山东大学 软件 学院

数据结构与算法 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202100300063 | 姓名： 李彦浩 | | 班级： 工业软件菁英班 |
| 实验题目：等价类实现及其应用 | | | |
| 实验学时：48 | | 实验日期： 2023.01.05——2023.02.23 | |
| 实验目的：  问题描述：某工厂有一台机器能够执行n个任务，任务i的释放时间为ri（是一个整数），最后期限为di（也是整数）。在该机上完成每个任务都需要一个单元的时间。一种可行的调度方案是为每个任务分配相应的时间段，使得任务i的时间段正好位于释放时间和最后期限之间。一个时间段不允许分配给多个任务。  基本要求：  使用等价类实现以上机器调度问题。  等价类分别采取两种数据结构实现。 | | | |
| 硬件环境：  ThinkPad P15V,处理器Intel i5 @ 1.60GHZ | | | |
| 软件环境：  NodeJS运行环境、MySQL数据库、MySQLWorkBench数据库辅助工具、VSCODE集成开发环境 | | | |
| 实验步骤与内容：  题目内容  某工厂有一台机器能够执行n个任务，任务i的释放时间为ri（是一个整数），最后期限为di（也是整数）。在该机上完成每个任务都需要一个单元的时间。一种可行的调度方案是为每个任务分配相应的时间段，使得任务i的时间段正好位于释放时间和最后期限之间。一个时间段不允许分配给多个任务。  基本要求：使用等价类实现以上机器调度问题。等价类分别采取两种数据结构实现。  需求分析  1.利用数组、链表（模拟指针）、树（模拟指针）的数据结构实现等价类，其中树的结构要求使用路径紧缩的方式减少总体查找的时间。  2.增加三个功能：任务的执行时间不一定是单位时间、用户可以对每一个任务预设优先级、任务的的可执行时段不一定连续，可以是分时段的。  2.1 执行时间:指一个任务需要执行的时间。例如，一个任务的执行时间为2，则需要执行2个单位时间这个任务才算执行结束。  2.2 优先级:不采用动态调整优先级的固定策略。（如通过ci=pi-wi,wi=c-si,pi=ei-si其中ci为当前任务优先级，si为开始时间，ei为结束时间，c为当前时间。实际上在这是第十六题的策略）用户可以手动调整每个任务的优先级，输入一经提交优先级不可更改。  2.3分时段:原题中一个任务的开始和结束的时间组成了一个连续的时段。分时段允许一个任务可执行的时段是离散的。例如2:00-4:00和7:00-9:00，这两个时段都是同一个任务的可执行时段，该任务在这两个时段中成功完成即可。  3.创建web服务，将算法模块作为服务端的依赖。辅以一个美观的客户端系统，客户可以通过操作前端页面来获取算法输出的结果。  4.为了确保代码具有一定的鲁棒性，建立本地测试模块，引入算法模块作为依赖，生成随机（某些需要具有较为刁钻的边界）的，结果不同的数据进行测试。  数据结构设计  共有三种数据结构实现等价类，分别为：  1.数组(未投入生产环境)  2.链表(模拟指针，曾投入生产环境，目前弃用)  3.树（使用路径紧缩优化整体查找的效率，目前使用）  3.1 树节点的设计  class EquivTreeNode{  equivClass *// 等价类 同时也是最近前置位的标识*  parent *// 伪树结构的父节点模拟指针 但当该节点为树的根节点时代表这棵树的节点数*  root *// 是否为根的bool标识*  constructor(equivClass,parent,root){  this.equivClass=equivClass;this.parent=parent;this.root=root;  }  }  其中,equivClass代表等价类，parent在root为true时代表整棵树的节点数，否则代表该节点的父节点。  3.2 合并规则  合并时，必须遵从equivClass较大的等价类合并入equivClass较小的等价类中，而不能采用任何可以增进合并效率的规则，如高度规则、重量规则。这是因为，在基础算法的实际意义中，unite操作实际上将当前任务调度到某一位置上，并令该位置为后续任务指路，指出前方最近可用的位置，这是有时间先后意义的，因此必须按这个顺序合并。  3.3 路径紧缩  这是众多增进查找操作效率的方式中的一种，类似的还有路径对折等方式。简单来说，这种方式将每次从当前待查找节点到根节点的路径上的每一个节点（除根节点）直接挂在了根节点下，成为根节点的子节点，以后再查找时，路径长便为1，增进了查找效率。    树结构与路径紧缩（红线为节点9成为节点5的子节点）  应用程序设计  （作者按：主要分为三个部分：数据格式的要求，前后端通信的设计，以及算法模块的工作流程）  1.数据格式的要求  待调度的任务列表以一个javascript的对象数组传入，其中包含字段：任务名称(name)、开始时间(start)、结束时间(end)、执行时间(exeTime)、优先级(weight)。下面是一个实例。  [  {  name:'A',*//任务名要求为字符串，可空(用户对此负责)*  *//开始结束时间均为自然数数组，0索引弃用，且同一个时段的开始和结束时间在对应数组中的索引需相同(客户端为此负责)*  start:[null,2,13],  end:[null,6,16],  exeTime:3,*//执行时间要求为大于等于1的正整数*  weight:5*//优先级可从1到5选择，1为最小优先级，逐渐递增至5*  }  ]  2.怎样定义一个脏数据？  上述的数据格式要求中实例的注释已经将各字段的合法格式描述了一遍，这里唯一需要额外说明的是开始结束时间数组。我们不允许同一个任务存在两个时段，其交集不为空。但两个时段是可以相接的。例如，2：00-4：00和3：00-5：00是一个脏数据，因为它可以写为2：00到5：00。但2：00-4：00和4：00-6：00虽然也能被写作2：00-6：00，但因为两个时段仅仅是相接，被认为是一个合法的数据。    脏数据图解  3.如何维护输入信息？  对于任务名称、执行时间和优先级，我们只需要让客户端在输入时向数据域中待发送的任务对象数组插入新的对象，这个对象中包含当前一次输入的三个对应字段。但还有两个字段，分别是开始时间和结束时间，不能简单地直接插入当前任务对象的对应字段的数组中，纵观整个工作流程，认为在这里使用堆这一数据结构维护这两个字段比较好。  根据后续工作流程，这里使用小根堆。当每次需要向堆中插入新的时段时，我们通过比较新时段的开始时间和结束时间堆顶的时间以及新时段的结束时间和开始时间堆顶的时间来确定是否有交集来判断是否是脏数据。若数据合法，则进行插入；否则提示错误信息。这一做法使得维护输入信息的程序拥有对数时间的性能，而如果使用有序数组或有序链表，将会是线性时间。  4.基础算法的流程与实际意义  这里先不考虑任务的执行时间、优先级和分时段，仅考虑构成连续可执行时段的开始结束时间。  第一步，按照任务开始时间的大小降序排列，即，开始最晚的任务在整个列表最前方，依次递减。  初始化一个等价类数组，0索引弃用，长度自定义，其中每个元素的等价类为当前数组索引。以同样的方式伴随一个空的结果数组。  每次取出当前列表中头部的任务，根据其结束时间查询它的等价类，并判断这个等价类是否超出了它的可执行时段的范围。若超出，调度失败；若不超出，将该任务安排到结果数组中索引为其等价类的位置，并合并当前等价类与其前一个位置的等价类。重复这个过程直到某一个任务调度失败或是整个任务列表中的任务均得到调度。  接下来我们将等价类中的三个重要函数,initialize,unite,find与三个实体操作进行双射映射，这有助于我们理解这个算法的实际意义。  initialize函数对应操作一：设置一排椅子，这排椅子都将自己的索引作为标号记录在椅子上，这个标号的实际意义是指出距离当前椅子的最近可用的椅子的位置。unite函数对应操作二：将这个标号修正为这把椅子前一把椅子的标号，这个操作的实际意义是，当这个椅子被占用时，若后续有任务来询问这把椅子他是否可用时，这把椅子会告诉这个任务：“我已经不可用了，但我知道在我前方离我最近的且是可用的那把椅子在哪里，要想知道这个椅子的位置，看我的标号即可。”。find函数对应操作三：根据当前查询到的椅子去找到位置索引等于当前椅子标号的椅子，这是在查询椅子前方最近可用的椅子。    等价类的三个操作与实际操作的双射映射  这样整体工作流程就变成了，每个任务去查询自己可用的椅子（即开始结束时段内）中的最后一把椅子，询问这把椅子前方（包括这把椅子本身）的所有椅子中，距离这把椅子最近的可用的椅子。例如一个任务可用的椅子是13号至16号椅子，它上来先查询了16号椅子，16号椅子为其指路，告诉该任务可用的椅子是14号椅子，该任务发现14号椅子是自己可用的椅子，欣然坐在了14号椅子上。如果16号椅子指出的椅子是11号，该任务发现11号椅子不可用，则说明13号到16号椅子均被占用，自己将无处可去，则调度失败。  接下来说明为什么要按开始时间降序排列，严谨的证明会在后续的部分给出。这里通过一个实例来感受它的作用。  [  {  name:'任务1',  start:0,  end:2  },  {  name:'任务2',  start:1,  end:2  }  ]  这两个任务的执行时间都是单位时间。如果就按照现在这个顺序调度。首先任务1会查询2号位置，并发现2号位置指出自己可用，于是他就占用了2号位置。这样2号位置更正指路结果为1，改指1号位置可用。任务2同样查询2号位置，发现执行1号位置，可1号位置代表时段0：00-1：00,不在1：00-2：00中，于是调度失败。    调度失败图解  但明明可以让任务1去1号位置，任务2去2号位置完成一个成功的调度。于是，这个降序排列的意义便不言自明了：即让开始时间晚，可操作空间小的任务优先得到调度，防止被操作空间大的任务卡住位置。    为什么要按开始时间降序排列图解  5.加入额外功能后的调度策略  我们可以将额外功能拆解为子任务来逐一完成。  5.1执行时间  我们可以这样拆解一个任务A：A=（a1,a2,a3,...,an)，其中n为A任务的执行时间。而ai代表一个子任务，该任务在除了执行时间上为单位时间，其余成员与A任务的成员完全一致。这样就把一个大任务拆解成一堆单位时间的小任务，然后进行基本算法的调度即可。  5.2优先级  对于优先级，我们的追求是，在满足调度的基础上要让用户设置的优先级较高的任务尽可能早地被执行。但这里的“尽可能早”一词的含义不同于往常。仿照上学期数据结构课程中学习的WEP（外部路径加权和）的概念，得出一个新的定义：  WPLT（Weighted Priority Latency Time,加权优先级等待时间）  = 其中n为总调度时长。  我们的目标是要在满足调度的基础上找到一个WPLT最小的方案。  为了完成这个功能，我们采取先调度，后调整的策略。即，先按照基本算法完成调度，然后通过操作，将其调整为优先级最优。  调整的策略如下：假设我们有一位监管者，想要让优先级高的任务尽可能早执行。这位监管者从上述的一排位置的头部开始，（这排位置上已经通过基本算法完成了调度）不断命令当前走到的位置上的任务（可能没有任务，这可以通过构造特殊对象实现，例如空位置认为开始时间为0，结束时间为理论最大值，比如24，即一天中最后的时间）向当前位置后面的所有位置询问是否可以位置，这个询问包括两部分：首先要保证当前任务置换到后续某一个位置时，后续的那个位置位于当前任务的可执行时段内；第二要保证后续任务置换到当前位置时，当前位置位于后续任务的可执行时段内。在可交换的基础上，比较优先级，若当前任务优先级更高，不予交换；若后续位置的任务优先级更高，交换。这样一轮下来，可以保证每个位置上的任务是所有可交换的任务中优先级最高的，这样就完成了优先级最优的调整。  5.3分时段  实际上，分时段只不过是流程上复杂了一些而已，并没有修改这个算法的根本。在调度时，逆序遍历所有可执行时段，（这些时段升序排列）我们将查询得到的结果置于每个可执行时段中查询，如果位于某一个可执行时段中，结束遍历，安排任务于当前位置。如果任何一个可执行时段都不包含这个查询结果，调度失败。  调整时，在两次询问过程中，只要保证两个任务各存在一个可执行时段，交换后的位置分别位于这两个时段中即可。  6.最终工作流程  6.1如何排序？  6.1.1先来看一个例子  任务A有两个时间段,2:00-4:00,7:00-8:00;任务B只有一个时间段，7：00-8：00。那么此时虽然任务A的最大开始时间和任务B的最大开始时间一样，按理说先分配任务A即可。但是任务A一旦分配到7-8，那么任务B将无处可去。但显然任务A是可以分配到2-4的。  6.1.2最小中的最大  之所以成为“最小”中的“最大”，是因为我决定将每个任务的开始时间中的最小值拿出来，进行排序。依次调度拥有当前这群最小值中的最大值的任务。以2.1中的例子为例，实际上代表A的开始时间是2:00，而不是7：00。这样可以保证B先进行调度。而将最小值降序排列，可以保证那些开始时间靠后，比较紧张的任务可以优先调度。  6.2整体工作流程  1.用户填写数据，此时不需要整理时间堆和任务堆，交给算法模块整理  2.数据交送后端，接口转送算法模块，首先转换为实体类Task数组列表  3.在转化为实体数组的过程中，先对每个任务的开始结束时间做升序的堆排序  4.然后将时间堆排序后的任务列表做基数排序  4.1 首先先对优先级做升序的桶排序  4.2 然后对开始时间做降序的堆排序 这样任务列表按开始时间为第一优先级，优先级为第二优先级进行的排序  5.初始化结果数组，0索引弃用  6.初始化等价类数组，0索引弃用  7.根据任务列表做调度  7.1 获取当前任务  7.2 循环直到将这个任务执行完毕(剩余需执行时长为0)  7.2.1 遍历所有时段，每个时段的结束时间所在位置指出了在其之前最近的可用的位置，检查这个位置是否在当前时段内，如果在，则占用此位置并合并至前一个等价类。如果不在，则检查下一个时段。  7.2.2 若所有时段均检查完但并未成功调度，那么整个调度方案失败。若其中的某一个时段完成本轮单位时长的调度，则将之后每次单位时长的调度的结束时间改为该时段的结束时间。  8.遍历整个结果数组，将结果调整至优先级最优  8.1 获取当前位置(可能有任务也可能没任务)  8.2 循环查询当前位置后的所有位置 8.2.1 两次询问  8.2.1.1 第一次，询问当前任务若换到后面遍历到的某一个位置时，这个位置所在的时段是否处于当前任务的可执行时段中的某一个。也即，遍历当前任务的开始和结束时间，如果后面的这个位置处于当前的时段，则进行第二次询问  8.2.2.2 第二次，询问后面位置的任务若换到当前位置，当前位置所处的时段是否处于后面的任务的可执行时段中的一个。方式同8.2.1.1  8.2.2 一次比较 8.2.2.1 若8.2.1的两次询问的结果时可以交换，那么比较优先级，如果后面任务的优先级较高，则交换；否则，不换 这个循环过程持续到没有交换产生为止  9.结果数组由业务接口响应回前端  10.前端接收后，预处理结果数组，并回显  存在的缺陷   1. 精度的问题   本程序中取单位时间为一小时，但单位时间可能有其他的可能性，导致精度不够，适配场景较少。单位时间的可能性：1小时，半小时，10分钟，1秒，1微秒，1纳秒……  2.精度带来的复杂度  精度高了后，等价类数组将非常大。例如单位时间为1ns，则等价类数组长度为  (24\*3600\*10E9+1)，任务数量大了后，高复杂度将带来低运行效能。  3.功能上的呆板  是否可以考虑抢占？（实际角度）调度策略只能有一种吗？（课设16题）是否能寻找并行最优？（课设35题）  调度策略上，例如，用户预设优先级，然后和任务结束时间与当前时间之差相加，得到最终优先级，进行调度？（当然本题背景中无“当前时间”这个概念，但能否扩展题目背景？）  4.编码规范  每一个比较长的流程都抽象成函数了吗？算法模块的层次结构如何？是否有良好的设计模式？如何对当下代码进行解耦/重构？项目的可维护性、可扩展性如何？  5.性能  是否有更好的算法与数据结构？暴露为web服务，可以在短时间抗下多少运算量？  编码  编码部分见具体代码，报告末会呈交一份关键部分的代码。  测试  建立本地测试模块，通过excel软件随机生成数组进行代码鲁棒性、安全性的检验，保证结果与实际描述的功能相符合。 | | | |
| 心得与体会：  1.算法普适度  如果一个算法（策略）仅仅可以解决一个普通的且狭隘的场景下的问题，即便它的复杂度很优秀，可能也不是一个好算法。  2.怎样选择合适的数据结构？  存储方面，数组、链表、栈、队列、散列、各种树还是其他？增删改方面，怎么做到整体性能最好？（路径紧缩）  3.项目工程的能力  技术选型上，选用什么技术，这套技术栈的优点在哪?具体实现上，架构、设计模式如何学习应用？ | | | |

**实验主要代码：**

1.等价类树结构节点

class EquivTreeNode{

    equivClass // 等价类 同时也是最近前置位的标识

    parent // 伪树结构的父节点模拟指针 但当该节点为树的根节点时代表这棵树的节点数

    root // 是否为根的bool标识

    constructor(equivClass,parent,root){

        this.equivClass=equivClass;this.parent=parent;this.root=root;

    }

}

2.基于树的等价类的三个操作

function initializeInTree(length,Nodes){

    //初始化等价类数组 树结构节点

    Nodes.push(new EquivTreeNode)

    for(i=1;i<=length;i++){

        Nodes.push

        (

            //一开始的每一个节点都是自己这棵树的根,则parent域为1(节点个数为1),等价类也是自己的索引

            new EquivTreeNode

            (

                i,1,true

            )

        )

    }

}

//这个合并不能用重量规则 因为时间是有先后顺序的 必须是后面的时间合并到前面去

//不然先占了4 再占3 再占2 则为3->4|->2(|表示统一层级) 后来人问路4 是2

function uniteInTree(rootA,rootB,Nodes){

    //特判x合并到0

    //因为没有0这个节点 所以相当于没有实际的合并操作

    if(rootA===0){

        //修改rootB的等价类，提醒后人前面已经不再有空位了

        Nodes[rootB].equivClass=0

        return

    }

    //修改节点数量

    Nodes[rootA].parent+=Nodes[rootB].parent

    //修改B的根节点指向

    Nodes[rootB].parent=rootA

    //修改rootB的标识符 不为根节点了

    Nodes[rootB].root=false

}

//路径紧缩的查找方式 首次查找比较麻烦 但是方便了后续的查找

function findInTree(ele,Nodes){

    if(ele===0){

        return 0;

    }

    //定义根节点指针 最终返回

    var theRoot = ele

    ;

    while(!Nodes[theRoot].root){

        // console.log(theRoot);

        theRoot = Nodes[theRoot].parent

    }

    //紧缩路径 让这条路径上所有的节点统统指向根节点

    var currentNode = ele

    while(currentNode!==theRoot){

        let parentNode = Nodes[currentNode].parent

        Nodes[currentNode].parent=theRoot

        currentNode=parentNode

    }

    //未改善过的代码中我们认为节点的数组索引就是等价类

    //但是现在我们要通过等价类是否为0提示调用方前方是否还有空位

    //而数组索引是肯定不涉及到0的

    //因此即便大部分情况下theRoot的等价类equivClass就是数组索引theRoot

    //但我们还是要返回equivClass

    return Nodes[theRoot].equivClass

}

3.对任务按照开始时间为最高优先级、优先级为次高优先级的排序

这里面包含了对开始结束时间堆的排序和对任务本身的排序

//单位时间有优先级任务的桶排序

//@ param : standard 为标记,指示按开始时间排还是按优先级排

function binSortWeightedTask(tasks,standard,range){

    //一堆桶

    var bins = new Array()

    //初始化这堆桶

    for(i=0;i<=range;i++){

        bins.push(new TaskChain(null,0))

    }

    //根据开始时间排序,开始越晚越靠前

    if(standard===byStart){

        let length = tasks.length

        for(i=0;i<length;i++){

            let task = tasks[i]

            //修改开始时间指向的桶

            bins[task.start].insert(task,0)

        }

    }

    //根据优先级排序,优先级越高越靠前

    else if(standard===byPriority){

        let length = tasks.length

        for(i=0;i<length;i++){

            let task = tasks[i]

            //修改优先级指向的桶

            bins[task.weight].insert(task,0)

        }

    }

    //把所有桶中的元素收集回tasks[]数组

    var resultChain = new TaskChain(null,0)

    for(j=0;j<=range;j++){

        while(!bins[j].empty()){

            let task = bins[j].get(0)

            resultChain.insert(task,0)

            bins[j].erase(0)

        }

    }

    //把结果链表里的值全部复制给tasks

    tasks=resultChain.copyToArray(tasks)

    return tasks

}

//单位时间有优先级任务的基数排序

function radixSortWeightedTask(tasks){

    //先按照优先级进行桶排序，再按照开始时间进行堆排序

    tasks=binSortWeightedTask(tasks,byPriority,rangeForPriority)

    // tasks=binSortWeightedTask(tasks,byStart,rangeForStart)

    tasks = heapSortTasks(tasks)

    return tasks

}

//任务堆要降序

function heapSortTasks(tasks){

    //预处理tasks 使它的0索引弃用

    for(i=tasks.length-1;i>=0;i--){

        tasks[i+1]=tasks[i]

    }

    tasks[0]=new Task(null,null,null,null,null)

    var heap = new maxHeap()

    heap.initialize(tasks)

    var newTasks = new Array();newTasks.length=tasks.length-1;

    //tasks任务数组的0索引已经弃用了 所以他的length是n+1 0到n-1才是n个

    for(i=0;i<tasks.length-1;i++){

        var ele = heap.top()

        heap.pop()

        newTasks[i]=ele

    }

    //返回的tasks的0索引是启用的

    return newTasks

}

//时间堆其实最好降序 但是降序之后任务堆不方便构建 所以写成升序了

//顶多是在等价类合并的时候需要倒着遍历所有可执行时段(因为要根据结束时间降序)

//这个函数使用在 前端把Json的任务数组送过来后,对其进行排序，这样后续先对任务进行堆排序时可以通过1号索引访问start的最小值

//再对等价类做合并时，可以倒着遍历所有时段，从而进行调度

function heapSortTime(time){

    var heap = new minHeap()

    heap.initialize(time)

    let length = time.length

    //js里面是数据的引用，如果在下面的for循环里修改构建堆的数组，那么堆中的成员变量是这个数组的引用，也就是修改了堆的成员变量本身

    var newTime = new Array();newTime.length=time.length;newTime[0]=null

    //前端送过来的数据中,时间数组的0索引弃用了

    //不能使用i<=time.length，因为js的数组是不定长的

    for(i=1;i<=length-1;i++){

        var ele = heap.top()

        heap.pop()

        newTime[i]=ele

    }

    return newTime

}

4.调度以及调整到优先级最优

//初次调度结果

function getResult(result,Nodes,tasks){

    let length = tasks.length

    for(let i=0;i<length;i++){

        var task = tasks[i]

        //剩余执行时间

        var restTime = task.exeTime

        //剩余任务的结束时间始发点

        var restEnd = task.end[task.end.length-1]

        //第一个查找的位置是最后一个结束时间指出的位置

        var equiv = findInTree(restEnd,Nodes)

        while(restTime!==0){

            //标志着所有可执行时段中是否有可用的位置

            var flag = false

            //一旦equiv===0 说明前面的位置全没了，不用再往前找了

            if(equiv!==0){

                //逆序遍历所有时段，查询是否某个时段中有可用的位置

                for(let i=task.start.length-1;i>=1;i--){

                    let curStart = task.start[i]

                    if(equiv > curStart){

                        flag = true

                        //更新剩余任务始发点

                        restEnd = task.end[i]

                        //已经找到合适的位置，停止遍历

                        break

                    }

                    else{

                        //如果本轮不成功，让下一轮的end指出可用的位置

                        //当i为1，此时已经是所有时段中的最后一个，前面的判断说明前方已经没有位置了，调度失败

                        if(i===1){break}

                        else if(i>1){

                            equiv = findInTree(task.end[i-1],Nodes)

                        }

                    }

                }

            }

            if(equiv===0 || !flag){

                //调度失败

                return null

            }

            else{

                //调度成功

                //分配任务进结果数组

                result[equiv] = new Task

                (

                    task.name,

                    task.start,

                    task.end,

                    task.exeTime,

                    task.weight

                )

                //等价类数组合并

                uniteInTree(findInTree(equiv-1,Nodes),equiv,Nodes)

            }

            restTime += (-1)

            //之后的每个单位时长都从当前结束时间始发点出发即可，后面已经没了

            equiv = findInTree(restEnd,Nodes)

        }

    }

    return result

}

//两次询问的函数

/\*\*

 \*

 \* @param {当前任务} currentTask

 \* @param {后面的某个任务} afterTask

 \* @param {当前任务索引} indexOfCur

 \* @param {后面的任务的索引} indexOfAft

 \*/

function doubleHandShake(currentTask,afterTask,indexOfCur,indexOfAft){

    var flagOfCur = false

    var flagOfAft = false

    //遍历当前任务的开始和结束时间

    for(i=1;i<=currentTask.start.length-1;i++){

        //检查后面的位置是否处于当前时段中

        let curStart = currentTask.start[i]

        let curEnd = currentTask.end[i]

        if(curStart<indexOfAft+1 && curEnd >=indexOfAft+1){

            flagOfCur = true

            break

        }

    }

    if(!flagOfCur){return false}//第一次握手就失败了 不能交换

    //遍历后面的任务的开始和结束时间

    for(i=1;i<=afterTask.start.length-1;i++){

        //检查当前位置是否处于后面的任务的可执行时段中

        let aftStart = afterTask.start[i]

        let aftEnd = afterTask.end[i]

        if(aftStart<indexOfCur && aftEnd>=indexOfCur){

            flagOfAft = true

            break

        }

    }

    //保险起见取个与

    return flagOfCur&&flagOfAft

}

//调整至优先级最优

function adjustByPriority(result){

    //标志着上一个轮次是否有元素产生交换

    var flag = true

    while(flag){

        flag=false

        for(let i=1;i<=hours;i++){

            // console.log(result[i].name + ' ' + i);

            var currentTask = result[i]

            //遍历这个位置后面的所有位置，询问是否有元素交换

            for(let j=i+1;j<=hours;j++){

                var afterTask = result[j]

                //两次询问

                if(doubleHandShake(currentTask,afterTask,i,j)){

                    //一次比较

                    if(currentTask.weight<afterTask.weight){

                        let temp = new Task

                        (

                            currentTask.name,

                            currentTask.start,

                            currentTask.end,

                            currentTask.exeTime,

                            currentTask.weight

                        )

                        result[i]=afterTask

                        result[j]=temp

                        flag=true

                        //更新当前元素

                        currentTask = result[i]

                    }

                }

            }

        }

    }

    return result

}