



# 第一部分

## 线性结构应用



# 线性结构应用实例

---

- 迷宫老鼠问题：
  - 深度优先搜索
    - 回溯
      - 矩阵（二维数组），栈、一维数组
- 递归



## 思考

---

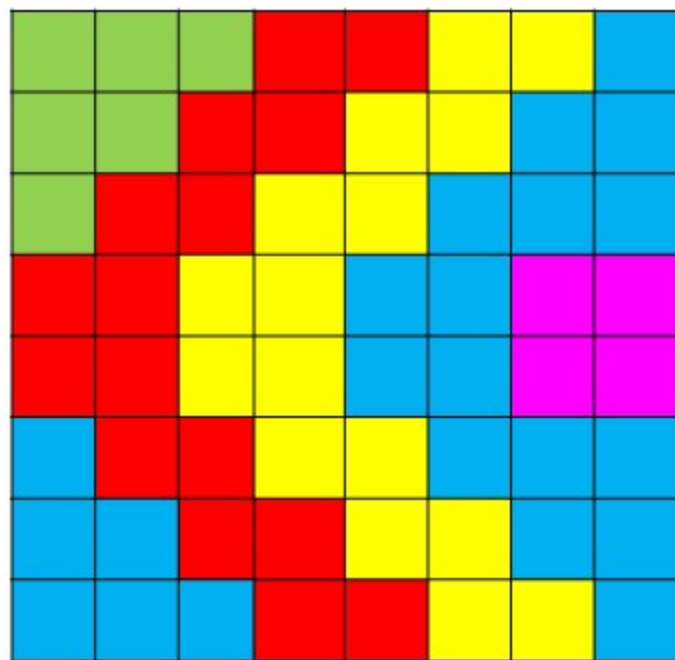
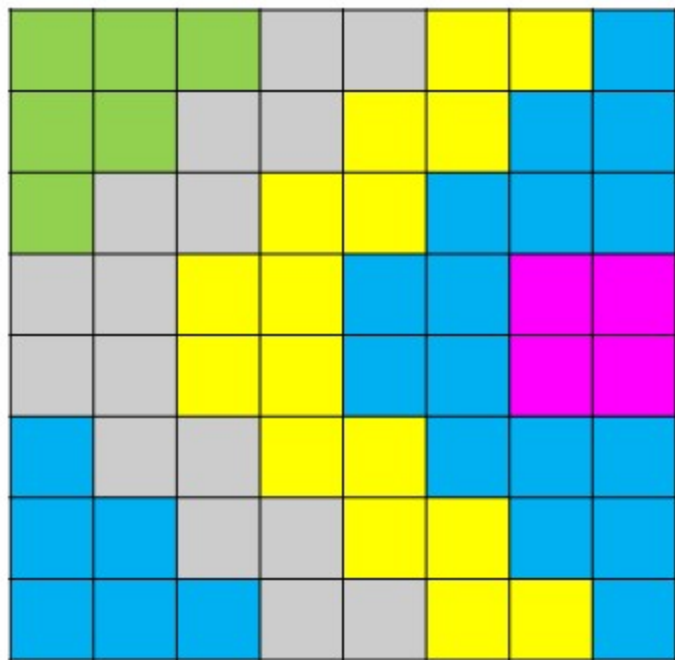
- 如何得到迷宫入口到迷宫出口的最短路径？





- 假设以二维数组 $g[1..m, 1..n]$ 表示一个图像区域， $g[i, j]$ 表示像素点 $(i, j)$ 所具颜色，其值为从0到 $k$ 的整数，图像区域的基本操作：  
 $\text{replace}(i_0, j_0, c)$ ——完成完成将 $(i_0, j_0)$ 所在同色区域的颜色置换为 $c$ ，，约定**和 $(i_0, j_0)$ 同色的上、下、左、右的相邻点为同色区域的点**，写出 $\text{replace}(i_0, j_0, c)$ ——的设计思路和算法。

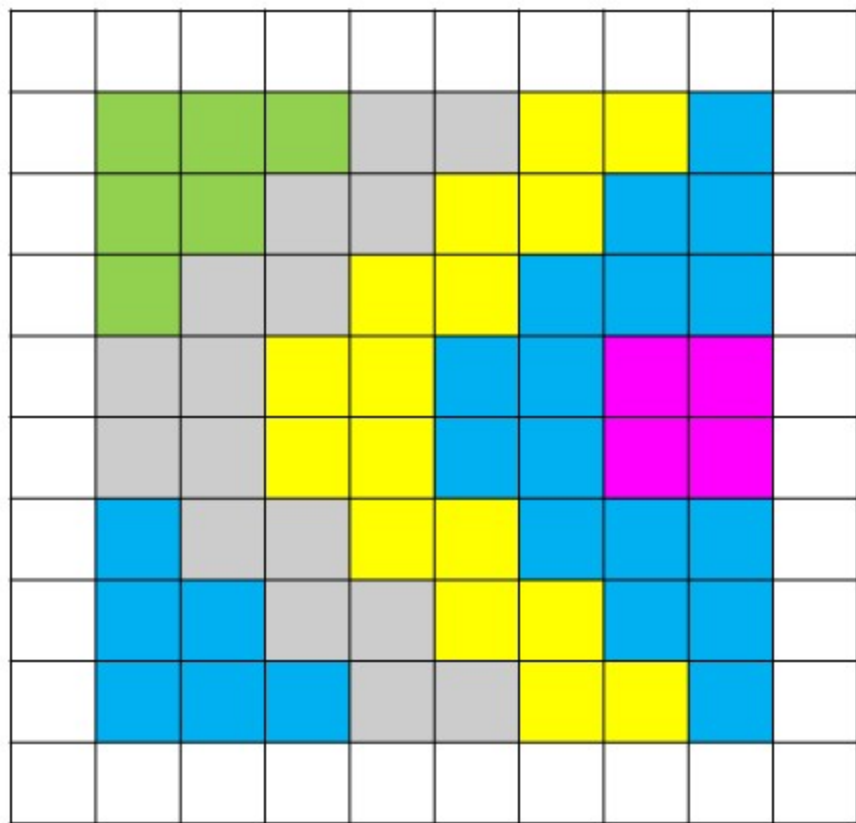
- $g[1..m, 1..n]$
- $(i, j) = (4, 1)$   $c = \text{“红色”}$  的颜色值





## 简化算法的处理

- 为了避免在处理内部像素和边界像素时存在差别，可以在图像的周围增加一圈特别像素（值为-1或 $k+1$ ）。







## 设计思路

replace(i0, j0, c)的设计思路:

假设(i0, j0, c)已验证有效性。

- 如果(i0, j0)的颜色=c, 则结束;
- 替换**图像**中**像素**(i0, j0)的颜色;
- 替换与(i0, j0)**相邻**的所有同色像素(**1-间距同色像素**);
- 替换与**1-间距同色像素**相邻的所有同色像素(**2-间距同色像素**);
- 替换与**2-间距同色像素**相邻的所有同色像素(**3-间距同色像素**);
- .....

直到找不到新的、相邻的同色像素为止。





- 用一个队列用来跟踪这样的像素：
  - 该像素本身颜色已被替换，而它的相邻同色像素尚未被替换。



# 数据结构选择

- 像素位置表示:
  - class `position` {  
    private:  
        int row; //像素的行坐标  
        int col; //像素的列坐标}
- 图像表示: 二维数组: `g[0..m+1,0..n+1]`
- 相邻像素位置: 采用数组 `offset` 来确定与一个给定像素相邻的像素。
- 队列 `q`: 保存本身颜色已被替换, 而它的相邻同色像素尚未被替换的像素。



## 设计思路

replace(i0, j0, c)的设计思路(结构化描述):

替换(i0,j0像素, (i0,j0)入队列保存;

当队列不空时, 循环:

{当前像素←队列出队元素;

依次处理当前像素的相邻像素,

(if 相邻像素的颜色=初始像素颜色)

{替换相邻像素颜色;

相邻像素位置入队列保存; }

}



# 置换同色区域颜色算法

```
void replace(i0, j0, c)
```

```
{ //将(i0, j0)所在同色区域的颜色置换为c
```

```
    数组 offsets初始化;
```

```
    设置一堵-1值像素“围墙”，把图像包围起来;
```

```
    初始化队列q;
```

```
    初始像素位置pixel_s=(i0,j0);
```

```
    初始像素颜色c_o=g[i0][j0];
```

```
    替换pixel_s上的颜色为c; pixel_s入队列q;
```

```
    while (队列q不为空)
```

```
        {here←从队列q中取出一个相邻位置未被处理的位置
```

```
        //处理当前位置here的相邻位置
```

```
        for (int i=0; i<相邻位置数; i++)
```

```
            {选择here的下一个的相邻位置nbr;
```

```
            if (g[nbr.row][nbr.col]==c_o)
```

```
                {g[nbr.row][nbr.col]=c;//替换颜色
```

```
                将位置nbr入队列q;}
```

```
        }
```

```
}
```



## 犯罪团伙问题

- 警察抓到了 $n$ 个罪犯，警察根据经验知道他们属于不同的犯罪团伙，却不能判断有多少个团伙，但通过警察的审讯，知道其中的一些罪犯之间相互认识，已知同一犯罪团伙的成员之间直接或间接认识。有可能一个犯罪团伙只有一个人。请你根据已知罪犯之间的关系，确定犯罪团伙的数量和每个犯罪团伙的罪犯。





## 问题分析

- 开始把 $n$ 个人看成 $n$ 个独立集合(团伙)。
- 每遇到两个相互认识的罪犯 $i$ 和 $j$ , 查找 $i$ 和 $j$ 所在的集合 $p$ 和 $q$ , 如果 $p$ 和 $q$ 是同一个集合, 不作处理; 如果 $p$ 和 $q$ 属于不同的集合, 则合并 $p$ 和 $q$ 为一个集合。
- 最后统计集合的个数即可得到犯罪团伙的数量; 每个集合中的元素则为犯罪团伙的成员。
- 集合(团伙): 等价类
- $i$ 和 $j$ 等价: 罪犯 $i$ 和 $j$ 属于同一个集合(团伙)



# 并查集

```
Class UnionFind{
```

```
public:
```

```
    UnionFind(int numberOfElements)
```

```
        //构造函数; n=numberOfElements,
```

```
        //初始化n个类, 每个类仅有一个元素
```

```
    ~UnionFind(); //析构函数
```

```
    void unite(int classA, int classB)
```

```
        //合并类classA和类classB, 假设classA!=classB
```

```
    int find(int theElement)
```

```
        //搜索包含元素theElement的类
```

```
protected:
```

```
    ..... //集合的存储结构
```

```
    n;     //元素个数
```





# Top K 问题

- 搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来，每个查询串的长度为1-255字节。假设目前有一千万个记录，请你统计**最热门的10个查询串**，要求使用的内存不能超过1G。
- 注：这些查询串的重复度比较高，虽然总数是1千万，但如果除去重复后，不超过**3百万**个。一个查询串的重复度越高，说明查询它的用户越多，也就是越热门。





## 问题解析

---

- ◆ 1. 统计每个查询串出现的次数
- ◆ 2. 根据统计结果，找出Top 10



# 1. 查询串统计

---

- (1) .直接排序法
  - 对日志里面的所有查询串进行排序
  - 遍历排好序的查询串，统计每个查询串出现的次数
- 问题：
  - 一千万条记录，每条记录是255Byte，占据2.375G内存
  - 内存不能超过1G



# 1. 查询串统计


---

- (2) 外排序法
  - 对日志里面的所有查询串进行排序
  - 遍历排好序的查询串，统计每个查询串出现的次数
- 复杂度：
  - ◆ 排序的时间复杂度是 $O(N \log N)$
  - ◆ 遍历的时间复杂度是 $O(N)$
  - ◆ 因此总体时间复杂度 $O(N + N \log N) = O(N \log N)$



# Top K 问题

- 搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来，每个查询串的长度为1-255字节。
- 假设目前有一千万个记录，请你统计最热门的10个查询串，要求使用的内存不能超过1G。
- 注：这些查询串的重复度比较高，虽然总数是1千万，但如果除去重复后，不超过3百万个。一个查询串的重复度越高，说明查询它的用户越多，也就是越热门。



300万条记录，每条记录是255Byte，占据0.712G内存



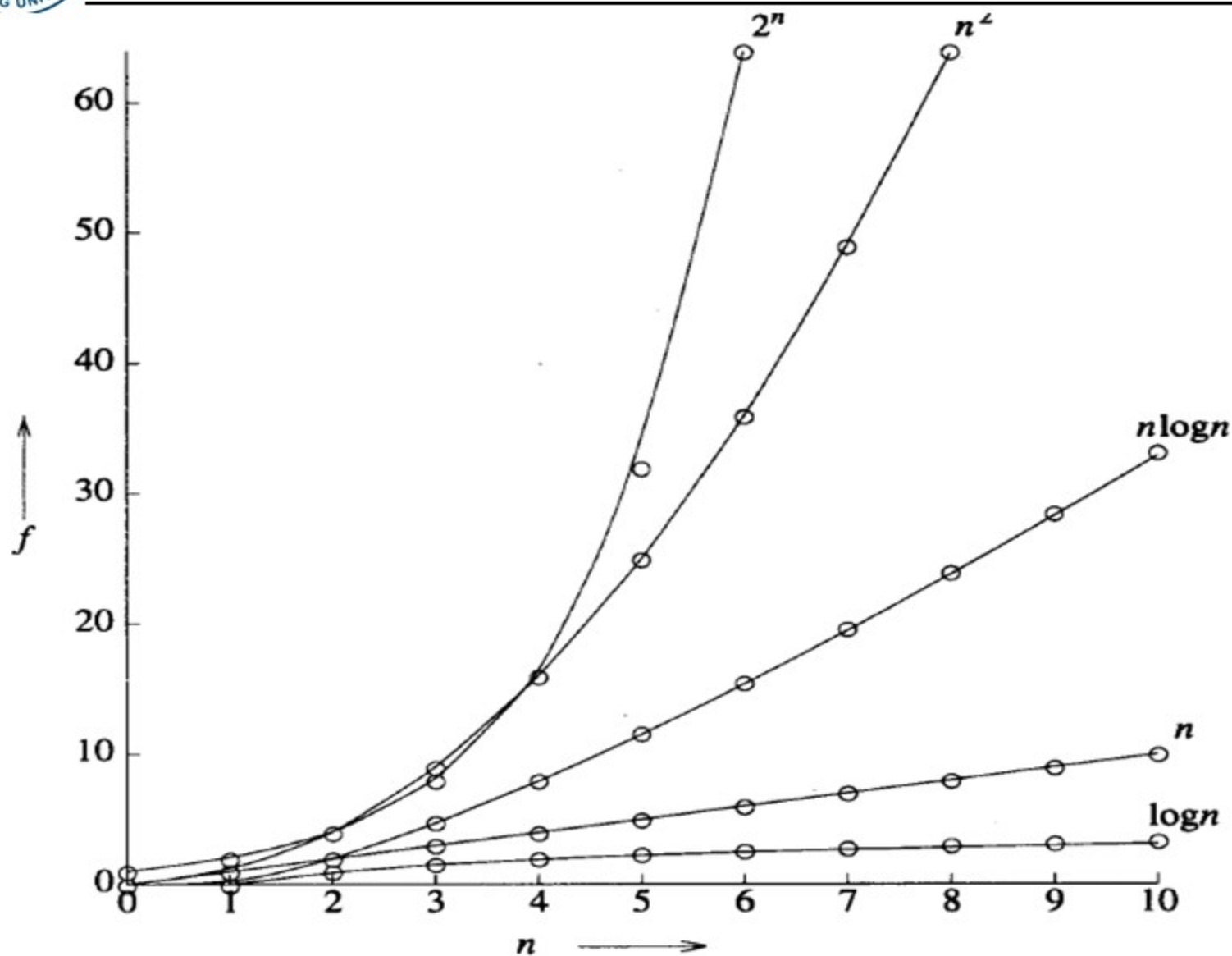


# 查询串统计

- (3) 散列表
  - Key为查询串字符串
  - Value为该查询串出现次数
- 设计思想:
- 每次读取一个查询串
  - ◆ 如果该字符串不在散列表Table中, 那么加入该字符串, 并且将Value值设为1;
  - ◆ 如果该字符串在散列表Table中, 那么将该字符串的计数加1。
- 复杂度:
  - 每一个查询串都是常量级, 总计 $O(N)$



# 查询串统计算法对比







# 算法复杂度对比

$\log n$	$n$	$n \log n$
0	1	0
10	1024	10240
20	1048576	20971520
22	3000000	64549593
23	10000000	232534966
26	100000000	2657542476

使用散列表，读写磁盘操作次数更少



## 2. 找出Top10

---

- (1) . 普通排序

- ◆300万条记录，可全部放在内存中

- ◆排序算法的时间复杂度是 $N\log N$

- 算法思想:

- ◆选择排序，名次排序，冒泡排序，堆排序，快速排序.....



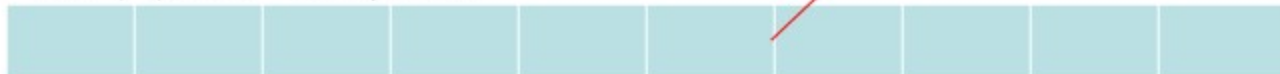
## 2. 找出Top10

### 有序数组搜索与插入

#### ■ (2). 部分排序

◆ 维护一个10个大小的数组

#### ■ 算法思想:



◆ 初始化放入10个查询串，按照统计次数由大到小排序

◆ 遍历这300万条记录

◆ 每读一条记录就和数组最后一个查询串比较

◆ 如果小于这个查询串，那么继续遍历

◆ 否则，将数组中最后一条数据淘汰，加入当前的查询串。

◆ 当所有数据遍历完成，数组：Top10

#### ■ 时间复杂度:

◆ 算法的最坏时间复杂度是 $N \times K$ ，其中 $K$ 是指top多少



# 找出Top10

## ■ (3) . 最小堆

◆ 维护一个K(该题目中是10)大小的最小堆

## ■ 算法思想:

◆ 初始化放入10个查询串, 按照统计次数建最小堆

◆ 遍历这300万条记录

◆ 每读一条记录就和根元素查询串比较

◆ 如果小于这个查询串, 那么继续遍历

◆ 否则, 当前的查询串替换堆的根。

◆ 当所有数据遍历完成, 堆: Top10

## ■ 时间复杂度:

◆ 算法最坏时间复杂度是 $N \cdot \log K$ , 其中K是指top多少



# 找出Top10

- (4). 最大堆
  - ◆ 维护一个N(该题目中是300万)大小的**最大堆**
  - ◆ 数组存储, **空间足够**
- 算法思想:
  - ◆ 初始化放入300万个查询串, 按照统计次数排列成堆
  - ◆ 依次删除根节点10次, 那么这个数组中的最后10个查询串便是我们要找的Top10了
- 时间复杂度:
  - ◆ 堆的初始化为300万
  - ◆ 每次删除根节点 $\text{Log}N$
  - ◆ 总体复杂度为 **$N+k*\text{Log}N$**



# 找出Top10

- (5) . 直接找最大值
- 算法思想:
  - ◆ 按照selectMax算法执行
  - ◆ 依次执行K遍
- 时间复杂度:
  - ◆ 最坏时间复杂度是 $N-1+N-2 + \dots + N-10$
  - ◆ 最好情况复杂度为 $N-1$





# 找出Top10

---

■ (6) . ?





# 算法复杂度比较

$\log n$	$n$	$n \log n$	$k$	$\log K$	$k * n$	$n * \log K$	$N + k * \log n$
0	1	0	10	3.3	10	3.3	1
10	1024	10240	10	3.3	10240	3401	1124
20	1048576	20971520	10	3.3	10485760	3483294	1048776
21	3000000	64549593	10	3.3	30000000	9965784	3000215
23	10000000	232534966	10	3.3	100000000	33219280	10000232
26	100000000	2657542476	10	3.3	1000000000	332192809	100000265



## 查找两个文件中共同的url

---

- 给定a、b两个文件，各存放50亿个url，每个url各占64字节，内存限制是4G，让你找出a、b文件共同的url？



- 估计每个文件的大小为 $50\text{亿} \times 64 = 320\text{G}$
- 远远大于内存限制的4G
- 所以不可能将其完全加载到内存中处理
- 采取分而治之的方法，来解决内存限制的问题



## 哈希切分

- 遍历文件a，对每个url求取 $\text{hash}(\text{url})\%1000$ ，然后根据所取得的值将url分别存储到1000个小文件（记为 $a_0, a_1, \dots, a_{999}$ ）中。这样每个小文件的大约为300M。
- 遍历文件b，采取和a相同的方式将url分别存储到1000小文件（记为 $b_0, b_1, \dots, b_{999}$ ）。



- 所有可能相同的url都在对应的小文件（ $a_0$  vs  $b_0$ ,  $a_1$  vs  $b_1$ , ...,  $a_{999}$  vs  $b_{999}$ ）中，
  - 不对应的小文件不可能有相同的url。
- ➡ 只要求出1000对小文件中相同的url即可。
- 求 $a_i$ 和 $b_i$ 小文件中相同的url
    - 把 $a_i$ 文件的url存储到hash\_set中
    - 遍历 $b_i$ 文件的每个url，在刚才构建的hash\_set中搜索，搜索成功，该url就是共同的url，存到文件中。