取文件大小：** `long blacklist_file_size = ftell(blacklist_file);`

`ftell` 函数获取当前文件指针的位置（即文件的大小，以字节为单位）。

**重新定位到文件开头：** `fseek(blacklist_file, 0, SEEK_SET);`

### 2. 分配足够的内存

根据文件的大小分配足够的内存来存储文件内容。

`+1` 是为了存储字符串结束符 `\0`，确保读取到的文件内容可以作为一个有效的C字符串使用。

### 3. 读取文件内容到内存中

**读取文件内容：** `fread(blacklist_content, 1, blacklist_file_size, blacklist_file);`

- 通过 `fread` 函数从文件中读取数据到分配的内存缓冲区 `blacklist_content` 中。
- 读取 `blacklist_file_size` 个字节，每次读取1个字节。

### 4. 添加字符串结束符

在读取到的内容末尾添加字符串结束符 `\0`，将其转换为一个有效的C字符串。

这样，`blacklist_content` 可以像普通字符串一样进行处理和操作。

### 5. 关闭文件

关闭文件，释放文件相关的资源。

### 6. 接着处理待检测的邮件地址文件里的内容

### 7. 打开待检测文件，然后异常检测

为待检测邮件地址定义字符数组存每一行读取内容和计数器便于处理

## 8. 外层的while循环在fgets读到一行时继续循环

fgets从文件中读一行内容，然后目标缓冲区 `underfit_emails[underfit_count]`，用于存储读取的字符串

然后读取的最大字符数（包括终止符），这里是

```
sizeof(underfit_emails[underfit_count])
```

最后是文件指针underfit\_file

```
while (fgets(...))
```

，只要 `fgets` 成功读取到一行内容，循环就继续

移除换行符（`\n->\0`）的同时，计数器+1

## 9. 打开输出文件，异常处理

10. 用KMPSearch算法检查待检测邮件是否在黑名单内，并用fprintf输出在underfit\_emails中

11. 关闭文件，释放空间

## 复杂度分析

- 时间

fgets逐行读文件内容，时间复杂度 $O(L)$ ,最坏情况 $O(N)$ .

L：行的长度，N：所有行的总长度

strcspn查找字符串中第一个换行符位置，时间复杂度 $O(L)$ ,最坏情况 $O(N)$

计数器 $O(1)$

打印输出邮件地址 $O(N)$

最后时间复杂度是 $O(N)$

- 空间

内存分配：固定大小数组，占用特定空间字节

局部变量：计数器，可忽略不计

最后空间复杂度是 $O(1)$

## 2. 使用布隆过滤器进行过滤

实际的邮件系统中邮件的吞吐量很大，如果每次查找都需要遍历黑名单，其耗时是不可接受的。

布隆过滤器是一种常见的垃圾邮件过滤方法

(1) 布隆过滤器的结构由k个哈希函数与一个位图（bitmap）构成。

(2) 过滤器添加新元素时，每一个字符串会经过k个不同的哈希函数生成k个哈希值，

这k个哈希值依次取模位图的长度得到索引，

位图索引对应的位置全部修改为1。

(3) 查询时，字符串同样经过k个哈希函数生成k个哈希值，依次取模位图的长度得到索引，若位图索引对应的位置全部为 1，则【可能存在】，反之则【一定不存在】。

手上有一份黑名单，黑名单的每一行都是一个邮件地址

编写代码实现k=5,位图长度8192的布隆过滤器（哈希函数需自行选择），对给定的邮件名单进行过滤。

并输出存在于黑名单中的邮件地址数量。

布隆(Bloom Filter)过滤器

[https://blog.csdn.net/qg\\_41125219/article/details/119982158](https://blog.csdn.net/qg_41125219/article/details/119982158)

## 2.1 代码逻辑和哈希函数选取

(1) 包含头文件和定义常量与结构体

(2) 实现 MurmurHash3和DJB2哈希函数，生成32位的哈希值，提高布隆过滤器的哈希质量和分布均匀性

- 用多种不同类型的哈希函数（如MurmurHash3和DJB2）可以提高哈希值的分布均匀性，减少冲突和误判率。MurmurHash3 高效性和低冲突，DJB2 的简单和良好分布。
- 给MurmurHash3提供不同的种子值，可以生成多个不同的哈希值，进一步增强布隆过滤器的效果。

<https://en.wikipedia.org/wiki/MurmurHash>

[en.wikipedia.org](https://en.wikipedia.org/wiki/MurmurHash)

```

algorithm Murmur3_32 is
  // Note: In this version, all arithmetic is performed with unsigned 32-bit integers.
  //       In the case of overflow, the result is reduced modulo 232.
  input: key, len, seed

  c1 ← 0xcc9e2d51
  c2 ← 0x1b873593
  r1 ← 15
  r2 ← 13
  m ← 5
  n ← 0xe6546b64

  hash ← seed

  for each fourByteChunk of key do
    k ← fourByteChunk

    k ← k × c1
    k ← k ROL r1
    k ← k × c2

    hash ← hash XOR k
    hash ← hash ROL r2
    hash ← (hash × m) + n

  with any remainingBytesInKey do
    remainingBytes ← SwapToLittleEndian(remainingBytesInKey)
    // Note: Endian swapping is only necessary on big-endian machines.
    //       The purpose is to place the meaningful digits towards the low end of the value,
    //       so that these digits have the greatest potential to affect the low range digits
    //       in the subsequent multiplication. Consider that locating the meaningful digits
    //       in the high range would produce a greater effect upon the high digits of the
    //       multiplication, and notably, that such high digits are likely to be discarded
    //       by the modulo arithmetic under overflow. We don't want that.

    remainingBytes ← remainingBytes × c1
    remainingBytes ← remainingBytes ROL r1
    remainingBytes ← remainingBytes × c2

    hash ← hash XOR remainingBytes

  hash ← hash XOR len

  hash ← hash XOR (hash >> 16)
  hash ← hash × 0x85ebca6b
  hash ← hash XOR (hash >> 13)
  hash ← hash × 0xc2b2ae35
  hash ← hash XOR (hash >> 16)

```

<https://theartincode.stanis.me/008-djb2/>

## 008 - djb2 hash

A very simple string hashing algorithm.

布隆过滤器的误判率  $\lambda$  可以通过以下公式计算：

$$P = \left(1 - e^{-kn/m}\right)^k$$

其中：

- k是哈希函数的数量。
- n是插入的元素数量。

- $m$ 是位数组的大小。

$$\frac{kn}{m} = \frac{5 \cdot 1000}{8192 \cdot 8} = \frac{5000}{65536} \approx 0.0763$$

$$1 - e^{-0.0763} \approx 1 - 0.9266 = 0.0734$$

$$(0.0734)^5 \approx 0.000165$$

$p$ 约为 0.0165，即 1.65%。

(3) `add_to_bloom_filter` 函数，将字符串添加到布隆过滤器中。用5个哈希函数生成哈希值，将对应的位图位置设置为1。

计算字符串 `str` 的长度 `len`。

使用 4 个不同种子的 MurmurHash3 哈希函数和一个 djb2 哈希函数计算字符串的 5 个哈希值，结果存储在 `hashes` 数组中。每个哈希值都对 `BITMAP_SIZE * 8` 取模，以确保结果在位图范围内。

遍历 `hashes` 数组，将对应的位图位置设置为 1：

- `hashes[i] / 8` 计算出哈希值对应的字节索引。
- `1 << (hashes[i] % 8)` 计算出在字节中的具体位置。
- `|=` 操作将对应的位置位设置为 1。

(4) `check_bloom_filter` 函数，检查字符串是否在布隆过滤器中。用5个哈希函数生成哈希值，检查对应的位图位置，如果所有位置都是1，则返回1（可能存在），否则返回0（一定不存在）。

计算字符串 `str` 的长度 `len`。

使用与 `add_to_bloom_filter` 相同的哈希函数和种子，计算字符串的 5 个哈希值，结果存储在 `hashes` 数组中。

遍历 `hashes` 数组，检查每个哈希值对应的位图位置是否都为 1：

- `filter->bitmap[hashes[i] / 8]` 访问哈希值对应的字节。
- `1 << (hashes[i] % 8)` 计算出在字节中的具体位置。
- `&` 操作检查对应的位置位是否为 1。如果其中任何一个位置不为 1，则返回 0，表示该字符串一定不存在于过滤器中。
- 如果所有位置都为 1，返回 1，表示该字符串可能存在于过滤器中。

(5)声明并初始化 `BloomFilter` 结构体实例 `filter`，将其位图中的所有字节设置为0。

(6)打开黑名单文件 `blacklist_2.txt`。

异常处理：检查文件是否成功打开，失败则输出错误信息并返回1。

逐行读取每个邮件地址并添加到布隆过滤器中。

(7)打开待检测的邮件文件 `underfilt_2.txt`。

异常处理：检查文件是否成功打开，失败则输出错误信息并返回1。

逐行读取每个邮件地址，检查其是否在布隆过滤器中，并统计匹配的数量。

(8)将匹配的邮件地址数量写入输出文件 `output_2.txt`，并在控制台打印匹配数量（Number of emails in the blacklist: %d(count)）。

## 2.2 分析算法的复杂度

### ①添加元素的复杂度

每个哈希函数的时间复杂度为 $O(\text{len})$ ,  $\text{len}$  是字符串的长度

添加一个元素的时间复杂度为  $O(k \cdot \text{len})$ ,  $k$ 是哈希函数的数量

空间复杂度主要由位图大小决定，为  $O(m)$ ,  $m=8192 \cdot 8$  位，是位图大小

### ②查询元素的复杂度

查询一个元素需要计算5个哈希函数，并检查5个位图位置，时间复杂度也是  $O(k \cdot \text{len})$

查询操作不需要额外的空间，因此空间复杂度为  $O(1)$

### ③整体复杂度

总体时间复杂度：设有  $n$  个元素需要添加和查询

添加操作的总时间复杂度为  $O(n \cdot k \cdot \text{len})$

查询操作的总时间复杂度为  $O(n \cdot k \cdot \text{len})$

总体时间复杂度：  $O(n \cdot k \cdot \text{len})$

总体空间复杂度：主要由位图大小决定，为  $O(m)$