数据库

175 组合两个表

```
表1: Person

+-------------+---------+
| 列名          | 类型      |
+-------------+---------+
| PersonId    | int     |
| FirstName   | varchar |
| LastName    | varchar |
+-------------+---------+
PersonId 是上表主键

表2: Address

+-------------+---------+
| 列名          | 类型     |
+-------------+---------+
| AddressId   | int     |
| PersonId    | int     |
| City        | varchar |
| State       | varchar |
+-------------+---------+
AddressId 是上表主键

编写一个 SQL 查询，满足条件：无论 person 是否有地址信息，都需要基于上述两表提供 person 的以下信息：
FirstName, LastName, City, State

# Write your MySQL query statement below
select FirstName, LastName, City, State from Person p left join Address a on p.PersonId = a.PersonId
```

176 第二高的薪水

```
编写一个 SQL 查询，获取 Employee 表中第二高的薪水（Salary） 。

+----+--------+
| Id | Salary |
+----+--------+
| 1  | 100    |
| 2  | 200    |
| 3  | 300    |
+----+--------+

例如上述 Employee 表，SQL查询应该返回 200 作为第二高的薪水。如果不存在第二高的薪水，那么查询应返回 null。

+---------------------+
| SecondHighestSalary |
```

```
+---------------------+
| 200                 |
+---------------------+

# mysql
# Write your MySQL query statement below
select
ifnull(select distinct salary from Employee order by salary limit 1,1,null)
as SecondHighestSalary

/* Write your PL/SQL query statement below */
select Salary as SecondHighestSalary
from(
    select Salary, dense_rank() over(order by Salary desc) rn
    from Employee
    ) t
where rn =2;
```

177 第N高的薪水

编写一个 SQL 查询，获取 Employee 表中第 n 高的薪水（Salary）。

```
+----+--------+
| Id | Salary |
+----+--------+
| 1  | 100    |
| 2  | 200    |
| 3  | 300    |
+----+--------+
```

例如上述 Employee 表，n = 2 时，应返回第二高的薪水 200。如果不存在第 n 高的薪水，那么查询应返回 null。

```
+------------------------+
| getNthHighestSalary(2) |
+------------------------+
| 200                    |
+------------------------+

#mysql
CREATE FUNCTION getNthHighestSalary(N INT) RETURNS INT
BEGIN
    RETURN(
        select if(cnt<N,null,mmax) from (
            select min(Salary) as mmax, count(1) as cnt from
            (
            select distinct Salary from Employee order by Salary desc limit n
            ) as a
        ) as b
    );
END

#oracle
CREATE FUNCTION getNthHighestSalary(N IN NUMBER) RETURN NUMBER IS
result NUMBER;
```

```
BEGIN
    /* Write your PL/SQL query statement below */

    RETURN result;
END;
```

178 分数排名

编写一个 SQL 查询来实现分数排名。

如果两个分数相同，则两个分数排名（Rank）相同。请注意，平分后的下一个名次应该是下一个连续的整数值。换句话说，名次之间不应该有“间隔”。

```
+----+-------+
| Id | Score |
+----+-------+
| 1  | 3.50  |
| 2  | 3.65  |
| 3  | 4.00  |
| 4  | 3.85  |
| 5  | 4.00  |
| 6  | 3.65  |
+----+-------+
```

例如，根据上述给定的 Scores 表，你的查询应该返回（按分数从高到低排列）：

```
+-------+------+
| Score | Rank |
+-------+------+
| 4.00  | 1    |
| 4.00  | 1    |
| 3.85  | 2    |
| 3.65  | 3    |
| 3.65  | 3    |
| 3.50  | 4    |
+-------+------+
```

```
select *,
   rank() over (order by 成绩 desc) as ranking,
   dense_rank() over (order by 成绩 desc) as dese_rank,
   row_number() over (order by 成绩 desc) as row_num
from 班级
```

1）rank函数：这个例子中是5位，5位，5位，8位，也就是如果有并列名次的行，会占用下一名次的位置。比如正常排名是1，2，3，4，但是现在前3名是并列的名次，结果是：1，1，1，4。

2）dense_rank函数：这个例子中是5位，5位，5位，6位，也就是如果有并列名次的行，不占用下一名次的位置。比如正常排名是1，2，3，4，但是现在前3名是并列的名次，结果是：1，1，1，2。

3）row_number函数：这个例子中是5位，6位，7位，8位，也就是不考虑并列名次的情况。比如前3名是并列的名次，排名是正常的1，2，3，4。

```
#mysql,oracle
SELECT Score, DENSE_RANK() OVER(ORDER BY Score DESC) AS 'Rank'
FROM Scores
```

180 连续出现的数字

编写一个 SQL 查询，查找所有至少连续出现三次的数字。

```
+----+-----+
| Id | Num |
+----+-----+
| 1  |  1  |
| 2  |  1  |
| 3  |  1  |
| 4  |  2  |
| 5  |  1  |
| 6  |  2  |
| 7  |  2  |
+----+-----+
```

例如，给定上面的 Logs 表， 1 是唯一连续出现至少三次的数字。

```
+-----------------+
| ConsecutiveNums |
+-----------------+
| 1               |
+-----------------+
```

```
# Write your MySQL query statement below
SELECT DISTINCT
    l1.Num AS ConsecutiveNums
FROM
    Logs l1,
    Logs l2,
    Logs l3
WHERE
    l1.Id = l2.Id - 1
    AND l2.Id = l3.Id - 1
    AND l1.Num = l2.Num
    AND l2.Num = l3.Num
;
```

181 超过经理收入的员工

Employee 表包含所有员工，他们的经理也属于员工。每个员工都有一个 Id，此外还有一列对应员工的经理的 Id。

```
+----+-------+--------+-----------+
| Id | Name  | Salary | ManagerId |
+----+-------+--------+-----------+
| 1  | Joe   | 70000  | 3         |
| 2  | Henry | 80000  | 4         |
| 3  | Sam   | 60000  | NULL      |
```

```
| 4  | Max   | 90000  | NULL      |
+----+-------+--------+-----------+
```

给定 Employee 表，编写一个 SQL 查询，该查询可以获取收入超过他们经理的员工的姓名。在上面的表格中，Joe 是唯一一个收入超过他的经理的员工。

```
+----------+
| Employee |
+----------+
| Joe      |
+----------+
```

```sql
SELECT
    a.NAME AS Employee
FROM Employee AS a JOIN Employee AS b
    ON a.ManagerId = b.Id
    AND a.Salary > b.Salary
;
```

182 查找重复的电子邮箱

编写一个 SQL 查询，查找 Person 表中所有重复的电子邮箱。

示例：

```
+----+---------+
| Id | Email   |
+----+---------+
| 1  | a@b.com |
| 2  | c@d.com |
| 3  | a@b.com |
+----+---------+
```

根据以上输入，你的查询应返回以下结果：

```
+---------+
| Email   |
+---------+
| a@b.com |
+---------+
```

```sql
# Write your MySQL query statement below
select Email from person group by email having count(Email) > 1
```

183 从不订购的客户

某网站包含两个表，Customers 表和 Orders 表。编写一个 SQL 查询，找出所有从不订购任何东西的客户。

Customers 表：

```
+----+-------+
| Id | Name  |
```

```
+----+-------+
| 1  | Joe   |
| 2  | Henry |
| 3  | Sam   |
| 4  | Max   |
+----+-------+

Orders 表：

+----+------------+
| Id | CustomerId |
+----+------------+
| 1  | 3          |
| 2  | 1          |
+----+------------+

例如给定上述表格，你的查询应返回：

+-----------+
| Customers |
+-----------+
| Henry     |
| Max       |
+-----------+
```

```sql
# Write your MySQL query statement below
-- select name as Customers from Customers c left join Orders o on c.id =
o.CustomerId
-- where o.CustomerId is null


select name as Customers from Customers where id not in
(

   select CustomerId from Orders
)
```

184 部门工资最高的员工

```
Employee 表包含所有员工信息，每个员工有其对应的 Id, salary 和 department Id。

+----+-------+--------+--------------+
| Id | Name  | Salary | DepartmentId |
+----+-------+--------+--------------+
| 1  | Joe   | 70000  | 1            |
| 2  | Jim   | 90000  | 1            |
| 3  | Henry | 80000  | 2            |
| 4  | Sam   | 60000  | 2            |
| 5  | Max   | 90000  | 1            |
+----+-------+--------+--------------+

Department 表包含公司所有部门的信息。

+----+----------+
| Id | Name     |
```

```
+----+----------+
| 1  | IT       |
| 2  | Sales    |
+----+----------+
```

编写一个 SQL 查询，找出每个部门工资最高的员工。对于上述表，您的 SQL 查询应返回以下行（行的顺序无关紧要）。

```
+------------+----------+--------+
| Department | Employee | Salary |
+------------+----------+--------+
| IT         | Max      | 90000  |
| IT         | Jim      | 90000  |
| Sales      | Henry    | 80000  |
+------------+----------+--------+
```

```sql
SELECT
    Department.name AS 'Department',
    Employee.name AS 'Employee',
    Salary
FROM
    Employee
        JOIN
    Department ON Employee.DepartmentId = Department.Id
WHERE
    (Employee.DepartmentId , Salary) IN
    (   SELECT
            DepartmentId, MAX(Salary)
        FROM
            Employee
        GROUP BY DepartmentId
    )
;
```

185 部门工资前三高的所有员工

Employee 表包含所有员工信息，每个员工有其对应的工号 Id，姓名 Name，工资 Salary 和部门编号 DepartmentId 。

```
+----+-------+--------+--------------+
| Id | Name  | Salary | DepartmentId |
+----+-------+--------+--------------+
| 1  | Joe   | 85000  | 1            |
| 2  | Henry | 80000  | 2            |
| 3  | Sam   | 60000  | 2            |
| 4  | Max   | 90000  | 1            |
| 5  | Janet | 69000  | 1            |
| 6  | Randy | 85000  | 1            |
| 7  | Will  | 70000  | 1            |
+----+-------+--------+--------------+
```

Department 表包含公司所有部门的信息。

```
+----+----------+
| Id | Name     |
```

```
+----+----------+
| 1  | IT       |
| 2  | Sales    |
+----+----------+
```

编写一个 SQL 查询，找出每个部门获得前三高工资的所有员工。例如，根据上述给定的表，查询结果应返回：

```
+------------+----------+--------+
| Department | Employee | Salary |
+------------+----------+--------+
| IT         | Max      | 90000  |
| IT         | Randy    | 85000  |
| IT         | Joe      | 85000  |
| IT         | Will     | 70000  |
| Sales      | Henry    | 80000  |
| Sales      | Sam      | 60000  |
+------------+----------+--------+
```

解释：

IT 部门中，Max 获得了最高的工资，Randy 和 Joe 都拿到了第二高的工资，Will 的工资排第三。销售部门（Sales）只有两名员工，Henry 的工资最高，Sam 的工资排第二。

```
#mysql
SELECT
    d.Name AS 'Department', e1.Name AS 'Employee', e1.Salary
FROM
    Employee e1
        JOIN
    Department d ON e1.DepartmentId = d.Id
WHERE
    3 > (SELECT
            COUNT(DISTINCT e2.Salary)
        FROM
            Employee e2
        WHERE
            e2.Salary > e1.Salary
                AND e1.DepartmentId = e2.DepartmentId
        )
;
#oracle
select de.name as Department  ,ee.employee as Employee,ee.salary as Salary
from (
    select e2.DepartmentId,e.salary,e.employee,count(e2.Salary) as qc
    from (select DepartmentId ,Salary  ,name as employee
        from Employee
        GROUP BY DepartmentId,Salary, name   ) e
    left join (select DepartmentId ,Salary
        from Employee
        GROUP BY DepartmentId,Salary )e2
    on e.DepartmentId = e2.DepartmentId
    where e.salary <= e2.Salary and e.DepartmentId = e2.DepartmentId
    group by e2.DepartmentId,e.salary,e.employee
) ee join Department  de on de.id = ee.DepartmentId
where ee.qc<=3
order by de.name,ee.qc asc
;
```

196 删除重复的电子邮箱

编写一个 SQL 查询，来删除 Person 表中所有重复的电子邮箱，重复的邮箱里只保留 Id 最小 的那个。

```
+----+------------------+
| Id | Email            |
+----+------------------+
| 1  | john@example.com |
| 2  | bob@example.com  |
| 3  | john@example.com |
+----+------------------+
```

Id 是这个表的主键。

例如，在运行你的查询语句之后，上面的 Person 表应返回以下几行:

```
+----+------------------+
| Id | Email            |
+----+------------------+
| 1  | john@example.com |
| 2  | bob@example.com  |
+----+------------------+
```

提示：

执行 SQL 之后，输出是整个 Person 表。
使用 delete 语句。

```
# Write your MySQL query statement below

-- delete p1 from person p1 left join person p2 on p1.email = p2.email where
p1.id > p2.id

delete p1 from person p1,person p2 where p1.email= p2.email and p1.id > p2.id
```

197 上升的温度

给定一个 Weather 表，编写一个 SQL 查询，来查找与之前（昨天的）日期相比温度更高的所有日期的 Id。

```
+---------+------------------+------------------+
| Id(INT) | RecordDate(DATE) | Temperature(INT) |
+---------+------------------+------------------+
|       1 |       2015-01-01 |               10 |
|       2 |       2015-01-02 |               25 |
|       3 |       2015-01-03 |               20 |
|       4 |       2015-01-04 |               30 |
+---------+------------------+------------------+
```

例如，根据上述给定的 Weather 表格，返回如下 Id:

```
+----+
| Id |
+----+
|  2 |
|  4 |
+----+
```

```mysql
# Write your MySQL query statement below
select w1.id from weather w1, weather w2
where datediff(w1.recorddate,w2.recorddate) = 1
and w1.Temperature > w2.Temperature

/* Write your PL/SQL query statement below */
select w1.id from weather w1, weather w2
where w1.recorddate = w2.recorddate + 1
and w1.Temperature > w2.Temperature
```

262 行程和用户

Trips 表中存所有出租车的行程信息。每段行程有唯一键 Id，Client_Id 和 Driver_Id 是 Users 表中 Users_Id 的外键。Status 是枚举类型，枚举成员为 (‘completed’, ‘cancelled_by_driver’, ‘cancelled_by_client’)。

```
+----+-----------+-----------+---------+--------------------+----------+
| Id | Client_Id | Driver_Id | City_Id |        Status      |Request_at|
+----+-----------+-----------+---------+--------------------+----------+
| 1  |     1     |    10     |    1    |     completed      |2013-10-01|
| 2  |     2     |    11     |    1    | cancelled_by_driver|2013-10-01|
| 3  |     3     |    12     |    6    |     completed      |2013-10-01|
| 4  |     4     |    13     |    6    | cancelled_by_client|2013-10-01|
| 5  |     1     |    10     |    1    |     completed      |2013-10-02|
| 6  |     2     |    11     |    6    |     completed      |2013-10-02|
| 7  |     3     |    12     |    6    |     completed      |2013-10-02|
| 8  |     2     |    12     |    12   |     completed      |2013-10-03|
| 9  |     3     |    10     |    12   |     completed      |2013-10-03|
| 10 |     4     |    13     |    12   | cancelled_by_driver|2013-10-03|
+----+-----------+-----------+---------+--------------------+----------+
```

Users 表存所有用户。每个用户有唯一键 Users_Id。Banned 表示这个用户是否被禁止，Role 则是一个表示（‘client’, ‘driver’, ‘partner’）的枚举类型。

```
+----------+--------+--------+
| Users_Id | Banned |  Role  |
+----------+--------+--------+
|    1     |   No   | client |
|    2     |   Yes  | client |
|    3     |   No   | client |
|    4     |   No   | client |
|    10    |   No   | driver |
|    11    |   No   | driver |
|    12    |   No   | driver |
|    13    |   No   | driver |
+----------+--------+--------+
```

写一段 SQL 语句查出 2013年10月1日 至 2013年10月3日 期间非禁止用户的取消率。基于上表，你的 SQL 语句应返回如下结果，取消率（Cancellation Rate）保留两位小数。

取消率的计算方式如下：(被司机或乘客取消的非禁止用户生成的订单数量) / (非禁止用户生成的订单总数)

```
+------------+-------------------+
|     Day    | Cancellation Rate |
+------------+-------------------+
| 2013-10-01 |       0.33        |
| 2013-10-02 |       0.00        |
| 2013-10-03 |       0.50        |
+------------+-------------------+
```

```
# Write your MySQL query statement below

SELECT
t.request_at as 'day',
ROUND(
sum(if(t.STATUS = 'completed',0,1)) / count(t.STATUS),2
) as 'Cancellation Rate'
from trips t join users u1 on t.client_id = u1.users_id
join users u2 on t.driver_id = u2.users_id
where u1.banned = 'no' and u2.banned = 'no'
and t.request_at BETWEEN '2013-10-01' AND '2013-10-03'
GROUP by t.request_at

-- SELECT T.request_at AS `Day`,
--  ROUND(
--          SUM(
--              IF(T.STATUS = 'completed',0,1)
--          )
--          /
--          COUNT(T.STATUS),
--          2
--  ) AS `Cancellation Rate`
-- FROM Trips AS T
-- JOIN Users AS U1 ON (T.client_id = U1.users_id AND U1.banned ='No')
-- JOIN Users AS U2 ON (T.driver_id = U2.users_id AND U2.banned ='No')
-- WHERE T.request_at BETWEEN '2013-10-01' AND '2013-10-03'
-- GROUP BY T.request_at

/* Write your PL/SQL query statement below */

SELECT
t.request_at as 'day',
ROUND(
sum(decode(t.STATUS,'completed',0,1)) / count(t.STATUS),2
) as 'Cancellation Rate'
from trips t join users u1 on t.client_id = u1.users_id
join users u2 on t.driver_id = u2.users_id
where u1.banned = 'no' and u2.banned = 'no'
and t.request_at BETWEEN '2013-10-01' AND '2013-10-03'
GROUP by t.request_at
```

多线程

1114 按序打印

```
  public class Foo {
  public void first() { print("first"); }
  public void second() { print("second"); }
  public void third() { print("third"); }
}
```

三个不同的线程将会共用一个 Foo 实例。

    线程 A 将会调用 first() 方法
    线程 B 将会调用 second() 方法
    线程 C 将会调用 third() 方法

请设计修改程序，以确保 second() 方法在 first() 方法之后被执行，third() 方法在 second() 方法之后被执行。

示例 1:

输入: [1,2,3]
输出: "firstsecondthird"
解释:
有三个线程会被异步启动。
输入 [1,2,3] 表示线程 A 将会调用 first() 方法，线程 B 将会调用 second() 方法，线程 C 将会调用 third() 方法。
正确的输出是 "firstsecondthird"。

示例 2:

输入: [1,3,2]
输出: "firstsecondthird"
解释:
输入 [1,3,2] 表示线程 A 将会调用 first() 方法，线程 B 将会调用 third() 方法，线程 C 将会调用 second() 方法。
正确的输出是 "firstsecondthird"。

```
  private Semaphore two = new Semaphore(0);
    private Semaphore three = new Semaphore(0);

    public Foo() {

    }

    public void first(Runnable printFirst) throws InterruptedException {

        // printFirst.run() outputs "first". Do not change or remove this line.
        printFirst.run();
        two.release();
    }

    public void second(Runnable printSecond) throws InterruptedException {
        two.acquire();
        // printSecond.run() outputs "second". Do not change or remove this
line.
        printSecond.run();
```

```
        three.release();
    }

    public void third(Runnable printThird) throws InterruptedException {
        three.acquire();
        // printThird.run() outputs "third". Do not change or remove this line.
        printThird.run();
    }

    //信号量
    private Semaphore two = new Semaphore(0);
    private Semaphore three = new Semaphore(0);

    public Foo() {

    }

    public void first(Runnable printFirst) throws InterruptedException {

        // printFirst.run() outputs "first". Do not change or remove this line.
        printFirst.run();
        two.release();
    }

    public void second(Runnable printSecond) throws InterruptedException {
        two.acquire();
        // printSecond.run() outputs "second". Do not change or remove this
line.
        printSecond.run();
        three.release();
    }

    public void third(Runnable printThird) throws InterruptedException {
        three.acquire();
        // printThird.run() outputs "third". Do not change or remove this line.
        printThird.run();
    }
```

1115 交替打印FooBar

我们提供一个类：

```
class FooBar {
  public void foo() {
    for (int i = 0; i < n; i++) {
      print("foo");
    }
  }

  public void bar() {
    for (int i = 0; i < n; i++) {
      print("bar");
    }
  }
}
```

两个不同的线程将会共用一个 FooBar 实例。其中一个线程将会调用 foo() 方法，另一个线程将会调用 bar() 方法。

请设计修改程序，以确保 "foobar" 被输出 n 次。

示例 1:

输入: n = 1
输出: "foobar"
解释: 这里有两个线程被异步启动。其中一个调用 foo() 方法，另一个调用 bar() 方法，"foobar" 将被输出一次。

示例 2:

输入: n = 2
输出: "foobarfoobar"
解释: "foobar" 将被输出两次。

```
class FooBar {
    private int n;

    public FooBar(int n) {
        this.n = n;
    }

    Semaphore foo = new Semaphore(1);
    Semaphore bar = new Semaphore(0);

    public void foo(Runnable printFoo) throws InterruptedException {
        for (int i = 0; i < n; i++) {
            foo.acquire();
            printFoo.run();
            bar.release();
        }
    }

    public void bar(Runnable printBar) throws InterruptedException {
        for (int i = 0; i < n; i++) {
            bar.acquire();
            printBar.run();
            foo.release();
        }
    }
}
```

1116 打印零与奇偶数

假设有这么一个类：

```
class ZeroEvenOdd {
  public ZeroEvenOdd(int n) { ... }      // 构造函数
  public void zero(printNumber) { ... }  // 仅打印出 0
  public void even(printNumber) { ... }  // 仅打印出 偶数
  public void odd(printNumber) { ... }   // 仅打印出 奇数
}
```

相同的一个 ZeroEvenOdd 类实例将会传递给三个不同的线程：

线程 A 将调用 zero()，它只输出 0 。
线程 B 将调用 even()，它只输出偶数。
线程 C 将调用 odd()，它只输出奇数。

每个线程都有一个 printNumber 方法来输出一个整数。请修改给出的代码以输出整数序列 010203040506... ，其中序列的长度必须为 2n。

示例 1：

输入：n = 2
输出："0102"
说明：三条线程异步执行，其中一个调用 zero()，另一个线程调用 even()，最后一个线程调用odd()。正确的输出为 "0102"。

示例 2：

输入：n = 5
输出："0102030405"

```
class ZeroEvenOdd {
    private int n;

    public ZeroEvenOdd(int n) {
        this.n = n;
    }

    // printNumber.accept(x) outputs "x", where x is an integer.
    Semaphore z = new Semaphore(1);
    Semaphore e = new Semaphore(0);
    Semaphore o = new Semaphore(0);

    public void zero(IntConsumer printNumber) throws InterruptedException {
        for(int i=0; i<n;i++) {
            z.acquire();
            printNumber.accept(0);
            if((i&1)==0) {
                o.release();
            }else {
                e.release();
            }
        }
    }

    public void even(IntConsumer printNumber) throws InterruptedException {
        for(int i=2; i<=n; i+=2) {
            e.acquire();
            printNumber.accept(i);
            z.release();
        }
    }
```

1117 H2O 生成

现在有两种线程，氧 oxygen 和氢 hydrogen，你的目标是组织这两种线程来产生水分子。

存在一个屏障（barrier）使得每个线程必须等候直到一个完整水分子能够被产生出来。

氢和氧线程会被分别给予 releaseHydrogen 和 releaseOxygen 方法来允许它们突破屏障。

这些线程应该三三成组突破屏障并能立即组合产生一个水分子。

你必须保证产生一个水分子所需线程的结合必须发生在下一个水分子产生之前。

换句话说:

> 如果一个氧线程到达屏障时没有氢线程到达，它必须等候直到两个氢线程到达。
> 如果一个氢线程到达屏障时没有其它线程到达，它必须等候直到一个氧线程和另一个氢线程到达。

书写满足这些限制条件的氢、氧线程同步代码。

示例 1:

输入: "HOH"
输出: "HHO"
解释: "HOH" 和 "OHH" 依然都是有效解。

示例 2:

输入: "OOHHHH"
输出: "HHOHHO"
解释: "HOHHHO", "OHHHHO", "HHOHOH", "HOHHOH", "OHHHOH", "HHOOHH", "HOHOHH" 和 "OHHOHH" 依然都是有效解。

```
class H2O {

    public H2O() {

    }

    private Semaphore h = new Semaphore(2);
    private Semaphore o = new Semaphore(0);

    public void hydrogen(Runnable releaseHydrogen) throws InterruptedException {
        h.acquire();

        // releaseHydrogen.run() outputs "H". Do not change or remove this line.
        releaseHydrogen.run();
        o.release();
    }

    public void oxygen(Runnable releaseOxygen) throws InterruptedException {
        o.acquire(2);
        // releaseOxygen.run() outputs "O". Do not change or remove this line.
        releaseOxygen.run();
        h.release(2);
    }
}
```

1188 设计有限阻塞队列

实现一个拥有如下方法的线程安全有限阻塞队列：

`BoundedBlockingQueue(int capacity)` 构造方法初始化队列，其中`capacity`代表队列长度上限。
`void enqueue(int element)` 在队首增加一个`element`. 如果队列满，调用线程被阻塞直到队列非满。
`int dequeue()` 返回队尾元素并从队列中将其删除. 如果队列为空，调用线程被阻塞直到队列非空。
`int size()` 返回当前队列元素个数。

你的实现将会被多线程同时访问进行测试。每一个线程要么是一个只调用`enqueue`方法的生产者线程，要么是一个只调用`dequeue`方法的消费者线程。`size`方法将会在每一个测试用例之后进行调用。

请不要使用内置的有限阻塞队列实现，否则面试将不会通过。

示例 1:

输入:
```
1
1
["BoundedBlockingQueue","enqueue","dequeue","dequeue","enqueue","enqueue","enqueue","enqueue","dequeue"]
[[2],[1],[],[],[0],[2],[3],[4],[]]
```

输出:
```
[1,0,2,2]
```

解释:
```
生产者线程数目 = 1
消费者线程数目 = 1

BoundedBlockingQueue queue = new BoundedBlockingQueue(2);   // 使用capacity = 2初始化队列。

queue.enqueue(1);   // 生产者线程将1插入队列。
queue.dequeue();    // 消费者线程调用dequeue并返回1。
queue.dequeue();    // 由于队列为空，消费者线程被阻塞。
queue.enqueue(0);   // 生产者线程将0插入队列。消费者线程被解除阻塞同时将0弹出队列并返回。
queue.enqueue(2);   // 生产者线程将2插入队列。
queue.enqueue(3);   // 生产者线程将3插入队列。
queue.enqueue(4);   // 生产者线程由于队列长度已达到上限2而被阻塞。
queue.dequeue();    // 消费者线程将2从队列弹出并返回。生产者线程解除阻塞同时将4插入队列。
queue.size();       // 队列中还有2个元素。size()方法在每组测试用例最后调用。
```

示例 2:

输入:
```
3
4
```

```
["BoundedBlockingQueue","enqueue","enqueue","enqueue","dequeue","dequeue","deque
ue","enqueue"]
[[3],[1],[0],[2],[],[],[],[3]]

输出:
[1,0,2,1]

解释:
生产者线程数目 = 3
消费者线程数目 = 4

BoundedBlockingQueue queue = new BoundedBlockingQueue(3);   // 使用capacity = 3初
始化队列。

queue.enqueue(1);   // 生产者线程P1将1插入队列。
queue.enqueue(0);   // 生产者线程P2将0插入队列。
queue.enqueue(2);   // 生产者线程P3将2插入队列。
queue.dequeue();    // 消费者线程C1调用dequeue。
queue.dequeue();    // 消费者线程C2调用dequeue。
queue.dequeue();    // 消费者线程C3调用dequeue。
queue.enqueue(3);   // 其中一个生产者线程将3插入队列。
queue.size();       // 队列中还有1个元素。
```

由于生产者/消费者线程的数目可能大于1，我们并不知道线程如何被操作系统调度，即使输入看上去隐含了顺序。因此任意一种输出[1,0,2]或[1,2,0]或[0,1,2]或[0,2,1]或[2,0,1]或[2,1,0]都可被接受。

```
import java.util.LinkedList;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

class BoundedBlockingQueue {


    //原子类保证原子性，也可以使用volatile
    //普通的int被读取，会被读入内存的缓存中，完成加减乘除后再放回内存中，而每一个线程都有自己的
寄存器，这样子会导致可能读取不到最新的数据
    //volatile则可以直接在主内存读写，当一个线程更新了值，其他线程能够及时获知。
    AtomicInteger size = new AtomicInteger(0);
    private volatile int capacity;
    //自己实现阻塞队列，需要一个容器，内部实现了一个node，如果改造为不只是int的，使用T泛型
    private LinkedList<Integer> container;

    //可重入锁
    private static ReentrantLock lock = new ReentrantLock();
    Condition procuder = lock.newCondition();//用来通知生产（入队）线程等待await还是可
以执行signal
    Condition consumer = lock.newCondition();//用来通知消费（出队）线程等待await还是可
以执行signal

    public BoundedBlockingQueue(int capacity) {
        this.capacity = capacity;
        container = new LinkedList<>();
    }

    /**
     * 入队
     *
```

```
     * @param element
     * @throws InterruptedException
     */
    public void enqueue(int element) throws InterruptedException {
        //每一个线程都会获得锁，但是如果条件不满足则会阻塞
        lock.lock();
        try {
            //阻塞的话必须用循环，让这个线程再次获得cpu片段的时候能够够执行
            while (size.get() >= capacity) {
                //入队线程阻塞，把锁释放？
                procuder.await();
            }
            container.addFirst(element);
            size.incrementAndGet();

            //通知出队线程
            consumer.signal();
        } finally {
            lock.unlock();
        }
    }

    public int dequeue() throws InterruptedException {
        lock.lock();
        try {
            while (size.get() == 0) {
                consumer.await();
            }
            int lastValue = container.getLast();
            container.removeLast();
            size.decrementAndGet();

            //通知入队线程
            procuder.signal();
            return lastValue;
        } finally {
            lock.unlock();
        }
    }

    public int size() {
        lock.lock();
        try {
            return size.get();
        } finally {
            lock.unlock();
        }
    }
}
```

1195 交替打印字符串

编写一个可以从 1 到 n 输出代表这个数字的字符串的程序，但是：

如果这个数字可以被 3 整除，输出 "fizz"。
如果这个数字可以被 5 整除，输出 "buzz"。
如果这个数字可以同时被 3 和 5 整除，输出 "fizzbuzz"。

例如，当 n = 15，输出： 1, 2, fizz, 4, buzz, fizz, 7, 8, fizz, buzz, 11, fizz, 13, 14, fizzbuzz。

假设有这么一个类：

```
class FizzBuzz {
  public FizzBuzz(int n) { ... }               // constructor
  public void fizz(printFizz) { ... }          // only output "fizz"
  public void buzz(printBuzz) { ... }          // only output "buzz"
  public void fizzbuzz(printFizzBuzz) { ... }  // only output "fizzbuzz"
  public void number(printNumber) { ... }      // only output the numbers
}
```

请你实现一个有四个线程的多线程版 FizzBuzz， 同一个 FizzBuzz 实例会被如下四个线程使用：

线程A将调用 fizz() 来判断是否能被 3 整除，如果可以，则输出 fizz。
线程B将调用 buzz() 来判断是否能被 5 整除，如果可以，则输出 buzz。
线程C将调用 fizzbuzz() 来判断是否同时能被 3 和 5 整除，如果可以，则输出 fizzbuzz。
线程D将调用 number() 来实现输出既不能被 3 整除也不能被 5 整除的数字。


```
class FizzBuzz {
    private int n;

    public FizzBuzz(int n) {
        this.n = n;
    }

    private Semaphore fSema = new Semaphore(0);
    private Semaphore bSema = new Semaphore(0);
    private Semaphore fbSema = new Semaphore(0);
    private Semaphore nSema = new Semaphore(1);

    // private ReentrantLock lock = new ReentrantLock();
    // private Condition fCond = lock.newCondition();
    // private Condition bCond = lock.newCondition();
    // private Condition fbCond = lock.newCondition();
    // private Condition nCond = lock.newCondition();
    // private volatile Boolean state = false;

    /**
    * use semaphore
    * @param printFizz
    * @throws InterruptedException
    */
    // printFizz.run() outputs "fizz".
    public void fizz(Runnable printFizz) throws InterruptedException {
        for (int i = 3; i <= n; i = i + 3) {
            if (i % 5 != 0) {
```

```
                fSema.acquire();
                printFizz.run();
                nSema.release();
            }
        }
    }

    // printBuzz.run() outputs "buzz".
    public void buzz(Runnable printBuzz) throws InterruptedException {
        for (int i = 5; i <= n; i = i + 5) {
            if (i % 3 != 0) {
                bSema.acquire();
                printBuzz.run();
                nSema.release();
            }
        }
    }

    // printFizzBuzz.run() outputs "fizzbuzz".
    public void fizzbuzz(Runnable printFizzBuzz) throws InterruptedException {
        for (int i = 15; i <= n; i = i + 15) {
            fbSema.acquire();
            printFizzBuzz.run();
            nSema.release();
        }
    }

    // printNumber.accept(x) outputs "x", where x is an integer.
    public void number(IntConsumer printNumber) throws InterruptedException {
        for (int i = 1; i <= n; i++) {
            nSema.acquire();
            if (i % 3 == 0 && i % 5 == 0) {
                fbSema.release();
            }else if (i % 3 == 0) {
                fSema.release();
            }else if (i % 5 == 0) {
                bSema.release();
            }else {
                printNumber.accept(i);
                nSema.release();
            }
        }
    }


    /**
    * use lock
    * @param printFizz
    * @throws InterruptedException
    */
    // printFizz.run() outputs "fizz".
    // public void fizz1(Runnable printFizz) throws InterruptedException {
    //     for (int i = 3; i <= n; i = i + 3) {
    //         lock.lock();
    //         try {
    //             if (i % 5 != 0) {
    //                 while (!state) {
    //                     fCond.await();
```

```
    //                 }
    //                 printFizz.run();
    //                 state = false;
    //                 nCond.signal();
    //             }
    //         }finally {
    //             lock.unlock();
    //         }
    //     }
    // }

    // // printBuzz.run() outputs "buzz".
    // public void buzz1(Runnable printBuzz) throws InterruptedException {
    //     for (int i = 5; i <= n; i = i + 5) {
    //         lock.lock();
    //         try {
    //             if (i % 3 != 0) {
    //                 while (!state) {
    //                     bCond.await();
    //                 }
    //                 printBuzz.run();
    //                 state = false;
    //                 nCond.signal();
    //             }
    //         }finally {
    //             lock.unlock();
    //         }
    //     }
    // }

    // // printFizzBuzz.run() outputs "fizzbuzz".
    // public void fizzbuzz1(Runnable printFizzBuzz) throws InterruptedException
{
    //     for (int i = 15; i <= n; i = i + 15) {
    //         lock.lock();
    //         try {
    //             while (!state) {
    //                 fbCond.await();
    //             }
    //             printFizzBuzz.run();
    //             state = false;
    //             nCond.signal();
    //         }finally {
    //             lock.unlock();
    //         }

    //     }
    // }

    // // printNumber.accept(x) outputs "x", where x is an integer.
    // public void number1(IntConsumer printNumber) throws InterruptedException
{
    //     for (int i = 1; i <= n; i++) {
    //         lock.lock();
    //         try {
    //             while (state) {
    //                 nCond.await();
    //             }
```

```
    //             if (i % 3 == 0 && i % 5 == 0) {
    //                 fbCond.signal();
    //                 state = true;
    //             }else if (i % 3 == 0) {
    //                 fCond.signal();
    //                 state = true;
    //             }else if (i % 5 == 0) {
    //                 bCond.signal();
    //                 state = true;
    //             }else {
    //                 printNumber.accept(i);
    //                 nCond.signal();
    //             }
    //         }finally {
    //             lock.unlock();
    //         }
    //     }
    // }

    // public static void main(String[] args) {
    //     PrintFizzBuzz pfb = new PrintFizzBuzz(15);
    //     Thread t1 = new Thread(() -> {
    //         try {
    //             pfb.fizz1(() -> System.out.print("fizz"));
    //         } catch (InterruptedException e) {
    //             e.printStackTrace();
    //         }
    //     });

    //     Thread t2 = new Thread(() -> {
    //         try {
    //             pfb.buzz1(() -> System.out.print("buzz"));
    //         } catch (InterruptedException e) {
    //             e.printStackTrace();
    //         }
    //     });

    //     Thread t3 = new Thread(() -> {
    //         try {
    //             pfb.fizzbuzz1(() -> System.out.print("fizzbuzz"));
    //         } catch (InterruptedException e) {
    //             e.printStackTrace();
    //         }
    //     });

    //     Thread t4 = new Thread(() -> {
    //         try {
    //             pfb.number1(value -> System.out.print(value));
    //         } catch (InterruptedException e) {
    //             e.printStackTrace();
    //         }
    //     });

    //     t1.start();
    //     t2.start();
    //     t3.start();
    //     t4.start();
```

```
    // }


}
```

1226 哲学家进餐

5 个沉默寡言的哲学家围坐在圆桌前，每人面前一盘意面。叉子放在哲学家之间的桌面上。（5 个哲学家，5 根叉子）

所有的哲学家都只会在思考和进餐两种行为间交替。哲学家只有同时拿到左边和右边的叉子才能吃到面，而同一根叉子在同一时间只能被一个哲学家使用。每个哲学家吃完面后都需要把叉子放回桌面以供其他哲学家吃面。只要条件允许，哲学家可以拿起左边或者右边的叉子，但在没有同时拿到左右叉子时不能进食。

假设面的数量没有限制，哲学家也能随便吃，不需要考虑吃不吃得下。

设计一个进餐规则（并行算法）使得每个哲学家都不会挨饿；也就是说，在没有人知道别人什么时候想吃东西或思考的情况下，每个哲学家都可以在吃饭和思考之间一直交替下去。

哲学家从 0 到 4 按 顺时针 编号。请实现函数 `void wantsToEat(philosopher, pickLeftFork, pickRightFork, eat, putLeftFork, putRightFork)`：

- `philosopher` 哲学家的编号。
- `pickLeftFork` 和 `pickRightFork` 表示拿起左边或右边的叉子。
- `eat` 表示吃面。
- `putLeftFork` 和 `putRightFork` 表示放下左边或右边的叉子。
- 由于哲学家不是在吃面就是在想着啥时候吃面，所以思考这个方法没有对应的回调。

给你 5 个线程，每个都代表一个哲学家，请你使用类的同一个对象来模拟这个过程。在最后一次调用结束之前，可能会为同一个哲学家多次调用该函数。

示例：

输入：n = 1
输出：[[4,2,1],[4,1,1],[0,1,1],[2,2,1],[2,1,1],[2,0,3],[2,1,2],[2,2,2],[4,0,3],[4,1,2],[0,2,1],[4,2,2],[3,2,1],[3,1,1],[0,0,3],[0,1,2],[0,2,2],[1,2,1],[1,1,1],[3,0,3],[3,1,2],[3,2,2],[1,0,3],[1,1,2],[1,2,2]]
解释：
n 表示每个哲学家需要进餐的次数。
输出数组描述了叉子的控制和进餐的调用，它的格式如下：
output[i] = [a, b, c] (3个整数)
- a 哲学家编号。
- b 指定叉子：{1 : 左边, 2 : 右边}.
- c 指定行为：{1 : 拿起, 2 : 放下, 3 : 吃面}。
如 [4,2,1] 表示 4 号哲学家拿起了右边的叉子。

```
class DiningPhilosophers {
    int num = 5;
    //五个叉子的信号量
    private Semaphore[] semaphores = new Semaphore[5];

    public DiningPhilosophers() {
```

```
        for (int i = 0; i < num; i++) {
            //每只叉子只有1个
            semaphores[i] = new Semaphore(1);
        }

    }

    // call the run() method of any runnable to execute its code
    public void wantsToEat(int philosopher,
                           Runnable pickLeftFork,
                           Runnable pickRightFork,
                           Runnable eat,
                           Runnable putLeftFork,
                           Runnable putRightFork) throws InterruptedException {
        //左边叉子的位置
        int left = philosopher;
        //右边叉子的位置
        int right = (philosopher + 1) % num;
        while (true) {
            if (semaphores[left].tryAcquire()) {
                //先尝试获取左边叉子，如果成功再尝试获取右边叉子
                if (semaphores[right].tryAcquire()) {
                    //两个叉子都得到了，进餐
                    pickLeftFork.run();
                    pickRightFork.run();
                    eat.run();
                    putLeftFork.run();
                    //释放左边叉子
                    semaphores[left].release();
                    putRightFork.run();
                    //释放右边边叉子
                    semaphores[right].release();

                    //吃完了，就跳出循环
                    break;
                } else {
                    //如果拿到了左边的叉子，但没拿到右边的叉子：  就释放左边叉子
                    semaphores[left].release();
                    //让出cpu等一会
                    Thread.yield();
                }
            } else {
                //连左边叉子都没拿到，就让出cpu等会吧
                Thread.yield();
            }
        }

    }

}
```

1242 多线程网页爬虫

给你一个初始地址 `startUrl` 和一个 `HTML` 解析器接口 `HtmlParser`，请你实现一个 多线程的网页爬虫，用于获取与 `startUrl` 有 相同主机名 的所有链接。

以 任意 顺序返回爬虫获取的路径。

爬虫应该遵循：

- 从 startUrl 开始
- 调用 HtmlParser.getUrls(url) 从指定网页路径获得的所有路径。
- 不要抓取相同的链接两次。
- 仅浏览与 startUrl 相同主机名 的链接。

如上图所示，主机名是 example.org 。简单起见，你可以假设所有链接都采用 http 协议，并且没有指定端口号。举个例子，链接 http://leetcode.com/problems 和链接 http://leetcode.com/contest 属于同一个 主机名， 而 http://example.org/test 与 http://example.com/abc 并不属于同一个 主机名。

HtmlParser 的接口定义如下：

```
interface HtmlParser {
  // Return a list of all urls from a webpage of given url.
  // This is a blocking call, that means it will do HTTP request and return when this request is finished.
  public List<String> getUrls(String url);
}
```

注意一点，getUrls(String url) 模拟执行一个HTTP的请求。 你可以将它当做一个阻塞式的方法，直到请求结束。 getUrls(String url) 保证会在 15ms 内返回所有的路径。 单线程的方案会超过时间限制，你能用多线程方案做的更好吗？

对于问题所需的功能，下面提供了两个例子。为了方便自定义测试，你可以声明三个变量 urls，edges 和 startUrl。但要注意你只能在代码中访问 startUrl，并不能直接访问 urls 和 edges。

拓展问题：

- 假设我们要要抓取 10000 个节点和 10 亿个路径。并且在每个节点部署相同的的软件。软件可以发现所有的节点。我们必须尽可能减少机器之间的通讯，并确保每个节点负载均衡。你将如何设计这个网页爬虫？
- 如果有一个节点发生故障不工作该怎么办？
- 如何确认爬虫任务已经完成？

示例 1：

输入：
```
urls = [
  "http://news.yahoo.com",
  "http://news.yahoo.com/news",
  "http://news.yahoo.com/news/topics/",
  "http://news.google.com",
  "http://news.yahoo.com/us"
]
edges = [[2,0],[2,1],[3,2],[3,1],[0,4]]
startUrl = "http://news.yahoo.com/news/topics/"
输出：[
  "http://news.yahoo.com",
  "http://news.yahoo.com/news",
  "http://news.yahoo.com/news/topics/",
```

```
  "http://news.yahoo.com/us"
]
```

示例 2：

输入：
```
urls = [
  "http://news.yahoo.com",
  "http://news.yahoo.com/news",
  "http://news.yahoo.com/news/topics/",
  "http://news.google.com"
]
edges = [[0,2],[2,1],[3,2],[3,1],[3,0]]
startUrl = "http://news.google.com"
```
输出：["http://news.google.com"]
解释：startUrl 链接与其他页面不共享一个主机名。

提示：

1 <= urls.length <= 1000
1 <= urls[i].length <= 300
startUrl 是 urls 中的一个。
主机名的长度必须为 1 到 63 个字符（包括点 . 在内），只能包含从 “a” 到 “z” 的 ASCII 字母和 “0” 到 “9” 的数字，以及中划线 “-”。
主机名开头和结尾不能是中划线 “-”。
参考资料：
https://en.wikipedia.org/wiki/Hostname#Restrictions_on_valid_hostnames
你可以假设路径都是不重复的。

这道题目是test case 有问题还是写的代码有问题。 为啥不给一下开启的线程数量。
MAX_ALIVE_THREAD_NUM 这个量定小了就会超时，多了就会超出内存限制。

我给的做法是用 CountDownLatch。 起某一数量的线程，把耗时的操作htmlParser.getUrls() 放到一个独立的线程中去进行操作。

这里需要维护一个 线程安全的 queue （也可以去限制queue的大小，e.g boundedblockingqueue）来去保存需要 CrawlWorker 去 crawl 的request url .

因为这里queue和set 都会有多个线程同时读写。所以 要用线程安全的 queue 和set。
ConcurrentLinkedQueue 已经保证了多个线程同时读/写访问的安全性了。

每个阶段开的线程数量取决于queue的size和 MAX_ALIVE_THREAD_NUM 中的最小值。

这道题目的本质就是找到独立互不影响的操作， 开启一个线程去执行。对于这道题目就是对每一个url 爬虫都是一个独立request。 然后从爬出的url选出同一个host以及没出现在结果集的作为新的request放到queue中。

还有一个比较好的练习可以写一下， Merge k sorted list Multithreaded ，关键也是找到线程独立的操作。

```
/**
 * // This is the HtmlParser's API interface.
 * // You should not implement it, or speculate about its implementation
 * interface HtmlParser {
 *     public List<String> getUrls(String url) {}
 * }
```

```
 */
class Solution {


    class CrawlWorker implements Runnable {

        private String startUrl;

        private CountDownLatch countDownLatch;

        private HtmlParser htmlParser;

        CrawlWorker(String startUrl, CountDownLatch countDownLatch,HtmlParser
htmlParser){
            this.startUrl = startUrl;
            this.countDownLatch = countDownLatch;
            this.htmlParser = htmlParser;
        }

        @Override
        public void run() {
            parse();
        }

        private void parse(){
            urlSet.add(startUrl);
            List<String> urlList = htmlParser.getUrls(startUrl);
            for(String url : urlList){
                if(urlSet.contains(url) || !getHost(url).equals(hostName))
continue;
                queue.offer(url);
            }

            this.countDownLatch.countDown();
        }
    }

    private final Set<String> urlSet = ConcurrentHashMap.newKeySet();
    private final Queue<String> queue = new ConcurrentLinkedQueue<>();

    private String hostName;
    private static final Integer MAX_ALIVE_THREAD_NUM = 128;



    public List<String> crawl(String startUrl, HtmlParser htmlParser) {

        hostName = getHost(startUrl);

        queue.offer(startUrl);
        while(!queue.isEmpty()){

            int curThreadNum = Math.min(MAX_ALIVE_THREAD_NUM, queue.size());

            CountDownLatch countDownLatch = new CountDownLatch(curThreadNum);

            for(int idx = 0; idx < curThreadNum ;idx++){
                String curUrl = queue.poll();
```

```
                CrawlWorker crawlWorker  = new
CrawlWorker(curUrl,countDownLatch, htmlParser);
                Thread thread = new Thread(crawlWorker);
                thread.start();
            }

            try {
                countDownLatch.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        return new ArrayList<>(urlSet);
    }


    private static String getHost(String url){
        String host = url.substring(7); // all urls use http protocol
        int idx = host.indexOf('/');
        if(idx == -1) return host;
        return host.substring(0,idx);
    }
}
```

解题思路简单描述：

使用一个ConcurrentHashMap来存储已知的所有URL，主线程向其添加起始URL，多个工作线程可以并发添加已抓取的、符合规则要求的URL。

使用一个LinkedList来保存结果URL，使用对应的ReentrantLock来同步多线程访问，结果只增不减。

使用一个LinkedList来保存待抓取的URL，使用对应的ReentrantLock来同步多线程访问。主线程添加起始URL，工作线程从该List中取出要开始抓取的URL，抓取后将符合规范的URL添加至该链表及结果URL链表。

主循环的退出条件：（1）待抓取URL链表中无元素；（2）已无工作线程（所有的工作线程已完成工作）。

需要说明的是，如果待抓取的URL集合很大，那么创建的工作线程会偏多，在生产实践中可以使用线程池来限制并发度。

```
/**
 * // This is the HtmlParser's API interface.
 * // You should not implement it, or speculate about its implementation
 * interface HtmlParser {
 *     public List<String> getUrls(String url) {}
 * }
 */
class Solution {
    // 已知URL集合，存储当前可见的所有URL。
    private ConcurrentHashMap<String, Boolean> totalUrls = new
ConcurrentHashMap<>();

    // 结果URL链表及对应锁。
    private ReentrantLock resultLock = new ReentrantLock();
    private LinkedList<String> resultUrls = new LinkedList<>();

    // 待抓取URL链表及对应锁。
```

```
    private ReentrantLock crawlLock = new ReentrantLock();
    private LinkedList<String> urlsToCrawl = new LinkedList<>();

    // •当前正在执行的工作线程个数。
    private AtomicInteger choreCount = new AtomicInteger(0);

    public List<String> crawl(String startUrl, HtmlParser htmlParser) {
        String hostName = extractHostName(startUrl);

        this.totalUrls.put(startUrl, true);

        addUrlToResult(startUrl);
        addUrlToCrawl(startUrl);

        while (true) {
            String urlToCrawl = fetchUrlToCrawl();
            if (urlToCrawl != null) {
                incrChore();
                Chore chore = new Chore(this, hostName, htmlParser, urlToCrawl);
                (new Thread(chore)).start();
            } else {
                if (this.choreCount.get() == 0) {
                    break;
                }
                LockSupport.parkNanos(1L);
            }
        }

        return fetchResultUrls();
    }

    private String extractHostName(String url) {
        // HTTP protocol only.
        String processedUrl = url.substring(7);

        int index = processedUrl.indexOf("/");
        if (index == -1) {
            return processedUrl;
        } else {
            return processedUrl.substring(0, index);
        }
    }

    private class Chore implements Runnable {
        private Solution solution;
        private String hostName;
        private HtmlParser htmlParser;
        private String urlToCrawl;

        public Chore(Solution solution, String hostName, HtmlParser htmlParser,
String urlToCrawl) {
            this.solution = solution;
            this.hostName = hostName;
            this.htmlParser = htmlParser;
            this.urlToCrawl = urlToCrawl;
        }

        @Override
```

```
        public void run() {
            try {
                filterUrls(this.htmlParser.getUrls(urlToCrawl));
            } finally {
                this.solution.decrChore();
            }
        }

        private void filterUrls(List<String> crawledUrls) {
            if (crawledUrls == null || crawledUrls.isEmpty()) {
                return;
            }

            for (String url : crawledUrls) {
                // 如果该URL在已知的URL集合中已存在，那么不需要再重复抓取。
                if (this.solution.totalUrls.containsKey(url)) {
                    continue;
                }

                this.solution.totalUrls.put(url, true);

                String crawlHostName = this.solution.extractHostName(url);
                if (!crawlHostName.equals(this.hostName)) {
                    // 如果抓取的URL对应的HostName同Start URL对应的HostName不同，那么
直接丢弃该URL。
                    continue;
                }

                // 将该URL添加至结果链表。
                this.solution.addUrlToResult(url);
                // 将该URL添加至待抓取链表，以便进行下一跳抓取。
                this.solution.addUrlToCrawl(url);
            }
        }
    }

    private void addUrlToResult(String url) {
        this.resultLock.lock();
        try {
            this.resultUrls.add(url);
        } finally {
            this.resultLock.unlock();
        }
    }

    private List<String> fetchResultUrls() {
        this.resultLock.lock();
        try {
            return this.resultUrls;
        } finally {
            this.resultLock.unlock();
        }
    }

    private void addUrlToCrawl(String url) {
        this.crawlLock.lock();
        try {
            this.urlsToCrawl.add(url);
```

```
        } finally {
            this.crawlLock.unlock();
        }
    }

    private String fetchUrlToCrawl() {
        this.crawlLock.lock();
        try {
            return this.urlsToCrawl.poll();
        } finally {
            this.crawlLock.unlock();
        }
    }

    private void incrChore() {
        this.choreCount.incrementAndGet();
    }

    private void decrChore() {
        this.choreCount.decrementAndGet();
    }
}
```

1279 红绿灯路口

这是两条路的交叉路口。第一条路是 `A` 路，车辆可沿 `1` 号方向由北向南行驶，也可沿 `2` 号方向由南向北行驶。第二条路是 `B` 路，车辆可沿 `3` 号方向由西向东行驶，也可沿 `4` 号方向由东向西行驶。

每条路在路口前都有一个红绿灯。红绿灯可以亮起红灯或绿灯。

> 绿灯表示两个方向的车辆都可通过路口。
> 红灯表示两个方向的车辆都不可以通过路口，必须等待绿灯亮起。

两条路上的红绿灯不可以同时为绿灯。这意味着，当 `A` 路上的绿灯亮起时，`B` 路上的红灯会亮起；当 `B` 路上的绿灯亮起时，`A` 路上的红灯会亮起.

开始时，`A` 路上的绿灯亮起，`B` 路上的红灯亮起。当一条路上的绿灯亮起时，所有车辆都可以从任意两个方向通过路口，直到另一条路上的绿灯亮起。不同路上的车辆不可以同时通过路口。

给这个路口设计一个没有死锁的红绿灯控制系统。

实现函数 `void carArrived(carId, roadId, direction, turnGreen, crossCar)` :

> `carId` 为到达车辆的编号。
> `roadId` 为车辆所在道路的编号。
> `direction` 为车辆的行进方向。
> `turnGreen` 是一个函数，调用此函数会使当前道路上的绿灯亮起。
> `crossCar` 是一个函数，调用此函数会允许车辆通过路口。

当你的答案避免了车辆在路口出现死锁，此答案会被认定为正确的。当路口已经亮起绿灯时仍打开绿灯，此答案会被认定为错误的。

示例 1:

```
输入: cars = [1,3,5,2,4], directions = [2,1,2,4,3], arrivalTimes =
[10,20,30,40,50]
输出: [
"Car 1 Has Passed Road A In Direction 2",    // A 路上的红绿灯为绿色，1 号车可通过路
口。
"Car 3 Has Passed Road A In Direction 1",    // 红绿灯仍为绿色，3 号车通过路口。
"Car 5 Has Passed Road A In Direction 2",    // 红绿灯仍为绿色，5 号车通过路口。
"Traffic Light On Road B Is Green",          // 2 号车在 B 路请求绿灯。
"Car 2 Has Passed Road B In Direction 4",    // B 路上的绿灯现已亮起，2 号车通过路口。
"Car 4 Has Passed Road B In Direction 3"     // 红绿灯仍为绿色，4 号车通过路口。
]
```

示例 2:

```
输入: cars = [1,2,3,4,5], directions = [2,4,3,3,1], arrivalTimes =
[10,20,30,40,40]
输出: [
"Car 1 Has Passed Road A In Direction 2",    // A 路上的红绿灯为绿色，1 号车可通过路
口。
"Traffic Light On Road B Is Green",          // 2 号车在 B 路请求绿灯。
"Car 2 Has Passed Road B In Direction 4",    // B 路上的绿灯现已亮起，2 号车通过路口。
"Car 3 Has Passed Road B In Direction 3",    // B 路上的绿灯现已亮起，3 号车通过路口。
"Traffic Light On Road A Is Green",          // 5 号车在 A 路请求绿灯。
"Car 5 Has Passed Road A In Direction 1",    // A 路上的绿灯现已亮起，5 号车通过路口。
"Traffic Light On Road B Is Green",          // 4 号车在 B 路请求绿灯。4 号车在路口等
灯，直到 5 号车通过路口，B 路的绿灯亮起。
"Car 4 Has Passed Road B In Direction 3"     // B 路上的绿灯现已亮起，4 号车通过路口。
]
```

解释：这是一个无死锁的方案。注意，在 A 路上的绿灯亮起、5 号车通过前让 4 号车通过，也是一个正确且可被接受的方案。

提示：

- 1 <= cars.length <= 20
- cars.length = directions.length
- cars.length = arrivalTimes.length
- cars 中的所有值都是唯一的。
- 1 <= directions[i] <= 4
- arrivalTimes 是非递减的。

```
class TrafficLight {
        private Semaphore greenLight;
        private boolean road1CanGo;//表示道路1是绿灯
        private boolean road2CanGo;//表示道路2是绿灯

        public TrafficLight() {
            this.greenLight = new Semaphore(1, true);
            this.road1CanGo = true;
            this.road2CanGo = false;
        }

        public void carArrived(
                int carId,           // ID of the car
```

```
                int roadId,          // ID of the road the car travels on. Can
be 1 (road A) or 2 (road B)
                int direction,       // Direction of the car
                Runnable turnGreen,  // Use turnGreen.run() to turn light to
green on current road
                Runnable crossCar    // Use crossCar.run() to make car cross the
intersection
        ) {
            try {
                greenLight.acquire();//申请获取遥控器
                //如果当前车道已经是绿灯了，直接通过
                if ((roadId == 1 && road1CanGo) || (roadId == 2 && road2CanGo))
crossCar.run();
                else if (roadId == 1 && !road1CanGo) {//否则，如果道路1不是绿灯，用遥
控器变成绿灯
                    turnGreen.run();
                    road1CanGo = true;
                    road2CanGo = false;
                    crossCar.run();
                } else if (roadId == 2 && !road2CanGo) {//如果道路2不是绿灯，用遥控
器变成绿灯
                    turnGreen.run();
                    road2CanGo = true;
                    road1CanGo = false;
                    crossCar.run();
                }
                greenLight.release();//最后把遥控器归还
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
}
```

阿里

1.两数之和

给定一个整数数组 nums 和一个目标值 target，请你在该数组中找出和为目标值的那 两个 整数，并返回他们的数组下标。你可以假设每种输入只会对应一个答案。但是，数组中同一个元素不能使用两遍。

```
示例:
给定 nums = [2, 7, 11, 15], target = 9
因为 nums[0] + nums[1] = 2 + 7 = 9
所以返回 [0, 1]

class Solution {
    public int[] twoSum(int[] nums, int target) {
        Map<Integer,Integer> map= new HashMap<>();

        for(int i = 0; i < nums.length;i++){
            int complement = target - nums[i];
```

```
            if (map.containsKey(complement)){
                return new int[]{map.get(complement),i};
            }
            map.put(nums[i],i);
        }
        throw new IllegalArgumentException("no solution");
    }
}

//  Map<Integer, Integer> map = new HashMap<>();
//         for (int i = 0; i < nums.length; i++) {
//             int complement = target - nums[i];
//             if (map.containsKey(complement)) {
//                 return new int[] { map.get(complement), i };
//             }
//             map.put(nums[i], i);
//         }
//         throw new IllegalArgumentException("No two sum solution");
```

1114 按序打印

我们提供了一个类：

```
public class Foo {
  public void first() { print("first"); }
  public void second() { print("second"); }
  public void third() { print("third"); }
}
```

三个不同的线程将会共用一个 `Foo` 实例。

```
    线程 A 将会调用 first() 方法
    线程 B 将会调用 second() 方法
    线程 C 将会调用 third() 方法
```

请设计修改程序，以确保 `second()` 方法在 `first()` 方法之后被执行，`third()` 方法在 `second()` 方法之后被执行。

示例 1:

```
输入: [1,2,3]
输出: "firstsecondthird"
解释:
有三个线程会被异步启动。
输入 [1,2,3] 表示线程 A 将会调用 first() 方法，线程 B 将会调用 second() 方法，线程 C 将会调用 third() 方法。
正确的输出是 "firstsecondthird"。
```

示例 2:

```
输入: [1,3,2]
输出: "firstsecondthird"
解释:
```

输入 [1,3,2] 表示线程 A 将会调用 first() 方法，线程 B 将会调用 third() 方法，线程 C 将会调用 second() 方法。
正确的输出是 "firstsecondthird"。

提示：

尽管输入中的数字似乎暗示了顺序，但是我们并不保证线程在操作系统中的调度顺序。
你看到的输入格式主要是为了确保测试的全面性。

```
class Foo {

    public Foo() {

    }

    private AtomicInteger one = new AtomicInteger(0);
    private AtomicInteger two = new AtomicInteger(0);

    public void first(Runnable printFirst) throws InterruptedException {

        // printFirst.run() outputs "first". Do not change or remove this line.
        printFirst.run();
        one.incrementAndGet();
    }

    public void second(Runnable printSecond) throws InterruptedException {
        while(one.get() != 1){

        }
        // printSecond.run() outputs "second". Do not change or remove this
line.
        printSecond.run();
        two.incrementAndGet();
    }

    public void third(Runnable printThird) throws InterruptedException {
        while(two.get() != 1){

        }
        // printThird.run() outputs "third". Do not change or remove this line.
        printThird.run();
    }
}


// private boolean firstFinished;
//     private boolean secondFinished;
//     private Object lock = new Object();

//     public Foo() {

//     }

//     public void first(Runnable printFirst) throws InterruptedException {

//         synchronized (lock) {
```

```
//             // printFirst.run() outputs "first". Do not change or remove this
line.
//             printFirst.run();
//             firstFinished = true;
//             lock.notifyAll();
//         }
//     }

//     public void second(Runnable printSecond) throws InterruptedException {

//         synchronized (lock) {
//             while (!firstFinished) {
//                 lock.wait();
//             }

//             // printSecond.run() outputs "second". Do not change or remove
this line.
//             printSecond.run();
//             secondFinished = true;
//             lock.notifyAll();
//         }
//     }

//     public void third(Runnable printThird) throws InterruptedException {

//         synchronized (lock) {
//            while (!secondFinished) {
//                 lock.wait();
//             }

//             // printThird.run() outputs "third". Do not change or remove this
line.
//             printThird.run();
//         }
//     }
```

193 有效电话号码

给定一个包含电话号码列表（一行一个电话号码）的文本文件 `file.txt`，写一个 `bash` 脚本输出所有有效的电话号码。

你可以假设一个有效的电话号码必须满足以下两种格式： `(xxx) xxx-xxxx` 或 `xxx-xxx-xxxx`。（`x` 表示一个数字）

你也可以假设每行前后没有多余的空格字符。

示例：

假设 `file.txt` 内容如下：

```
987-123-4567
123 456 7890
(123) 456-7890
```

你的脚本应当输出下列有效的电话号码：

```
987-123-4567
(123) 456-7890

# Read from the file file.txt and output all valid phone numbers to stdout.

grep -P '^([0-9]{3}-|\([0-9]{3}\) )[0-9]{3}-[0-9]{4}$' file.txt
```

206 反转链表

反转一个单链表。

示例:

输入: 1->2->3->4->5->NULL
输出: 5->4->3->2->1->NULL

进阶:
你可以迭代或递归地反转链表。你能否用两种方法解决这道题?

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode reverseList(ListNode head) {
        if(head == null || head.next == null){
            return head;
        }

        ListNode p = reverseList(head.next);
        head.next.next = head;
        head.next = null;
        return p;
    }
}



    // if (head == null || head.next == null) return head;
    // ListNode p = reverseList(head.next);
    // head.next.next = head;
    // head.next = null;
    // return p;
```

#716 最大栈

设计一个最大栈，支持 push、pop、top、peekMax 和 popMax 操作。

push(x) -- 将元素 x 压入栈中。
pop() -- 移除栈顶元素并返回这个值。
top() -- 返回栈顶元素。
peekMax() -- 返回栈中最大元素。
popMax() -- 返回栈中最大的元素，并将其删除。如果有多个最大元素，只要删除最靠近栈顶的那个。

样例 1:

```
MaxStack stack = new MaxStack();
stack.push(5);
stack.push(1);
stack.push(5);
stack.top(); -> 5
stack.popMax(); -> 5
stack.top(); -> 1
stack.peekMax(); -> 5
stack.pop(); -> 1
stack.top(); -> 5
```

注释:

-1e7 <= x <= 1e7
操作次数不会超过 10000。
当栈为空的时候不会出现后四个操作。

```
class MaxStack {

    Stack<Integer> stack;
    Stack<Integer> maxStack;

    public MaxStack() {
        stack = new Stack();
        maxStack = new Stack();
    }

    public void push(int x) {
        int max = maxStack.isEmpty() ? x : maxStack.peek();
        maxStack.push(max > x ? max : x);
        stack.push(x);
    }

    public int pop() {
        maxStack.pop();
        return stack.pop();
    }

    public int top() {
        return stack.peek();
    }

    public int peekMax() {
        return maxStack.peek();
    }
```

```
    public int popMax() {
        int max = peekMax();
        Stack<Integer> buffer = new Stack();
        while (top() != max) buffer.push(pop());
        pop();
        while (!buffer.isEmpty()) push(buffer.pop());
        return max;
    }

}

/**
 * Your MaxStack object will be instantiated and called as such:
 * MaxStack obj = new MaxStack();
 * obj.push(x);
 * int param_2 = obj.pop();
 * int param_3 = obj.top();
 * int param_4 = obj.peekMax();
 * int param_5 = obj.popMax();
 */
```

#53 最大子序和

给定一个整数数组 nums ，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

示例:

```
输入: [-2,1,-3,4,-1,2,1,-5,4]
输出: 6
解释: 连续子数组 [4,-1,2,1] 的和最大，为 6。
```

```
class Solution {
    public int maxSubArray(int[] nums) {
int ans = nums[0];
        int sum = 0;
        for(int num: nums) {
            if(sum > 0) {
                sum += num;
            } else {
                sum = num;
            }
            ans = Math.max(ans, sum);
        }
        return ans;


    }
}
```

#141 环形链表

给定一个链表，判断链表中是否有环。

为了表示给定链表中的环，我们使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 pos 是 -1，则在该链表中没有环。

示例 1：

输入：head = [3,2,0,-4], pos = 1
输出：true
解释：链表中有一个环，其尾部连接到第二个节点。

示例 2：

输入：head = [1,2], pos = 0
输出：true
解释：链表中有一个环，其尾部连接到第一个节点。

示例 3：

输入：head = [1], pos = -1
输出：false
解释：链表中没有环。

进阶：

你能用 O(1)（即，常量）内存解决此问题吗？

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public boolean hasCycle(ListNode head) {
        if(head == null || head.next == null){
            return false;
        }

        ListNode slow = head;
        ListNode fast = head.next;
        while(slow != fast){
            if(fast == null || fast.next== null){
                return false;
            }
            slow = slow.next;
            fast = fast.next.next;
        }

        return true;
    }
}
```

```
    // 哈希表
    // Set<ListNode> nodesSeen = new HashSet<>();
    // while (head != null) {
    //     if (nodesSeen.contains(head)) {
    //         return true;
    //     } else {
    //         nodesSeen.add(head);
    //     }
    //     head = head.next;
    // }
    // return false;




    // 快慢指针
    // if (head == null || head.next == null) {
    //     return false;
    // }
    // ListNode slow = head;
    // ListNode fast = head.next;
    // while (slow != fast) {
    //     if (fast == null || fast.next == null) {
    //         return false;
    //     }
    //     slow = slow.next;
    //     fast = fast.next.next;
    // }
    // return true;
```

#70 爬楼梯

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？

注意：给定 n 是一个正整数。

示例 1：

输入： 2
输出： 2
解释： 有两种方法可以爬到楼顶。
1. 1 阶 + 1 阶
2. 2 阶

示例 2：

输入： 3
输出： 3
解释： 有三种方法可以爬到楼顶。
1. 1 阶 + 1 阶 + 1 阶
2. 1 阶 + 2 阶
3. 2 阶 + 1 阶

```
class Solution {
```

```
    public int climbStairs(int n) {
        int[] dp = new int[n+1];
        dp[0] = 1;
        dp[1] = 1;
        for(int i = 2; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }

        return dp[n];
    }
}
```

#21 合并两个有序链表

将两个升序链表合并为一个新的 升序 链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

示例：

输入：1->2->4, 1->3->4
输出：1->1->2->3->4->4

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
                if (l1 == null) {
            return l2;
        }
        else if (l2 == null) {
            return l1;
        }
        else if (l1.val < l2.val) {
            l1.next = mergeTwoLists(l1.next, l2);
            return l1;
        }
        else {
            l2.next = mergeTwoLists(l1, l2.next);
            return l2;
        }
    }
}


        // ListNode prehead = new ListNode(-1);

        // ListNode prev = prehead;
        // while (l1 != null && l2 != null) {
```

```
        //     if (l1.val <= l2.val) {
        //         prev.next = l1;
        //         l1 = l1.next;
        //     } else {
        //         prev.next = l2;
        //         l2 = l2.next;
        //     }
        //     prev = prev.next;
        // }

        // // 合并后 l1 和 l2 最多只有一个还未被合并完，我们直接将链表末尾指向未合并完的链表即可
        // prev.next = l1 == null ? l2 : l1;

        // return prehead.next;
```

#7 整数反转

给出一个 32 位的有符号整数，你需要将这个整数中每位上的数字进行反转。

示例 1:

输入: 123
输出: 321

 示例 2:

输入: -123
输出: -321

示例 3:

输入: 120
输出: 21

注意:

假设我们的环境只能存储得下 32 位的有符号整数，则其数值范围为 [−231, 231 − 1]。请根据这个假设，如果反转后整数溢出那么就返回 0。

```
class Solution {
    public int reverse(int x) {
    int ans = 0;
    while (x != 0) {
        if ((ans * 10) / 10 != ans) {
            ans = 0;
            break;
        }
        ans = ans * 10 + x % 10;
        x = x / 10;
    }
    return ans;
    }
}。
```

#572 另一个树的子树

给定两个非空二叉树 s 和 t，检验 s 中是否包含和 t 具有相同结构和节点值的子树。s 的一个子树包括 s 的一个节点和这个节点的所有子孙。s 也可以看做它自身的一棵子树。

示例 1:
给定的树 s:

给定的树 t:

返回 true，因为 t 与 s 的一个子树拥有相同的结构和节点值。

示例 2:
给定的树 s:

给定的树 t:

返回 false。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public boolean isSubtree(TreeNode s, TreeNode t) {
        if (t == null) return true;   // t 为 null 一定都是 true
```

```
        if (s == null) return false;  // 这里 t 一定不为 null，只要 s 为 null，肯定是 false
        return isSubtree(s.left, t) || isSubtree(s.right, t) || isSameTree(s,t);
    }

    /**
     * 判断两棵树是否相同
     */
    public boolean isSameTree(TreeNode s, TreeNode t){
        if (s == null && t == null) return true;
        if (s == null || t == null) return false;
        if (s.val != t.val) return false;
        return isSameTree(s.left, t.left) && isSameTree(s.right, t.right);
    }
}
```

#9 回文数

判断一个整数是否是回文数。回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。

示例 1:

输入: 121
输出: true

示例 2:

输入: -121
输出: false
解释: 从左向右读, 为 -121 。 从右向左读, 为 121- 。因此它不是一个回文数。

示例 3:

输入: 10
输出: false
解释: 从右向左读, 为 01 。因此它不是一个回文数。

进阶:

你能不将整数转为字符串来解决这个问题吗？

```
class Solution {
    public boolean isPalindrome(int x) {
            String reverse = new StringBuilder(x+"").reverse().toString();
            return (x+"").equals(reverse);
    }


}



    // public boolean isPalindrome(int x) {
    //     //思考：这里大家可以思考一下，为什么末尾为 0 就可以直接返回 false
    //     if (x < 0 || (x % 10 == 0 && x != 0)) return false;
    //     int revertedNumber = 0;
    //     while (x > revertedNumber) {
```

```
    //         revertedNumber = revertedNumber * 10 + x % 10;
    //         x /= 10;
    //     }
    //     return x == revertedNumber || x == revertedNumber / 10;
    // }
```

#344 反转字符

编写一个函数，其作用是将输入的字符串反转过来。输入字符串以字符数组 `char[]` 的形式给出。

不要给另外的数组分配额外的空间，你必须原地修改输入数组、使用 O(1) 的额外空间解决这一问题。

你可以假设数组中的所有字符都是 ASCII 码表中的可打印字符。

示例 1：

输入：["h","e","l","l","o"]
输出：["o","l","l","e","h"]

示例 2：

输入：["H","a","n","n","a","h"]
输出：["h","a","n","n","a","H"]

```
class Solution {
    public void reverseString(char[] s) {
        //双指针
        if(s == null || s.length < 2){
            return;
        }

        int left = -1;
        int right = s.length;

        while(++left<--right){
            char c = s[left];
            s[left] = s[right];
            s[right] = c;
        }

        return;
    }
}
```

```
        // if (s == null || s.length < 2) {
        //     return;
        // }
        // int left = -1;
        // int right = s.length;
        // while (++left < --right) {
        //     char c = s[left];
        //     s[left] = s[right];
        //     s[right] = c;
```

```
        // }

        // return;
```

#266 回文排列

给定一个字符串，判断该字符串中是否可以通过重新排列组合，形成一个回文字符串。

示例 1：

输入: "code"
输出: false

示例 2：

输入: "aab"
输出: true

示例 3：

输入: "carerac"
输出: true

```
class Solution {
    public boolean canPermutePalindrome(String s) {
        Set<Character> set = new HashSet<>();
        for(int i =0;i<s.length();i++){
            if( !set.add(s.charAt(i))){
                set.remove(s.charAt(i));
            }
        }

        return set.size() <= 1;
    }
}
```

#276 栅栏涂色

有 k 种颜色的涂料和一个包含 n 个栅栏柱的栅栏，每个栅栏柱可以用其中一种颜色进行上色。

你需要给所有栅栏柱上色，并且保证其中相邻的栅栏柱 最多连续两个 颜色相同。然后，返回所有有效涂色的方案数。

注意:
n 和 k 均为非负的整数。

示例:

输入: n = 3, k = 2
输出: 6
解析: 用 c1 表示颜色 1，c2 表示颜色 2，所有可能的涂色方案有:

```
            柱 1    柱 2   柱 3
 -----      -----  -----  -----
   1         c1     c1     c2
   2         c1     c2     c1
   3         c1     c2     c2
```

```
   4         c2     c1     c1
   5         c2     c1     c2
   6         c2     c2     c1

class Solution {
    public int numWays(int n, int k) {
        if (n == 0 || k == 0) {
            return 0;
        }
        if (n == 1) {
            return k;
        }
        if (n == 2) {
            return k * k;
        }
        int[] dp = new int[n];
        dp[0] = k;
        dp[1] = k * k;
        for (int i = 2; i < n; i++) {
            dp[i] = dp[i - 2] * (k - 1) + dp[i - 1] * (k - 1);
        }
        return dp[n - 1];

    }
}
```

#1078 Bigram 分词

给出第一个词 first 和第二个词 second，考虑在某些文本 text 中可能以 "first second third" 形式出现的情况，其中 second 紧随 first 出现，third 紧随 second 出现。

对于每种这样的情况，将第三个词 "third" 添加到答案中，并返回答案。

示例 1：

输入：text = "alice is a good girl she is a good student", first = "a", second = "good"
输出：["girl","student"]

示例 2：

输入：text = "we will we will rock you", first = "we", second = "will"
输出：["we","rock"]

提示：

- 1 <= text.length <= 1000
- text 由一些用空格分隔的单词组成，每个单词都由小写英文字母组成
- 1 <= first.length, second.length <= 10
- first 和 second 由小写英文字母组成

```
class Solution {
    public String[] findOcurrences(String text, String first, String second) {
```

```
        String[] word = text.split("\\s");
        List<String> list = new ArrayList<>();
        for(int i=0;i<word.length -2;i++){
            if(word[i].equals(first)&&word[i+1].equals(second)){
                list.add(word[i+2]);
            }
        }

        return list.toArray(new String[list.size()]);
    }
}
```

#160 相交链表

编写一个程序，找到两个单链表相交的起始节点。

如下面的两个链表：

在节点 c1 开始相交。

示例 1：

输入：intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
输出：Reference of the node with value = 8
输入解释：相交节点的值为 8 （注意，如果两个链表相交则不能为 0）。从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。

示例 2：

输入：intersectVal = 2, listA = [0,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1
输出：Reference of the node with value = 2
输入解释：相交节点的值为 2 （注意，如果两个链表相交则不能为 0）。从各自的表头开始算起，链表 A 为 [0,9,1,2,4]，链表 B 为 [3,2,4]。在 A 中，相交节点前有 3 个节点；在 B 中，相交节点前有 1 个节点。

示例 3：

输入：intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2
输出：null
输入解释：从各自的表头开始算起，链表 A 为 [2,6,4]，链表 B 为 [1,5]。由于这两个链表不相交，所以 intersectVal 必须为 0，而 skipA 和 skipB 可以是任意值。
解释：这两个链表不相交，因此返回 null。

注意：

- 如果两个链表没有交点，返回 null.

在返回结果后，两个链表仍须保持原有的结构。
可假定整个链表结构中没有循环。
程序尽量满足 O(n) 时间复杂度，且仅用 O(1) 内存

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        if(headA == null || headB == null){
            return null;
        }

        ListNode a = headA, b = headB;
        while (a != b){
            a = a == null ? headB : a.next;
            b = b == null ? headA : b.next;
        }

        return a;
    }




    // public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
    // if (headA == null || headB == null) return null;
    // ListNode pA = headA, pB = headB;
    // while (pA != pB) {
    //     pA = pA == null ? headB : pA.next;
    //     pB = pB == null ? headA : pB.next;
    // }
    // return pA;
}
```

#706 设计哈希映射

不使用任何内建的哈希表库设计一个哈希映射

具体地说，你的设计应该包含以下的功能

put(key, value)：向哈希映射中插入(键,值)的数值对。如果键对应的值已经存在，更新这个值。
get(key)：返回给定的键所对应的值，如果映射中不包含这个键，返回-1。
remove(key)：如果映射中存在这个键，删除这个数值对。

示例：

```
MyHashMap hashMap = new MyHashMap();
hashMap.put(1, 1);
hashMap.put(2, 2);
hashMap.get(1);            // 返回 1
hashMap.get(3);            // 返回 -1 (未找到)
hashMap.put(2, 1);         // 更新已有的值
hashMap.get(2);            // 返回 1
hashMap.remove(2);         // 删除键为2的数据
hashMap.get(2);            // 返回 -1 (未找到)
```

注意：

- 所有的值都在 [0, 1000000]的范围内。
- 操作的总数目在[1, 10000]范围内。
- 不要使用内建的哈希库。

```
class Pair<U, V> {
  public U first;
  public V second;

  public Pair(U first, V second) {
    this.first = first;
    this.second = second;
  }
}


class Bucket {
  private List<Pair<Integer, Integer>> bucket;

  public Bucket() {
    this.bucket = new LinkedList<Pair<Integer, Integer>>();
  }

  public Integer get(Integer key) {
    for (Pair<Integer, Integer> pair : this.bucket) {
      if (pair.first.equals(key))
        return pair.second;
    }
    return -1;
  }

  public void update(Integer key, Integer value) {
    boolean found = false;
    for (Pair<Integer, Integer> pair : this.bucket) {
      if (pair.first.equals(key)) {
        pair.second = value;
        found = true;
      }
    }
    if (!found)
      this.bucket.add(new Pair<Integer, Integer>(key, value));
  }

  public void remove(Integer key) {
    for (Pair<Integer, Integer> pair : this.bucket) {
      if (pair.first.equals(key)) {
```

```
        this.bucket.remove(pair);
        break;
      }
    }
  }
}

class MyHashMap {
  private int key_space;
  private List<Bucket> hash_table;

  /** Initialize your data structure here. */
  public MyHashMap() {
    this.key_space = 2069;
    this.hash_table = new ArrayList<Bucket>();
    for (int i = 0; i < this.key_space; ++i) {
      this.hash_table.add(new Bucket());
    }
  }

  /** value will always be non-negative. */
  public void put(int key, int value) {
    int hash_key = key % this.key_space;
    this.hash_table.get(hash_key).update(key, value);
  }

  /**
   * Returns the value to which the specified key is mapped, or -1 if this map
contains no mapping
   * for the key
   */
  public int get(int key) {
    int hash_key = key % this.key_space;
    return this.hash_table.get(hash_key).get(key);
  }

  /** Removes the mapping of the specified value key if this map contains a
mapping for the key */
  public void remove(int key) {
    int hash_key = key % this.key_space;
    this.hash_table.get(hash_key).remove(key);
  }
}

/**
 * Your MyHashMap object will be instantiated and called as such: MyHashMap obj
= new MyHashMap();
 * obj.put(key,value); int param_2 = obj.get(key); obj.remove(key);
 */
```

#387 字符串中的第一个唯一字符

给定一个字符串，找到它的第一个不重复的字符，并返回它的索引。如果不存在，则返回 -1。

示例：

```
s = "leetcode"
返回 0

s = "loveleetcode"
返回 2
```

```
class Solution {
    public int firstUniqChar(String s) {
        int[] result = new int[26];
        for(int i =0;i<s.length();i++){
            result[s.charAt(i) - 'a']++;
        }

        for(int i=0; i <s.length(); i++){
            if(result[s.charAt(i) - 'a'] == 1){
                return i;
            }
        }

        return -1;
    }
}
```

#136 只出现一次的数字

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

说明：

你的算法应该具有线性时间复杂度。 你可以不使用额外空间来实现吗？

示例 1:

```
输入: [2,2,1]
输出: 1
```

示例 2:

```
输入: [4,1,2,1,2]
输出: 4
```

```
class Solution {
    public int singleNumber(int[] nums) {
       int ans = 0;
        for(int num: nums) {
            ans ^= num;
        }
        return ans;

    }
```

#196 删除重复的电子邮箱

编写一个 SQL 查询，来删除 Person 表中所有重复的电子邮箱，重复的邮箱里只保留 Id 最小 的那个。

```
+----+------------------+
| Id | Email            |
+----+------------------+
| 1  | john@example.com |
| 2  | bob@example.com  |
| 3  | john@example.com |
+----+------------------+
```

Id 是这个表的主键。

例如，在运行你的查询语句之后，上面的 Person 表应返回以下几行:

```
+----+------------------+
| Id | Email            |
+----+------------------+
| 1  | john@example.com |
| 2  | bob@example.com  |
+----+------------------+
```

```
# Write your MySQL query statement below

-- delete p1 from person p1 left join person p2 on p1.email = p2.email where
p1.id > p2.id

delete p1 from person p1,person p2 where p1.email= p2.email and p1.id > p2.id
```

#876 链表的中间结点

给定一个带有头结点 head 的非空单链表，返回链表的中间结点。

如果有两个中间结点，则返回第二个中间结点。

示例 1：

```
输入：[1,2,3,4,5]
输出：此列表中的结点 3 (序列化形式：[3,4,5])
返回的结点值为 3 。 (测评系统对该结点序列化表述是 [3,4,5])。
注意，我们返回了一个 ListNode 类型的对象 ans，这样：
ans.val = 3, ans.next.val = 4, ans.next.next.val = 5, 以及 ans.next.next.next =
NULL.
```

示例 2：

```
输入：[1,2,3,4,5,6]
输出：此列表中的结点 4 (序列化形式：[4,5,6])
由于该列表有两个中间结点，值分别为 3 和 4，我们返回第二个结点。
```

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
```

```
 */
class Solution {
    public ListNode middleNode(ListNode head) {
        ListNode[] A= new ListNode[100];
        int t = 0;
        while (head != null){
            A[t++] = head;
            head = head.next;
        }

        return A[t/2];
    }
}
```

#409 最长回文串

给定一个包含大写字母和小写字母的字符串，找到通过这些字母构造成的最长的回文串。

在构造过程中，请注意区分大小写。比如 "Aa" 不能当做一个回文字符串。

注意:
假设字符串的长度不会超过 1010。

示例 1:

输入:
"abccccdd"

输出:
7

解释:
我们可以构造的最长的回文串是"dccaccd", 它的长度是 7。

```
class Solution {
    public int longestPalindrome(String s) {
        int[] count = new int[128];
        for (char c: s.toCharArray())
            count[c]++;

        int ans = 0;
        for (int v: count) {
            ans += v / 2 * 2;
            if (v % 2 == 1 && ans % 2 == 0)
                ans++;
        }
        return ans;

    }
}
```

#198 打家劫舍

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你 不触动警报装置的情况下 ，一夜之内能够偷窃到的最高金额。

示例 1：

输入：[1,2,3,1]
输出：4
解释：偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。
      偷窃到的最高金额 = 1 + 3 = 4 。

示例 2：

输入：[2,7,9,3,1]
输出：12
解释：偷窃 1 号房屋 (金额 = 2)，偷窃 3 号房屋 (金额 = 9)，接着偷窃 5 号房屋 (金额 = 1)。
      偷窃到的最高金额 = 2 + 9 + 1 = 12 。

```
class Solution {
       public int rob(int[] nums) {
        int len = nums.length;
        if(len == 0)
            return 0;
        int[] dp = new int[len + 1];
        dp[0] = 0;
        dp[1] = nums[0];
        for(int i = 2; i <= len; i++) {
            dp[i] = Math.max(dp[i-1], dp[i-2] + nums[i-1]);
        }
        return dp[len];
    }
}
```

#283 移动零

给定一个数组 nums，编写一个函数将所有 0 移动到数组的末尾，同时保持非零元素的相对顺序。

示例:

输入: [0,1,0,3,12]
输出: [1,3,12,0,0]

说明:

    必须在原数组上操作，不能拷贝额外的数组。
    尽量减少操作次数。

```
class Solution {
    public void moveZeroes(int[] nums) {
        if(null == nums){
            return;
        }

        int j= 0;
        // for(int i=0;i< nums.length;i++){
        //     if(nums[i] != 0){
```

```
        //         int temp = nums[i];
        //         nums[i] = nums[j];
        //         nums[j++] = temp;
        //     }

        for(int i =0;i<nums.length;i++){
            if(nums[i] != 0){
                nums[j++] = nums[i];
            }
        }

        for(int i=j;i<nums.length;i++){
            nums[i] = 0;
        }
        // }
    }
}
```

#182 查找重复的电子邮箱

编写一个 SQL 查询，查找 Person 表中所有重复的电子邮箱。

示例：

```
+----+---------+
| Id | Email   |
+----+---------+
| 1  | a@b.com |
| 2  | c@d.com |
| 3  | a@b.com |
+----+---------+
```

根据以上输入，你的查询应返回以下结果：

```
+---------+
| Email   |
+---------+
| a@b.com |
+---------+
```

说明：所有电子邮箱都是小写字母。

```
# Write your MySQL query statement below
select Email from person group by email having count(Email) > 1
```

#415 字符串相加

给定两个字符串形式的非负整数 num1 和num2 ，计算它们的和。

提示：

- num1 和num2 的长度都小于 5100
- num1 和num2 都只包含数字 0-9
- num1 和num2 都不包含任何前导零
- 你不能使用任何内建 BigInteger 库， 也不能直接将输入的字符串转换为整数形式

```java
class Solution {
    public String addStrings(String num1, String num2) {
       StringBuilder res = new StringBuilder("");
        int i = num1.length() - 1, j = num2.length() - 1, carry = 0;
        while(i >= 0 || j >= 0){
            int n1 = i >= 0 ? num1.charAt(i) - '0' : 0;
            int n2 = j >= 0 ? num2.charAt(j) - '0' : 0;
            int tmp = n1 + n2 + carry;
            carry = tmp / 10;
            res.append(tmp % 10);
            i--; j--;
        }
        if(carry == 1) res.append(1);
        return res.reverse().toString();
    }
}
```

面试题 01.01 判定字符是否唯一

实现一个算法，确定一个字符串 `s` 的所有字符是否全都不同。

示例 1：

```
输入: s = "leetcode"
输出: false
```

示例 2：

```
输入: s = "abc"
输出: true
```

```java
class Solution {
    public boolean isUnique(String astr) {
        for(char a: astr.toCharArray()){
            if(astr.indexOf(a) != astr.lastIndexOf(a)){
                return false;
            }
        }
        return true;
    }
}
```

中高难度

#1236 网络爬虫

给定一个链接 `startUrl` 和一个接口 `HtmlParser` ，请你实现一个网络爬虫，以实现爬取同 `startUrl` 拥有相同 域名标签 的全部链接。该爬虫得到的全部链接可以 任何顺序 返回结果。

你的网络爬虫应当按照如下模式工作：

- 自链接 `startUrl` 开始爬取
- 调用 `HtmlParser.getUrls(url)` 来获得链接`url`页面中的全部链接
- 同一个链接最多只爬取一次
- 只输出 域名 与 `startUrl` 相同 的链接集合

如上所示的一个链接，其域名为 example.org。简单起见，你可以假设所有的链接都采用 http协议 并没有指定 端口。例如，链接 http://leetcode.com/problems 和 http://leetcode.com/contest 是同一个域名下的，而链接http://example.org/test 和 http://example.com/abc 是不在同一域名下的。

HtmlParser 接口定义如下：

```
interface HtmlParser {
  // 返回给定 url 对应的页面中的全部 url 。
  public List<String> getUrls(String url);
}
```

下面是两个实例，用以解释该问题的设计功能，对于自定义测试，你可以使用三个变量 urls，edges 和 startUrl。注意在代码实现中，你只可以访问 startUrl ，而 urls 和 edges 不可以在你的代码中被直接访问。

示例 1：

输入：
```
urls = [
  "http://news.yahoo.com",
  "http://news.yahoo.com/news",
  "http://news.yahoo.com/news/topics/",
  "http://news.google.com",
  "http://news.yahoo.com/us"
]
edges = [[2,0],[2,1],[3,2],[3,1],[0,4]]
startUrl = "http://news.yahoo.com/news/topics/"
```
输出：
```
[
  "http://news.yahoo.com",
  "http://news.yahoo.com/news",
  "http://news.yahoo.com/news/topics/",
  "http://news.yahoo.com/us"
]
```

示例 2：

输入：
```
urls = [
  "http://news.yahoo.com",
  "http://news.yahoo.com/news",
  "http://news.yahoo.com/news/topics/",
  "http://news.google.com"
]
edges = [[0,2],[2,1],[3,2],[3,1],[3,0]]
startUrl = "http://news.google.com"
```
输入：["http://news.google.com"]
解释：startUrl 链接到所有其他不共享相同主机名的页面。

提示：

```
    1 <= urls.length <= 1000
    1 <= urls[i].length <= 300
```

```
    startUrl 为 urls 中的一个。
    域名标签的长为1到63个字符（包括点），只能包含从‘a’到‘z’的ASCII字母、‘0’到‘9’的数字以及连字符即减号（‘-’）。
    域名标签不会以连字符即减号（‘-’）开头或结尾。
    关于域名有效性的约束可参考:
 https://en.wikipedia.org/wiki/Hostname#Restrictions_on_valid_hostnames
    你可以假定url库中不包含重复项。

/**
 * // This is the HtmlParser's API interface.
 * // You should not implement it, or speculate about its implementation
 * interface HtmlParser {
 *     public List<String> getUrls(String url) {}
 * }
 */

class Solution {
public List<String> crawl(String startUrl, HtmlParser htmlParser) {
        Queue<String> bfsQ = new LinkedList<>();
        Set<String> urlVisited = new HashSet<String>();
        urlVisited.add(startUrl);
        bfsQ.add(startUrl);
        while (!bfsQ.isEmpty()) {
            String currentUrl = bfsQ.poll();
            List<String> nextUrls = htmlParser.getUrls(currentUrl);
            for (String nextUrl : nextUrls) {
                if (!urlVisited.contains(nextUrl) &&
getHostName(nextUrl).equals(getHostName(currentUrl))) {
                    urlVisited.add(nextUrl);
                    bfsQ.add(nextUrl);
                }
            }
        }
        List<String> ans = new ArrayList<>();
        String startHostName = getHostName(startUrl);
        for (String url : urlVisited) {
            if (startHostName.equals(getHostName(url))) {
                ans.add(url);
            }
        }

        return ans;
    }
    private String getHostName(String url) {
        url = url.substring(7);
        if (url.contains("/")) {
            int end = url.indexOf('/');
            return url.substring(0, end);
        } else {
            return url;
        }

    }

}
```

#1116 打印零与奇偶数

假设有这么一个类：

```
class ZeroEvenOdd {
  public ZeroEvenOdd(int n) { ... }      // 构造函数
  public void zero(printNumber) { ... }  // 仅打印出 0
  public void even(printNumber) { ... }  // 仅打印出 偶数
  public void odd(printNumber) { ... }   // 仅打印出 奇数
}
```

相同的一个 ZeroEvenOdd 类实例将会传递给三个不同的线程：

线程 A 将调用 zero()，它只输出 0 。
线程 B 将调用 even()，它只输出偶数。
线程 C 将调用 odd()，它只输出奇数。

每个线程都有一个 printNumber 方法来输出一个整数。请修改给出的代码以输出整数序列 010203040506... ，其中序列的长度必须为 2n。

示例 1：

输入：n = 2
输出："0102"
说明：三条线程异步执行，其中一个调用 zero()，另一个线程调用 even()，最后一个线程调用odd()。正确的输出为 "0102"。

示例 2：

输入：n = 5
输出："0102030405"

```
class ZeroEvenOdd {
    private int n;

    public ZeroEvenOdd(int n) {
        this.n = n;
    }

    // printNumber.accept(x) outputs "x", where x is an integer.
    Semaphore z = new Semaphore(1);
    Semaphore e = new Semaphore(0);
    Semaphore o = new Semaphore(0);

    public void zero(IntConsumer printNumber) throws InterruptedException {
        for(int i=0; i<n;i++) {
            z.acquire();
            printNumber.accept(0);
            if((i&1)==0) {
                o.release();
            }else {
                e.release();
            }
        }
    }

    public void even(IntConsumer printNumber) throws InterruptedException {
```

```
        for(int i=2; i<=n; i+=2) {
            e.acquire();
            printNumber.accept(i);
            z.release();
        }
    }

    public void odd(IntConsumer printNumber) throws InterruptedException {
        for(int i=1; i<=n; i+=2) {
            o.acquire();
            printNumber.accept(i);
            z.release();
        }
    }


}
```

#1188 设计有限阻塞队列

实现一个拥有如下方法的线程安全有限阻塞队列：

`BoundedBlockingQueue(int capacity)` 构造方法初始化队列，其中`capacity`代表队列长度上限。
`void enqueue(int element)` 在队首增加一个`element`. 如果队列满，调用线程被阻塞直到队列非满。
`int dequeue()` 返回队尾元素并从队列中将其删除. 如果队列为空，调用线程被阻塞直到队列非空。
`int size()` 返回当前队列元素个数。

你的实现将会被多线程同时访问进行测试。每一个线程要么是一个只调用`enqueue`方法的生产者线程，要么是一个只调用`dequeue`方法的消费者线程。`size`方法将会在每一个测试用例之后进行调用。

请不要使用内置的有限阻塞队列实现，否则面试将不会通过。

示例 1:

输入:
```
1
1
["BoundedBlockingQueue","enqueue","dequeue","dequeue","enqueue","enqueue","enqueue","enqueue","dequeue"]
[[2],[1],[],[],[0],[2],[3],[4],[]]
```

输出:
```
[1,0,2,2]
```

解释:
生产者线程数目 = 1
消费者线程数目 = 1

```
BoundedBlockingQueue queue = new BoundedBlockingQueue(2);   // 使用capacity = 2初始化队列。

queue.enqueue(1);   // 生产者线程将1插入队列。
queue.dequeue();    // 消费者线程调用dequeue并返回1。
```

```
queue.dequeue();    // 由于队列为空，消费者线程被阻塞。
queue.enqueue(0);   // 生产者线程将0插入队列。消费者线程被解除阻塞同时将0弹出队列并返回。
queue.enqueue(2);   // 生产者线程将2插入队列。
queue.enqueue(3);   // 生产者线程将3插入队列。
queue.enqueue(4);   // 生产者线程由于队列长度已达到上限2而被阻塞。
queue.dequeue();    // 消费者线程将2从队列弹出并返回。生产者线程解除阻塞同时将4插入队列。
queue.size();       // 队列中还有2个元素。size()方法在每组测试用例最后调用。
```

示例 2:

输入:
3
4
["BoundedBlockingQueue","enqueue","enqueue","enqueue","dequeue","dequeue","dequeue","enqueue"]
[[3],[1],[0],[2],[],[],[],[3]]

输出:
[1,0,2,1]

解释:
生产者线程数目 = 3
消费者线程数目 = 4

```
BoundedBlockingQueue queue = new BoundedBlockingQueue(3);   // 使用capacity = 3初始化队列。

queue.enqueue(1);   // 生产者线程P1将1插入队列。
queue.enqueue(0);   // 生产者线程P2将0插入队列。
queue.enqueue(2);   // 生产者线程P3将2插入队列。
queue.dequeue();    // 消费者线程C1调用dequeue。
queue.dequeue();    // 消费者线程C2调用dequeue。
queue.dequeue();    // 消费者线程C3调用dequeue。
queue.enqueue(3);   // 其中一个生产者线程将3插入队列。
queue.size();       // 队列中还有1个元素。
```

由于生产者/消费者线程的数目可能大于1，我们并不知道线程如何被操作系统调度，即使输入看上去隐含了顺序。因此任意一种输出[1,0,2]或[1,2,0]或[0,1,2]或[0,2,1]或[2,0,1]或[2,1,0]都可被接受。

```
import java.util.LinkedList;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

class BoundedBlockingQueue {


    //原子类保证原子性，也可以使用volatile
    //普通的int被读取，会被读入内存的缓存中，完成加减乘除后再放回内存中，而每一个线程都有自己的寄存器，这样子会导致可能读取不到最新的数据
    //volatile则可以直接在主内存读写，当一个线程更新了值，其他线程能够及时获知。
    AtomicInteger size = new AtomicInteger(0);
    private volatile int capacity;
    //自己实现阻塞队列，需要一个容器，内部实现了一个node，如果改造为不只是int的，使用T泛型
    private LinkedList<Integer> container;
```

```
    //可重入锁
    private static ReentrantLock lock = new ReentrantLock();
    Condition procuder = lock.newCondition();//用来通知生产（入队）线程等待await还是可以执行signal
    Condition consumer = lock.newCondition();//用来通知消费（出队）线程等待await还是可以执行signal

    public BoundedBlockingQueue(int capacity) {
        this.capacity = capacity;
        container = new LinkedList<>();
    }

    /**
     * 入队
     *
     * @param element
     * @throws InterruptedException
     */
    public void enqueue(int element) throws InterruptedException {
        //每一个线程都会获得锁，但是如果条件不满足则会阻塞
        lock.lock();
        try {
            //阻塞的话必须用循环，让这个线程再次获得cpu片段的时候能够够执行
            while (size.get() >= capacity) {
                //入队线程阻塞，把锁释放？
                procuder.await();
            }
            container.addFirst(element);
            size.incrementAndGet();

            //通知出队线程
            consumer.signal();
        } finally {
            lock.unlock();
        }
    }

    public int dequeue() throws InterruptedException {
        lock.lock();
        try {
            while (size.get() == 0) {
                consumer.await();
            }
            int lastValue = container.getLast();
            container.removeLast();
            size.decrementAndGet();

            //通知入队线程
            procuder.signal();
            return lastValue;
        } finally {
            lock.unlock();
        }
    }

    public int size() {
        lock.lock();
        try {
```

```
            return size.get();
        } finally {
            lock.unlock();
        }
    }
}
```

#1115 交替打印FooBar

我们提供一个类：

```
class FooBar {
  public void foo() {
    for (int i = 0; i < n; i++) {
      print("foo");
    }
  }

  public void bar() {
    for (int i = 0; i < n; i++) {
      print("bar");
    }
  }
}
```

两个不同的线程将会共用一个 `FooBar` 实例。其中一个线程将会调用 `foo()` 方法，另一个线程将会调用 `bar()` 方法。

请设计修改程序，以确保 `"foobar"` 被输出 `n` 次。

示例 1:

输入: n = 1
输出: "foobar"
解释: 这里有两个线程被异步启动。其中一个调用 `foo()` 方法，另一个调用 `bar()` 方法，"foobar" 将被输出一次。

示例 2:

输入: n = 2
输出: "foobarfoobar"
解释: "foobar" 将被输出两次。

```
class FooBar {
    private int n;

    public FooBar(int n) {
        this.n = n;
    }

    Semaphore foo = new Semaphore(1);
    Semaphore bar = new Semaphore(0);

    public void foo(Runnable printFoo) throws InterruptedException {
        for (int i = 0; i < n; i++) {
```

```
            foo.acquire();
            printFoo.run();
            bar.release();
        }
    }

    public void bar(Runnable printBar) throws InterruptedException {
        for (int i = 0; i < n; i++) {
            bar.acquire();
            printBar.run();
            foo.release();
        }
    }
}
```

#192 统计词频

写一个 bash 脚本以统计一个文本文件 words.txt 中每个单词出现的频率。

为了简单起见，你可以假设：

    words.txt只包括小写字母和 ' ' 。
    每个单词只由小写字母组成。
    单词间由一个或多个空格字符分隔。

示例:

假设 words.txt 内容如下：

```
the day is sunny the the
the sunny is is
```

你的脚本应当输出（以词频降序排列）：

```
the 4
is 3
sunny 2
day 1
```

说明:

    不要担心词频相同的单词的排序问题，每个单词出现的频率都是唯一的。
    你可以使用一行 Unix pipes 实现吗？

```
# Read from the file words.txt and output the word frequency list to stdout.
cat words.txt|tr -s ' ' '\n'|sort |uniq -c|sort -r|awk '{print $2,$1}'
```

#174 地下城游戏

一些恶魔抓住了公主（P）并将她关在了地下城的右下角。地下城是由 M x N 个房间组成的二维网格。我们英勇的骑士（K）最初被安置在左上角的房间里，他必须穿过地下城并通过对抗恶魔来拯救公主。

骑士的初始健康点数为一个正整数。如果他的健康点数在某一时刻降至 0 或以下，他会立即死亡。

有些房间由恶魔守卫，因此骑士在进入这些房间时会失去健康点数（若房间里的值为负整数，则表示骑士将损失健康点数）；其他房间要么是空的（房间里的值为 0），要么包含增加骑士健康点数的魔法球（若房间里的值为正整数，则表示骑士将增加健康点数）。

为了尽快到达公主，骑士决定每次只向右或向下移动一步。

编写一个函数来计算确保骑士能够拯救到公主所需的最低初始健康点数。

例如，考虑到如下布局的地下城，如果骑士遵循最佳路径 右 -> 右 -> 下 -> 下，则骑士的初始健康点数至少为 7。

```
-2 (K)  -3  3
-5  -10     1
10  30  -5 (P)
```

说明：

骑士的健康点数没有上限。
任何房间都可能对骑士的健康点数造成威胁，也可能增加骑士的健康点数，包括骑士进入的左上角房间以及公主被监禁的右下角房间。

```java
class Solution {
    public int calculateMinimumHP(int[][] dungeon) {
        int n = dungeon.length, m = dungeon[0].length;
        int[][] dp = new int[n + 1][m + 1];
        for (int i = 0; i <= n; ++i) {
            Arrays.fill(dp[i], Integer.MAX_VALUE);
        }
        dp[n][m - 1] = dp[n - 1][m] = 1;
        for (int i = n - 1; i >= 0; --i) {
            for (int j = m - 1; j >= 0; --j) {
                int minn = Math.min(dp[i + 1][j], dp[i][j + 1]);
                dp[i][j] = Math.max(minn - dungeon[i][j], 1);
            }
        }
        return dp[0][0];
    }
}
```

#5 最长回文子串

给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。

示例 1：

输入: "babad"
输出: "bab"
注意: "aba" 也是一个有效答案。

示例 2：

输入: "cbbd"
输出: "bb"

```java
class Solution {



// public boolean isPalindromic(String s) {
//      int len = s.length();
//      for (int i = 0; i < len / 2; i++) {
//          if (s.charAt(i) != s.charAt(len - i - 1)) {
//              return false;
//          }
//      }
//      return true;
//  }

// // 暴力解法
// public String longestPalindrome(String s) {
//     String ans = "";
//     int max = 0;
//     int len = s.length();
//     for (int i = 0; i < len; i++)
//         for (int j = i + 1; j <= len; j++) {
//             String test = s.substring(i, j);
//             if (isPalindromic(test) && test.length() > max) {
//                 ans = s.substring(i, j);
//                 max = Math.max(max, ans.length());
//             }
//         }
//     return ans;
// }

public String longestPalindrome(String s) {
    if (s.equals(""))
        return "";
    String origin = s;
    String reverse = new StringBuffer(s).reverse().toString();
    int length = s.length();
    int[][] arr = new int[length][length];
    int maxLen = 0;
    int maxEnd = 0;
    for (int i = 0; i < length; i++)
        for (int j = 0; j < length; j++) {
            if (origin.charAt(i) == reverse.charAt(j)) {
                if (i == 0 || j == 0) {
                    arr[i][j] = 1;
                } else {
                    arr[i][j] = arr[i - 1][j - 1] + 1;
                }
            }
            /**********修改的地方******************/
            if (arr[i][j] > maxLen) {
                int beforeRev = length - 1 - j;
                if (beforeRev + arr[i][j] - 1 == i) { //判断下标是否对应
                    maxLen = arr[i][j];
                    maxEnd = i;
                }
                /************************************/
            }
        }
```

```
    return s.substring(maxEnd - maxLen + 1, maxEnd + 1);
}




}
```

#394 字符串解码

给定一个经过编码的字符串，返回它解码后的字符串。

编码规则为: k[encoded_string]，表示其中方括号内部的 encoded_string 正好重复 k 次。注意 k 保证为正整数。

你可以认为输入字符串总是有效的；输入字符串中没有额外的空格，且输入的方括号总是符合格式要求的。

此外，你可以认为原始数据不包含数字，所有的数字只表示重复的次数 k ，例如不会出现像 3a 或 2[4] 的输入。

示例 1：

输入：s = "3[a]2[bc]"
输出："aaabcbc"

示例 2：

输入：s = "3[a2[c]]"
输出："accaccacc"

示例 3：

输入：s = "2[abc]3[cd]ef"
输出："abcabccdcdcdef"

示例 4：

输入：s = "abc3[cd]xyz"
输出："abccdcdcdxyz"

```
class Solution {
 public String decodeString(String s) {
        return dfs(s, 0)[0];
    }
    private String[] dfs(String s, int i) {
        StringBuilder res = new StringBuilder();
        int multi = 0;
        while(i < s.length()) {
            if(s.charAt(i) >= '0' && s.charAt(i) <= '9')
                multi = multi * 10 +
Integer.parseInt(String.valueOf(s.charAt(i)));
            else if(s.charAt(i) == '[') {
                String[] tmp = dfs(s, i + 1);
                i = Integer.parseInt(tmp[0]);
                while(multi > 0) {
```

```
                    res.append(tmp[1]);
                    multi--;
                }
            }
            else if(s.charAt(i) == ']')
                return new String[] { String.valueOf(i), res.toString() };
            else
                res.append(String.valueOf(s.charAt(i)));
            i++;
        }
        return new String[] { res.toString() };
    }

}
```

#19 删除链表的倒数第N个节点

给定一个链表，删除链表的倒数第 n 个节点，并且返回链表的头结点。

示例：

给定一个链表: 1->2->3->4->5, 和 n = 2.

当删除了倒数第二个节点后，链表变为 1->2->3->5.

说明：

给定的 n 保证是有效的。

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
        public ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode first = dummy;
        ListNode second = dummy;
        // Advances first pointer so that the gap between first and second is n
nodes apart
        for (int i = 1; i <= n + 1; i++) {
            first = first.next;
        }
        // Move first to the end, maintaining the gap
        while (first != null) {
            first = first.next;
            second = second.next;
        }
        second.next = second.next.next;
        return dummy.next;
```

```
    }



    //     ListNode dummy = new ListNode(0);
    // dummy.next = head;
    // ListNode first = dummy;
    // ListNode second = dummy;
    // // Advances first pointer so that the gap between first and second is n
nodes apart
    // for (int i = 1; i <= n + 1; i++) {
    //     first = first.next;
    // }
    // // Move first to the end, maintaining the gap
    // while (first != null) {
    //     first = first.next;
    //     second = second.next;
    // }
    // second.next = second.next.next;
    // return dummy.next;

}
```

287 寻找重复数

给定一个包含 n + 1 个整数的数组 nums，其数字都在 1 到 n 之间（包括 1 和 n），可知至少存在一个重复的整数。假设只有一个重复的整数，找出这个重复的数。

示例 1:

```
输入: [1,3,4,2,2]
输出: 2
```

示例 2:

```
输入: [3,1,3,4,2]
输出: 3
```

说明：

不能更改原数组（假设数组是只读的）。
只能使用额外的 O(1) 的空间。
时间复杂度小于 O(n2) 。
数组中只有一个重复的数字，但它可能不止重复出现一次。

```
class Solution {
        public int findDuplicate(int[] nums) {
            int slow = nums[0];
            int fast = nums[nums[0]];
            //寻找相遇点
            while (slow != fast) {
                slow = nums[slow];
                fast = nums[nums[fast]];
            }
            //slow 从起点出发, fast 从相遇点出发, 一次走一步
            slow = 0;
            while (slow != fast) {
```

```
                slow = nums[slow];
                fast = nums[fast];
            }
            return slow;
        }
}
// public int findDuplicate(int[] nums) {
//     HashSet<Integer> set = new HashSet<>();
//     for (int i = 0; i < nums.length; i++) {
//         if (set.contains(nums[i])) {
//             return nums[i];
//         }
//         set.add(nums[i]);
//     }
//     return -1;
// }
```

#1195 交替打印字符串

编写一个可以从 1 到 n 输出代表这个数字的字符串的程序，但是：

    如果这个数字可以被 3 整除，输出 "fizz"。
    如果这个数字可以被 5 整除，输出 "buzz"。
    如果这个数字可以同时被 3 和 5 整除，输出 "fizzbuzz"。

例如，当 n = 15，输出： 1, 2, fizz, 4, buzz, fizz, 7, 8, fizz, buzz, 11, fizz, 13, 14, fizzbuzz。

假设有这么一个类：

```
class FizzBuzz {
  public FizzBuzz(int n) { ... }               // constructor
  public void fizz(printFizz) { ... }          // only output "fizz"
  public void buzz(printBuzz) { ... }          // only output "buzz"
  public void fizzbuzz(printFizzBuzz) { ... }  // only output "fizzbuzz"
  public void number(printNumber) { ... }      // only output the numbers
}
```

请你实现一个有四个线程的多线程版 FizzBuzz， 同一个 FizzBuzz 实例会被如下四个线程使用：

    线程A将调用 fizz() 来判断是否能被 3 整除，如果可以，则输出 fizz。
    线程B将调用 buzz() 来判断是否能被 5 整除，如果可以，则输出 buzz。
    线程C将调用 fizzbuzz() 来判断是否同时能被 3 和 5 整除，如果可以，则输出 fizzbuzz。
    线程D将调用 number() 来实现输出既不能被 3 整除也不能被 5 整除的数字。

```
class FizzBuzz {
    private int n;

    public FizzBuzz(int n) {
        this.n = n;
    }

    private Semaphore fSema = new Semaphore(0);
    private Semaphore bSema = new Semaphore(0);
    private Semaphore fbSema = new Semaphore(0);
    private Semaphore nSema = new Semaphore(1);
```

```
    // private ReentrantLock lock = new ReentrantLock();
    // private Condition fCond = lock.newCondition();
    // private Condition bCond = lock.newCondition();
    // private Condition fbCond = lock.newCondition();
    // private Condition nCond = lock.newCondition();
    // private volatile Boolean state = false;

    /**
    * use semaphore
    * @param printFizz
    * @throws InterruptedException
    */
    // printFizz.run() outputs "fizz".
    public void fizz(Runnable printFizz) throws InterruptedException {
        for (int i = 3; i <= n; i = i + 3) {
            if (i % 5 != 0) {
                fSema.acquire();
                printFizz.run();
                nSema.release();
            }
        }
    }

    // printBuzz.run() outputs "buzz".
    public void buzz(Runnable printBuzz) throws InterruptedException {
        for (int i = 5; i <= n; i = i + 5) {
            if (i % 3 != 0) {
                bSema.acquire();
                printBuzz.run();
                nSema.release();
            }
        }
    }

    // printFizzBuzz.run() outputs "fizzbuzz".
    public void fizzbuzz(Runnable printFizzBuzz) throws InterruptedException {
        for (int i = 15; i <= n; i = i + 15) {
            fbSema.acquire();
            printFizzBuzz.run();
            nSema.release();
        }
    }

    // printNumber.accept(x) outputs "x", where x is an integer.
    public void number(IntConsumer printNumber) throws InterruptedException {
        for (int i = 1; i <= n; i++) {
            nSema.acquire();
            if (i % 3 == 0 && i % 5 == 0) {
                fbSema.release();
            }else if (i % 3 == 0) {
                fSema.release();
            }else if (i % 5 == 0) {
                bSema.release();
            }else {
                printNumber.accept(i);
                nSema.release();
            }
        }
```

```
    }


    /**
    * use lock
    * @param printFizz
    * @throws InterruptedException
    */
    // printFizz.run() outputs "fizz".
    // public void fizz1(Runnable printFizz) throws InterruptedException {
    //     for (int i = 3; i <= n; i = i + 3) {
    //         lock.lock();
    //         try {
    //             if (i % 5 != 0) {
    //                 while (!state) {
    //                     fCond.await();
    //                 }
    //                 printFizz.run();
    //                 state = false;
    //                 nCond.signal();
    //             }
    //         }finally {
    //             lock.unlock();
    //         }
    //     }
    // }

    // // printBuzz.run() outputs "buzz".
    // public void buzz1(Runnable printBuzz) throws InterruptedException {
    //     for (int i = 5; i <= n; i = i + 5) {
    //         lock.lock();
    //         try {
    //             if (i % 3 != 0) {
    //                 while (!state) {
    //                     bCond.await();
    //                 }
    //                 printBuzz.run();
    //                 state = false;
    //                 nCond.signal();
    //             }
    //         }finally {
    //             lock.unlock();
    //         }
    //     }
    // }

    // // printFizzBuzz.run() outputs "fizzbuzz".
    // public void fizzbuzz1(Runnable printFizzBuzz) throws InterruptedException
{
    //     for (int i = 15; i <= n; i = i + 15) {
    //         lock.lock();
    //         try {
    //             while (!state) {
    //                 fbCond.await();
    //             }
    //             printFizzBuzz.run();
    //             state = false;
    //             nCond.signal();
```

```
    //         }finally {
    //             lock.unlock();
    //         }

    //     }
    // }

    // // printNumber.accept(x) outputs "x", where x is an integer.
    // public void number1(IntConsumer printNumber) throws InterruptedException
{
    //     for (int i = 1; i <= n; i++) {
    //         lock.lock();
    //         try {
    //             while (state) {
    //                 nCond.await();
    //             }
    //             if (i % 3 == 0 && i % 5 == 0) {
    //                 fbCond.signal();
    //                 state = true;
    //             }else if (i % 3 == 0) {
    //                 fCond.signal();
    //                 state = true;
    //             }else if (i % 5 == 0) {
    //                 bCond.signal();
    //                 state = true;
    //             }else {
    //                 printNumber.accept(i);
    //                 nCond.signal();
    //             }
    //         }finally {
    //             lock.unlock();
    //         }
    //     }
    // }

    // public static void main(String[] args) {
    //     PrintFizzBuzz pfb = new PrintFizzBuzz(15);
    //     Thread t1 = new Thread(() -> {
    //         try {
    //             pfb.fizz1(() -> System.out.print("fizz"));
    //         } catch (InterruptedException e) {
    //             e.printStackTrace();
    //         }
    //     });

    //     Thread t2 = new Thread(() -> {
    //         try {
    //             pfb.buzz1(() -> System.out.print("buzz"));
    //         } catch (InterruptedException e) {
    //             e.printStackTrace();
    //         }
    //     });

    //     Thread t3 = new Thread(() -> {
    //         try {
    //             pfb.fizzbuzz1(() -> System.out.print("fizzbuzz"));
    //         } catch (InterruptedException e) {
    //             e.printStackTrace();
```

```
    //         }
    //     });

    //     Thread t4 = new Thread(() -> {
    //         try {
    //             pfb.number1(value -> System.out.print(value));
    //         } catch (InterruptedException e) {
    //             e.printStackTrace();
    //         }
    //     });

    //     t1.start();
    //     t2.start();
    //     t3.start();
    //     t4.start();

    // }



}
```

#76 最小覆盖子串

给你一个字符串 `S`、一个字符串 `T` 。请你设计一种算法，可以在 O(n) 的时间复杂度内，从字符串 `S` 里面找出：包含 `T` 所有字符的最小子串。

示例：

输入：S = "ADOBECODEBANC", T = "ABC"
输出："BANC"

提示：

- 如果 `S` 中不存这样的子串，则返回空字符串 `""`。
- 如果 `S` 中存在这样的子串，我们保证它是唯一的答案。

```
class Solution {
    Map<Character, Integer> ori = new HashMap<Character, Integer>();
    Map<Character, Integer> cnt = new HashMap<Character, Integer>();

    public String minWindow(String s, String t) {
        int tLen = t.length();
        for (int i = 0; i < tLen; i++) {
            char c = t.charAt(i);
            ori.put(c, ori.getOrDefault(c, 0) + 1);
        }
        int l = 0, r = -1;
        int len = Integer.MAX_VALUE, ansL = -1, ansR = -1;
        int sLen = s.length();
        while (r < sLen) {
            ++r;
            if (r < sLen && ori.containsKey(s.charAt(r))) {
```

```
                cnt.put(s.charAt(r), cnt.getOrDefault(s.charAt(r), 0) + 1);
            }
            while (check() && l <= r) {
                if (r - l + 1 < len) {
                    len = r - l + 1;
                    ansL = l;
                    ansR = l + len;
                }
                if (ori.containsKey(s.charAt(l))) {
                    cnt.put(s.charAt(l), cnt.getOrDefault(s.charAt(l), 0) - 1);
                }
                ++l;
            }
        }
        return ansL == -1 ? "" : s.substring(ansL, ansR);
    }

    public boolean check() {
        Iterator iter = ori.entrySet().iterator();
        while (iter.hasNext()) {
            Map.Entry entry = (Map.Entry) iter.next();
            Character key = (Character) entry.getKey();
            Integer val = (Integer) entry.getValue();
            if (cnt.getOrDefault(key, 0) < val) {
                return false;
            }
        }
        return true;
    }
}
```

#262 行程和用户

Trips 表中存所有出租车的行程信息。每段行程有唯一键 Id，Client_Id 和 Driver_Id 是 Users 表中 Users_Id 的外键。Status 是枚举类型，枚举成员为 (‘completed’, ‘cancelled_by_driver’, ‘cancelled_by_client’)。

Id	Client_Id	Driver_Id	City_Id	Status	Request_at
1	1	10	1	completed	2013-10-01
2	2	11	1	cancelled_by_driver	2013-10-01
3	3	12	6	completed	2013-10-01
4	4	13	6	cancelled_by_client	2013-10-01
5	1	10	1	completed	2013-10-02
6	2	11	6	completed	2013-10-02
7	3	12	6	completed	2013-10-02
8	2	12	12	completed	2013-10-03
9	3	10	12	completed	2013-10-03
10	4	13	12	cancelled_by_driver	2013-10-03

Users 表存所有用户。每个用户有唯一键 Users_Id。Banned 表示这个用户是否被禁止，Role 则是一个表示（‘client’, ‘driver’, ‘partner’）的枚举类型。

Users_Id	Banned	Role

```
+----------+--------+--------+
|    1     |   No   | client |
|    2     |   Yes  | client |
|    3     |   No   | client |
|    4     |   No   | client |
|    10    |   No   | driver |
|    11    |   No   | driver |
|    12    |   No   | driver |
|    13    |   No   | driver |
+----------+--------+--------+
```

写一段 SQL 语句查出 2013年10月1日 至 2013年10月3日 期间非禁止用户的取消率。基于上表，你的 SQL 语句应返回如下结果，取消率（Cancellation Rate）保留两位小数。

取消率的计算方式如下：(被司机或乘客取消的非禁止用户生成的订单数量) / (非禁止用户生成的订单总数)

```
+------------+-------------------+
|     Day    | Cancellation Rate |
+------------+-------------------+
| 2013-10-01 |       0.33        |
| 2013-10-02 |       0.00        |
| 2013-10-03 |       0.50        |
+------------+-------------------+
```

```
# Write your MySQL query statement below

SELECT
t.request_at as 'day',
ROUND(
sum(if(t.STATUS = 'completed',0,1)) / count(t.STATUS),2
) as 'Cancellation Rate'
from trips t join users u1 on t.client_id = u1.users_id
join users u2 on t.driver_id = u2.users_id
where u1.banned = 'no' and u2.banned = 'no'
and t.request_at BETWEEN '2013-10-01' AND '2013-10-03'
GROUP by t.request_at
```

```
-- SELECT T.request_at AS `Day`,
--  ROUND(
--          SUM(
--              IF(T.STATUS = 'completed',0,1)
--          )
--          /
--          COUNT(T.STATUS),
--          2
--  ) AS `Cancellation Rate`
-- FROM Trips AS T
-- JOIN Users AS U1 ON (T.client_id = U1.users_id AND U1.banned ='No')
-- JOIN Users AS U2 ON (T.driver_id = U2.users_id AND U2.banned ='No')
-- WHERE T.request_at BETWEEN '2013-10-01' AND '2013-10-03'
```

```
-- GROUP BY T.request_at


/* Write your PL/SQL query statement below */

SELECT
t.request_at as 'day',
ROUND(
sum(decode(t.STATUS,'completed',0,1)) / count(t.STATUS),2
) as 'Cancellation Rate'
from trips t join users u1 on t.client_id = u1.users_id
join users u2 on t.driver_id = u2.users_id
where u1.banned = 'no' and u2.banned = 'no'
and t.request_at BETWEEN '2013-10-01' AND '2013-10-03'
GROUP by t.request_at
```

#974 和可被 K 整除的子数组

给定一个整数数组 A，返回其中元素之和可被 K 整除的（连续、非空）子数组的数目。

示例：

输入：A = [4,5,0,-2,-3,1], K = 5
输出：7
解释：
有 7 个子数组满足其元素之和可被 K = 5 整除：
[4, 5, 0, -2, -3, 1], [5], [5, 0], [5, 0, -2, -3], [0], [0, -2, -3], [-2, -3]

提示：

```
    1 <= A.length <= 30000
    -10000 <= A[i] <= 10000
    2 <= K <= 10000
```

来源：力扣（LeetCode）
链接：https://leetcode-cn.com/problems/subarray-sums-divisible-by-k
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

```
class Solution {
        public int subarraysDivByK(int[] A, int K) {
        Map<Integer, Integer> record = new HashMap<>();
        record.put(0, 1);
        int sum = 0, ans = 0;
        for (int elem: A) {
            sum += elem;
            // 注意 Java 取模的特殊性，当被除数为负数时取模结果为负数，需要纠正
            int modulus = (sum % K + K) % K;
            int same = record.getOrDefault(modulus, 0);
            ans += same;
            record.put(modulus, same + 1);
        }
        return ans;
    }
```

```
}
```

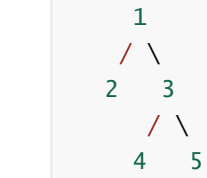
#297 二叉树的序列化与反序列化

序列化是将一个数据结构或者对象转换为连续的比特位的操作，进而可以将转换后的数据存储在一个文件或者内存中，同时也可以通过网络传输到另一个计算机环境，采取相反方式重构得到原数据。

请设计一个算法来实现二叉树的序列化与反序列化。这里不限定你的序列 / 反序列化算法执行逻辑，你只需要保证一个二叉树可以被序列化为一个字符串并且将这个字符串反序列化为原始的树结构。

示例：

你可以将以下二叉树：



序列化为 `"[1,2,3,null,null,4,5]"`

提示：这与 LeetCode 目前使用的方式一致，详情请参阅 LeetCode 序列化二叉树的格式。你并非必须采取这种方式，你也可以采用其他的方法解决这个问题。

说明：不要使用类的成员 / 全局 / 静态变量来存储状态，你的序列化和反序列化算法应该是无状态的。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Codec {
    public String rserialize(TreeNode root, String str) {
        if (root == null) {
            str += "None,";
        } else {
            str += str.valueOf(root.val) + ",";
            str = rserialize(root.left, str);
            str = rserialize(root.right, str);
        }
        return str;
    }

    public String serialize(TreeNode root) {
        return rserialize(root, "");
    }

    public TreeNode rdeserialize(List<String> l) {
        if (l.get(0).equals("None")) {
            l.remove(0);
            return null;
```

```
        }

        TreeNode root = new TreeNode(Integer.valueOf(l.get(0)));
        l.remove(0);
        root.left = rdeserialize(l);
        root.right = rdeserialize(l);

        return root;
    }

    public TreeNode deserialize(String data) {
        String[] data_array = data.split(",");
        List<String> data_list = new LinkedList<String>
(Arrays.asList(data_array));
        return rdeserialize(data_list);
    }
}



// Your Codec object will be instantiated and called as such:
// Codec codec = new Codec();
// codec.deserialize(codec.serialize(root));
```

#180 连续出现的数字

编写一个 SQL 查询，查找所有至少连续出现三次的数字。

```
+----+-----+
| Id | Num |
+----+-----+
| 1  |  1  |
| 2  |  1  |
| 3  |  1  |
| 4  |  2  |
| 5  |  1  |
| 6  |  2  |
| 7  |  2  |
+----+-----+
```

例如，给定上面的 Logs 表， 1 是唯一连续出现至少三次的数字。

```
+-----------------+
| ConsecutiveNums |
+-----------------+
| 1               |
+-----------------+
```

```
# Write your MySQL query statement below
SELECT DISTINCT
    l1.Num AS ConsecutiveNums
FROM
    Logs l1,
    Logs l2,
    Logs l3
WHERE
    l1.Id = l2.Id - 1
```

```
    AND l2.Id = l3.Id - 1
    AND l1.Num = l2.Num
    AND l2.Num = l3.Num
;
```

739 每日温度

请根据每日 气温 列表，重新生成一个列表。对应位置的输出为：要想观测到更高的气温，至少需要等待的天数。如果气温在这之后都不会升高，请在该位置用 0 来代替。

例如，给定一个列表 temperatures = [73, 74, 75, 71, 69, 72, 76, 73]，你的输出应该是 [1, 1, 4, 2, 1, 1, 0, 0]。

提示：气温 列表长度的范围是 [1, 30000]。每个气温的值的均为华氏度，都是在 [30, 100] 范围内的整数。
通过次数100,003
提交次数154,843

```
class Solution {
    public int[] dailyTemperatures(int[] T) {
        int length = T.length;
        int[] ans = new int[length];
        Deque<Integer> stack = new LinkedList<Integer>();
        for (int i = 0; i < length; i++) {
            int temperature = T[i];
            while (!stack.isEmpty() && temperature > T[stack.peek()]) {
                int prevIndex = stack.pop();
                ans[prevIndex] = i - prevIndex;
            }
            stack.push(i);
        }
        return ans;
    }

}
```

#105 从前序与中序遍历序列构造二叉树

根据一棵树的前序遍历与中序遍历构造二叉树。

注意:
你可以假设树中没有重复的元素。

例如，给出

前序遍历 preorder = [3,9,20,15,7]
中序遍历 inorder = [9,3,15,20,7]

返回如下的二叉树：

```
    3
   / \
  9  20
    /  \
   15   7
```

```
/**
```

```
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
public TreeNode buildTree(int[] preorder, int[] inorder) {
    return buildTreeHelper(preorder,  inorder, (long)Integer.MAX_VALUE + 1);
}
int pre = 0;
int in = 0;
private TreeNode buildTreeHelper(int[] preorder, int[] inorder, long stop) {
    //到达末尾返回 null
    if(pre == preorder.length){
        return null;
    }
    //到达停止点返回 null
    //当前停止点已经用了，in 后移
    if (inorder[in] == stop) {
        in++;
        return null;
    }
    int root_val = preorder[pre++];
    TreeNode root = new TreeNode(root_val);
    //左子树的停止点是当前的根节点
    root.left = buildTreeHelper(preorder,  inorder, root_val);
    //右子树的停止点是当前树的停止点
    root.right = buildTreeHelper(preorder, inorder, stop);
    return root;
}

}
```

#207 课程表

你这个学期必须选修 numCourse 门课程，记为 0 到 numCourse-1 。

在选修某些课程之前需要一些先修课程。 例如，想要学习课程 0 ，你需要先完成课程 1 ，我们用一个匹配来表示他们：[0,1]

给定课程总量以及它们的先决条件，请你判断是否可能完成所有课程的学习？

示例 1:

输入: 2, [[1,0]]
输出: true
解释: 总共有 2 门课程。学习课程 1 之前，你需要完成课程 0。所以这是可能的。

示例 2:

输入: 2, [[1,0],[0,1]]
输出: false

解释：总共有 2 门课程。学习课程 1 之前，你需要先完成课程 0；并且学习课程 0 之前，你还应先完成课程 1。这是不可能的。

提示：

- 输入的先决条件是由 边缘列表 表示的图形，而不是 邻接矩阵 。详情请参见图的表示法。
- 你可以假定输入的先决条件中没有重复的边。
- 1 <= numCourses <= 10^5

```
class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        int[] indegrees = new int[numCourses];
        List<List<Integer>> adjacency = new ArrayList<>();
        Queue<Integer> queue = new LinkedList<>();
        for(int i = 0; i < numCourses; i++)
            adjacency.add(new ArrayList<>());
        // Get the indegree and adjacency of every course.
        for(int[] cp : prerequisites) {
            indegrees[cp[0]]++;
            adjacency.get(cp[1]).add(cp[0]);
        }
        // Get all the courses with the indegree of 0.
        for(int i = 0; i < numCourses; i++)
            if(indegrees[i] == 0) queue.add(i);
        // BFS TopSort.
        while(!queue.isEmpty()) {
            int pre = queue.poll();
            numCourses--;
            for(int cur : adjacency.get(pre))
                if(--indegrees[cur] == 0) queue.add(cur);
        }
        return numCourses == 0;
    }

}
```