OO第三单元JML总结----------程序性能的思考

一、概述

第三单元针对JML进行规格设计，非常贴合根据用户需求实现设计这一过程。整体上来看，个人认为所谓“第三单元远没有一二单元复杂”只不过是一面虚词。从思考难度上来说，第一单元对于嵌套和WF的处理实在令人朝思夕计；从理解上，第二单元的实现有点难以下手；第三单元在这两个方面都降低了难度，似乎只要根据JML写就可以了，但是这也是第三单元的~~最折腾人的地方~~——它对于程序的性能提出了更为苛刻的要求。

本次博客将从三个方面介绍和分析这一单元：架构设计、程序性能以及最后谈及的JML测试技术性方案。架构设计**以三次作业作为核心步步分析个人在实现JML规格时所采用的的架构和方法**；程序性能主要谈及了在**面对上万条数据中所进行的性能优化和考虑，包括容器使用、算法优化、bug分析三个方面**；最后JML测试方案谈及了我对于JML测试一点点薄见。

虽然说本次博客是技术性博客，但是对于我这种对于上万条数据无从下手的同学而言，也很难有有用的技术方案，也愿各位读者不吝赐教！

二、架构设计和规格实现

第一次作业

依据之前多次作业的分析方式，我依然会从UML类图入手给出程序大体框架，不过类和方法复杂度就不专门列出表格分析了。

第一次作业的主要内容是熟悉JML语言，通过其实现一个社交网络`NetWork`，和成员类`Person`，其中能够满足人员查询和社交圈环（图）的分析。

- UML类图如下：
- `MyPerson`类属性分析
  - 包含5个数据类型和9个方法，其中7个来源于继承，根据JML定义的规格书写即可。
  - 其余两个方法用于获取数据类型`acquaintance`和`value`
  - 比较简单
- `MyNetWork`类属性分析
  - 包括3个数据类型和多个方法

- 3个数据类型一个用于JML规格定义的`personArrayList`，另外两个分别为`boolean`和`ArrayList`型变量，用于`isCircle`的判断。
- 多个方法中直接按照JML定义书写，其中`isCircle`采用了DFS方式，并添加了私有方法`circuit`进行递归。
- 没有故意卡超时，否则程序直接完蛋。

```
//一个比较糟糕的代码书写，也是初版对于DFS的设计
private void circuit(MyPerson p1, MyPerson p2) {
    //用于标记
    if (!flags.contains(p1.getId())) {
        flags.add(p1.getId());
    } else {
        return;
    }
    //用于判断
    if (p1.isLinked(p2) || ans) {
        ans = true;
        return;
    }
    //用于DFS递归
    ArrayList<Person> people = p1.getAcquaintance();
    for (Person p : people) {
        if (p instanceof MyPerson) {
            circuit((MyPerson) p, p2);
        }
    }
}
```

- 整体而言结构非常简单，这种简单的结构极易造成一种错觉，从而在二、三次作业中体现出来。

第二次作业

第二次作业修改很多，同时从第二次作业开始，关于性能方面的要求就体现出来了，仅仅根据JML中规定的方法是不够的。

- UML类图如下：
- 用于实例化接口的类的分析
  - `MyPerson`
    - 针对于数据类型，除了JML定义中的类型，另外设置了`private int fatherId`用于确定其对应的父节点，初始设置为自身`id`。
    - 针对于方法，除继承方法外，另外添加了对于`fatherId`的设置和`isSameUnit`的父节点判断。

        - 新增的部分主要用于`UnitSet`中找到对应的`circle`。
    - `MyGroup`
        - 针对数据类型，主要添加了很多保存当前某个值的变量，以及规定的`Person[]`数组。
        - 针对方法，按照要求的规格进行设计即可，没有添加私有方法进行实现
    - `MyMessage`
        - 这是整个单元中除`main`以外最简单的类，因此直接按照JML定义的数据类型和方法定义就好。
    - `MyNetWork`
        - 相对于第一单元，增加了对于`MyGroup`和对`MyMessage`的构建和使用。
        - 针对于数据类型，增加了自身定义类`UnitSet`和`preStatus`的定义，其余按照JML定义即可
        - 针对于方法，除需要按照JML定义之外，仍需要考虑时间复杂度，主要体现在`queryBlockSum`和对于`MyGroup`的一些分析中。
- 用于临时存储状态的类分析（本次设计重点）
    - `PreStatus`

```java
public class PreStatus {
    //dirty位，可以视为类似于操作系统中的脏位，用于标记状态是否改变
    private Map<Integer, Boolean> dirty = new HashMap<>();
    //data表示之前一个状态中保存的各个结果变量
    private Map<Integer, Object> data = new HashMap<>();
    //buffer可以视为缓存，暂时保存两个状态之间的新增数据，下一个状态到来时清空
    private Map<Integer, Map> buffer = new HashMap<>();
    //map是NetWork类中的映射关系
    private Map<Integer, ArrayList<Integer>> map = new HashMap<>();

    //初始化
    public void init(Map<Integer, ArrayList<Integer>> map) {...}

    //与data相关
    public Object getElement(int wayId) {return data.get(wayId);}

```

```
    public void setElement(int wayId, Object o) {...}

    //与dirty相关
    public boolean getDirty(int wayId) {return dirty.get(wayId);}

    //与buffer相关
    public void setBuffer(int changeId, Map map) {...}

    ...
}

```

对于类中重点内容的定义如注释所示。这里具体分析思考过程。

对于第二次作业中不断出现的TLE，尤其有上万条测试数据让我第一次觉得**程序改不了了**。做第二次作业我很欣然地按照规格定义写然后提交，却在强测中直接暴毙，我确确实实**需要面对性能的问题**。研讨课中有一位同学提及了可以考虑设置脏位等，当时产生了一点对于怎么修改产生了一点灵感——也就是我**需要想办法存储一些数据，而且能够侦查数据改变前后之间的一个过程——这也是为何需要设置`buffer`缓存的原因**。读者实际上可以注意到比较特殊的一个地方：

```
private Map<Integer, Map> buffer = new HashMap<>();
```

我所定义的`buffer`是一个`Map`作为存储数据的单元，因此其存储的数据可以是`Map<Integer, Person>`（对应`addPerson`），也可以是`Map<Integer, Map<Integer, Integer>>`(对应`addRelation`)，使用时可能需要先利用`instanceOf`判断是哪一种类型，继而进行对应哪一种操作。

另外，**需要考虑怎样在外界中使用这个类。**因此，我在`MyNetWork`中构造了两个通用的方法来进行使用。

```java
//更新，将涉及的改变变量加入缓存（buffer）
public void update(int changeId, Map map) {
    preStatus.setDirty(changeId);
    preStatus.setBuffer(changeId, map);
}

//存储数据
public void store(int wayId, Object map) {
    preStatus.recoverDirty(wayId);
    preStatus.recoverBuffer(wayId);
    preStatus.setElement(wayId, map);
}
```

最后，我仍需要考虑一点：也就最开始谈及的`init()`方法，以及所谓的数据类型`Map<Integer, ArrayList<Integer>> map`的作用。

在`MyNetWork`中我另外定义了`enum`类型变量，用于满足添加、删除操作会影响对应查询等操作的对应关系。比如，`addPerson`会直接影响`queryNameRank`，而`addPerson`和`addRelation`均会影响`queryBlockSum`，这种关系可以是一对多，也可以是多对一等等，建立好对应的关系有利于代码的理解和书写。

```java
private enum Ways { queryNameRank,  queryBlockSum }

private enum Change { addPerson, addRelation }
```

因此`init()`在构造`MyNetWork`中使用：

```java
//MyNetwork.java
public MyNetwork() {
    Map<Integer, ArrayList<Integer>> map = new HashMap<>
();
    ArrayList<Integer> array1 = new ArrayList<>();
    array1.add(Ways.queryNameRank.ordinal());
    array1.add(Ways.queryBlockSum.ordinal());
    map.put(Change.addPerson.ordinal(), array1);
    ArrayList<Integer> array2 = new ArrayList<>();
    array2.add(Ways.queryBlockSum.ordinal());
    map.put(Change.addRelation.ordinal(), array2);
    preStatus.init(map);
}

//preStatus.java
public void init(Map map) {
    this.map = map;
```

```
        //对于每个map对应的操作集ArrayList.get(i)
        dirty.put(i, false);
        data.put(i, null);
        buffer.put(i, new HashMap());
    }
```

因而会使得其满足如下关系：

- **每一个定义的`Ways`有着自己的`dirty`、`data`、`buffer`。**
- **`Change`可以一次性修改所对应的所有`Ways`的`dirty`、`buffer`。**
- **`Ways`调用时根据`dirty`状态进行运行，如果`dirty`为`false`，直接返回`data`中数据；否则先获得`buffer`，然后根据`buffer`进一步计算，得到结果后修改`dirty`为`false`，更新`data`，清空`buffer`。**

这样做有两个优点：

- 解决了多次执行重复指令的问题
- 方法执行不需要从头计算到最后，而是直接以上一状态作为开始继续计算，大大加速了程序运行。

最后由于篇幅因素，我就不具体介绍类之间具体的交互了，包括如何进行`buffer`中`Map`类型判断、怎样调用`store()`和`update()`等等。毕竟我的思路实现过程很复杂，实际上其他同学会有更好的方法。这里通过一个流程图片进行展示。

- 用于算法优化的类分析

  谈及算法优化，实际上第二次作业主要还是DFS算法太慢了，因此利用并查集进行了修改，增加`UnitSet`，并在`MyPerson`中添加了`fatherId`数据变量。

  - `UnitSet`

```
public class UnitSet {
    //用于存储每一个block
    private Map<Integer, Map<Integer, Person>> unit = new HashMap<>();
    //适用于下一次调用到来前临时缓存新增加的人
    private Map<Integer, Person> tmpArray = new HashMap<>();
    //deleteId只适用于内部，用于自身在判断两个人存在关联时，将两个人从tmpArray中删除
    private Map<Integer, Person> deleteId = new HashMap<>();
    private int sizeOfUnit = 0;

    //这一个方法用于判断两个人存在关系时，可能用到merge合并block
```

```
    public void countUnit(MyPerson d1, MyPerson d2) {}

    //这一个方法用于统计落单的人，并返回最后的结果
    public int countUnit() {}

    private void merge(int id1, int id2) {}

    public void addPerson(Person p) {}
    public void removePerson(Person p) {}
}
```

对于类自身内容的定义如代码注释。这里先利用伪代码简单分析一下实现过程。

```
public void countUnit(MyPerson d1, MyPerson d2) {
    /*用于addRelation操作*/
    int id1 = d1.getFatherId();
    int id2 = d2.getFatherId();
    if (unit.containsKey(id1) ||
unit.containsKey(id2)) {
        //如果id1和id2同时存在，则合并，两者fatherId更新为其中较小的那个
        //利用函数merge()
        //如果只存在其中一个，则将另一个加入，并更新另一个的fatherId
    }
    else {
        //如果均不存在，则构建1个新的，添加到unit中，设置键值为fatherId较小值
        sizeOfUnit++;
    }

    /*用于addPerson操作*/
    //将已经经过addRelaion单元的变量编入删除序列
    removePerson(d1);
    removePerson(d2);
}

public int countUnit() {
for (Integer id : deleteId.keySet()) {
        /*首先去除已经进行过addRelation操作后的人*/
    }
    deleteId.clear();
    for (Integer id : tmpArray.keySet()) {
```

```
27              //剩下的都是通过addPerson而并未通过addRelation的落单单元
28              //直接新建unit的元素
29          }
30          tmpArray.clear();
31          return sizeOfUnit;
32      }
```

可以看出这种方式时间复杂度一般是远小于O(n)的，非常快速。每次调用`queryBlockSum`时只需要对于`buffer`中保存的`addPerson`和`addRelation`单元进行两次`countUnit`即可。实际上是更好的并查集模式。

第三次作业

第三次作业在第二次作业已经进行很好修改后实际上需要增加的内容不多，实际架构不需要变化，除了新增定义，另外新增了一个用于最短路径的迪杰斯特拉算法类。

- UML类图

   MyPerson

- 新增实例化接口分析
  - `MyEmoijiMessage`等`Messge`的的继承类：直接调用`super`
  - `MyNetWork`
    - 针对于数据类型，由于`Message`的内容发生改变，于是尝试使用`emojiFinalMessages`、`envelopMessages`、`noticeMessages`、`normalMessages`保存各种类型的`Message`。同时构建不同`Message`的`id`与所属类型映射`mapMessage`，以及专门用于`deleteColdEmoji`的消息和表情的反映射关系。均采用`Map`
    - 针对于方法：除`sendIndirectMessage`需要考虑算法以外，其余可以根据ML进行编写。其中为了简化代码结构，从而构造私有类简化。
  - `Dijskra`
    - 针对于数据类型：
      - `personArrayList`需要保存所有的节点
      - `distance`需要保存从初始到某一位置的权值（可以是临时）
      - `bestDistance`保存某点已经确定最小的权值
      - `flag`用于判断某点是否已经确定

- 针对于方法：**由于Dijskra涉及两步操作，1、每次从未标记的节点中选择最近的节点，标记并收录到最小权值序列中。2、集散刚刚进入最小权值序列节点的附近节点的距离，如果该距离小于该节点原先的距离，就将其进行更新。因此采用多个类分解这种操作。**
  - `putBest()`用于第一个操作，找到可以收录的节点
  - `seek()`和`getValue()`用于第二个操作。
  - `run()`是整个操作的调用者

因此根据算法思想构造相应类。

通过以上阐述，相信**程序构建的架构和实现思想**应该都非常清楚了。

三、性能以及bug

容器选择

- `ArrayList`**与**`Map`

这一单元似乎体现出了`ArrayList`相对于`Map`在查询方面有多不足。不过客观来说，两者各有优劣。

- `ArrayList`（与`LinkedList`）
  - 线程不安全；可以存储null
  - 通过以数组方式存储元素，元素包含先后顺序、可以重复
  - `ArrayList`相对于`LinkedList`条目更小，开销更低；但是`List`构成的集合本身开销均较低
  - 在查询、删除操作中非常不利。查询需要遍历，删除需要将整个数据向前移动。
- `Map`
  - 线程不安全；可以存储null
  - 通过以键值的方式存储元素，键值不可以重复，否则会覆盖。
  - 存储开销较大
  - 查询直接通过键值查询，删除通过键值删除，非常迅速。
  - 存在`LinkedHashMap`等多种不同的类型，可以更便于使用

而本次作业中有三大特点：**数据量庞大、查询操作非常多、存在`id`作为唯一识别单位（适合作为键值）**。因此，显而易见，本次作业大量采用`HashMap`，少数可以预知的、存储数据比较少的使用`ArrayList`。

事实上，能够将`Map`和`List`结合使用，可能会使得程序的性能更加优良。

算法思考

- **第二次作业前后算法**
  - 不愿提起的DFS算法：

    根据每个人的`acquaintance()`直接进行遍历。

```
private void circuit(MyPerson p1, MyPerson p2) {
    //进行标记
    if (!flags.contains(p1.getId())) {
        flags.add(p1.getId());
    } else {
        return;
    }
    //进行判断
    if (p1.isLinked(p2) || ans) {
        ans = true;
        return;
    }
    //取出其路径中的每个节点，深层递归
    ArrayList<Person> people = p1.getAcquaintance();
    for (Person p : people) {
        if (p instanceof MyPerson) {
            circuit((MyPerson) p, p2);
        }
    }
}
```

    时间复杂度最大能够达到O(n^2)，一般不小于$O(n)$。对于`ArrayList`的$O(n)$也无法接受的要求，显然是不行的。
  - 修正的并查集算法：详情可见架构设计中`Unitset`部分，该部分就是专门用于处理`queryBlockSum`的并查集算法。

    修正后的时间复杂度是$O(n)$。实际上由于优化了数据存储，设立了缓存区，从而实际上算法时间复杂度会远远小于$O(n)$，从而对于考察`queryBlockSum`和`isCircle`的测试点均能通过。
- **第三次作业算法**
  - 由于这个bug确确实实还没有想到更好的方式，因此就按照迪杰斯特拉算法的思路进行分析。
  - 迪杰斯特拉算法分成两步，在第三次作业分析中已经进行过分析，因此这个地方不再赘述。
  - 如果想到更好的方法，再进行更新。

bug分析

- **性能bug**

  这个地方的性能bug就是指TLE。针对TLE，主要从算法和设计角度进行了优化。实际上在描述第二次作业过程中，笔者已经将自己的修改思路进行了比较详细的分析。因此这里通过`queryBlockSum`谈谈具体实现问题以及设计中的不足之处。

  - 一般而言根据JML定义`queryBlockSum`会如此书写：

```
public int queryBlockSum() {
    int ans = 1;
    int flag = 0;
    //非常标准的“根据JML定义撰写”
    for (int i = 0; i < personArrayList.size(), i++) {
        flag = 0;
        Person p1 = personArrayList.get(i);
        for (int j = 0; j < i; j++ {
            Person p2 = personArrayList.get(j);
            if (isCircle(p1, p2)) {
                flag = 1;
                break;
            }
        }
        if (flag == 0) {
            ans++;
        }
    }
    return ans;
}
```

  但是实际上实现过程中需要考虑数据存储，实际上笔者实现非常复杂。

```
public int queryBlockSum() {
    //wayId指调用方法
    int wayId = Ways.queryBlockSum.ordinal();
    //changeId表示能够影响该调用的增加方法
    int changeId1 = Change.addPerson.ordinal();
    int changeId2 = Change.addRelation.ordinal();
    /*第一部分需要先判断调用方法时是否addPerson或者
addRelation（状态是否改变）*/
    if (preStatus.getElement(wayId) != null) {
        Integer data = (Integer)
preStatus.getElement(wayId);
        if (!preStatus.getDirty(changeId1) &&
            !preStatus.getDirty(changeId2)) {
```

```
            return data;
        }
    }
    if (preStatus.getBuffer(wayId) == null) {
        return 0;
    }

    /*第二部分如果状态发生改变，则需要获取上一次调用到这一次调用中的缓存，即buffer*/
    Map buffer = preStatus.getBuffer(wayId);
    for (Object i : buffer.keySet()) {
        /*如果该Map是Person作为Value，证明添加操作是addPerson*/
        if (buffer.get(i) instanceof Person) {
            unitSet.addPerson((Person) buffer.get(i));
        }
        /*如果该Map是一个ArrayList，证明添加操作是addRelation*/
        else if (buffer.get(i) instanceof ArrayList) {
            ArrayList<Integer> relation = (ArrayList<Integer>) buffer.get(i);
            for (int s = 1; s < relation.size(); s += 2) {
                MyPerson p1 = (MyPerson) getPerson(relation.get(s - 1));
                MyPerson p2 = (MyPerson) getPerson(relation.get(s));
                unitSet.countUnit(p1, p2);
            }
        }
    }
    //根据不同类型的判断采用unitSet中不同的判定方法
    Integer data = unitSet.countUnit();
    //更新存储数据
    store(wayId, data);
    return data;
    }
```

从整个架构可以看出，改进的方法时间复杂度最多不超过$O(n)$。而且采用了所谓缓存结构，一般会远远小于$O(n)$，大大提升了性能。

- 另外我也谈及这种方法存在一个很大的缺陷，虽然设计出这种方法花费了很长时间，但是需要承认的是，即便是我自己使用时，我都希望能够尽量避免使用自己缓存的设计方式。

    - 整个过程实际上是难以理解的。比如在`addPerson`操作中需要更新的数据表示为`Map<Integer, Person>`，而笔者也说过每一个`WayId`都有自己的`buffer`，因此需要将更新的数据表示，通过`PreStatus`中的`map`找到其具体会影响哪些`WayId`，然后再将这个数据存储到`WayId`中的`buffer`；又比如，之前谈及多个`changeId`会影响一个`WayId`，而每个`changeId`存储数据的容器类型表示也不能相同（否则无法区分是哪个`changeId`进行的操作），因此需要保证每个`changeId`存入`buffer`的`Map`表示不同，同时在调用对应的`wayId`需要进行`instanceOf`判断等等。因此我尽量会避免使用这种复杂但是能够相当好控制时间复杂度的方式。
    - 代码很容易冗长。这弄得有点像操作系统的东西了。反观整个架构，确实非常复杂，而写代码一个很重要的原则是精简，这显然与之违背。
    - 还有debug的麻烦等等。

    具体的问题笔者也不细说了，虽然有着如此多的问题，**但是唯一有一点，个人觉得它在时间复杂度上实在是无可挑剔的；而且通过映射这种思维方式，能够使得其有相当良好的扩展性：只需要构建相应映射，那么所有的操作都能够在类之间自动运行。**这也大概是花费如此长的时间书写所获得的一点点安慰吧。（差点没赶上bug修复截止时间）

- **JML理解bug**

  这个也是容易出现的bug之一，实际上笔者出现的这类bug不仅是JML理解出错（括号），也涉及性能问题，于是在这个部分下一起与读者进行褒贬与夺。

  这便是反复折腾人的`MyGroup`中`getAgeVar`、`getValueSum`、`getAgeMean`。重点关注`ValueSum`。

  - 在处理这三个方法时，一方面需要理解JML（尤其是括号），另一方面需要考虑执行时间。
  - 之前出现的理解性bug，是因为漏看了括号，导致是先用每个人的Age除以总人数，再相加；而实际上是先将总age数相加，再除以总人数。
  - 考虑时间的问题，其中`getAgeVar`和`getAgeMean`通过设置一个`sumOfAge`统计，然后可以利用概率统计中的知识，利用`sumOfAge`等统计变量进行计算。
  - 其中最为特殊的便是`getValueSum`。相对于其他两个不大容易直接通过一两个变量调整时间复杂度，然而这个地方也频繁会出现TLE（简单而言任何算法都不能出现大于$O(n)$的情况）。对于这个地方我作以下考虑：
    - 一方面，`MyGroup`中增加或者删除`Person`时，将其与每个组内其他元素加/减**2倍**的`queryValue`。
    - 另一方面，由于中途可能会进行`addRelation`从而使得组内原本`queryValue`为0的两个元素发生改变，因此在`addRelation`中，如果

两个人恰好在同一组中，则需要增加**其value值的两倍。**

- 为了方便判断两人是否在同一组中，在`MyPerson`中增加映射`Map<Integer, Integer> groupId`。

基于如上的考虑，成功降低了时间复杂度。但是仔细思考实际上上述做法也不够完善——`atg`操作和`dfg`操作时间复杂度可能最大达到$O(n)$。

- **其他bug**

其他bug包括一些目测就能看出的bug，比如第一次作业中笔者采用DFS竟然忘记进行标记，从而出现了递归死循环，这些问题都是需要尽量避免的，因此也只说到这里。

四、JML测试方案

由于个人不太清楚如何构造数据进行测试，因此这部分对JUNIT进行相关介绍，提供一些可行的测试思路。

- 利用JUNIT进行单元测试

单元测试相对于普通测试而言是级别最低的一种测试（个人甚至觉得单元测试似乎并不那么有意义），是最小粒度的测试，常常用于测试摸一个代码块。但是这种测试是**由程序员来负责的**。因此仍有必要去了解其使用。

- JUnit框架

TestCase类	**测试对用户类的初始化以及测试方案的调用**
TestSuite类	负责包装和运行所有的测试类
TestRunner类	是运行测试代码的运行器

JUnit通过以上三个类进行单元测试，最后的结果显示也是通过`TestResult`=`TestCase`、`TestSuite`、`TestRunner`进行输出。

- 断言检查

对于断言检查，实际上主要是对于`Assert`相关语法的应用。

常见的断言检查语法有以下方式：

Assert方法名称	作用
assertEquals(Object expected, Object actual)	判断期望值和实际值是否相等
assertTrue()/assertFalse()	判断条件是真是假
assertThat	用法多样，大致可以认为判断前一个参数是否满足后一个参数条件

可以参考网络中的assert使用说明。

- JUnit注解

  对于JUnit使用中需要注意注解，区分使用规则。

  - @Before：标注每一个测试方法**执行前都要执行**的方法

```
- @After：用于标注每一个测试方法**执行后都要执行**的方法
```

```
- @Test：表示其为一个测试方法
```

```
- @Ignore：标注不参与测试的方法
```

```
- @BeforeClass：整个类的所有测试方法运行之前运行一次,且必须是static void
```

```
- @AfterClass：整个类的所有测试方法运行之后运行一次,且必须是static void
```

由于个人在实际测试中觉得这个东西并不好用，只针对`MyPerson`等简单的类进行过测试，而对于过于复杂的`NetWork`实在过于复杂。因此这个地方也不能有更好的间接，可以参考网上其他博文有更进一步分析。

最后提一下网络上谈及的JML工具链，同时针对OpenJML等验证手段进行了强烈讽刺。本次作业似乎也没有提及相关内容，可能这个地方因某些原因删除了。

无论如何测试是本次课程中很大的学问。但是我至今都很难get到点，希望最后一单元中，能够突破测试难关，真真正正达到课程训练的每一方面吧。

五、总结

本次博客内容很长，不过确实感触很多——第一次接触如此海量的测试数据，第一次必须直面性能和优化。想当初第一单元~~那个相当冗长的导数式~~和第二单元~~死活过不去的强测8,9两个点（一直等了近30s才输入）~~。性能的思考也在逐步加深。最后，期待下一个单元。