算法大作业

1. 对已知算法在计算机科学与技术领域的应用进行调研
2. 地图软件-最短路算法
   1. **最短路算法介绍**

最短路算法（shortest path algorithm）是为解决最短路径问题的算法，常见的有迪杰斯特拉算法（Dijkstra算法）（可进行堆优化），Bellman-Ford算法，SPFA算法（队列优化的Bellma-Ford算法）和Floyd-Warshall算法以及A\*算法、D\*算法、LPA\*算法、D\* Lite算法。

* 1. **各个算法介绍**

1. Dijkstra 算法

该算法采用了一种贪心模式，其解决的是有向图中单个节点到另一节点的最短路径问题，其主要特点是每次迭代时选择的下一个节点是当前节点最近的子节点，也就是说每一次迭代行进的路程是最短的。为了保证最终搜寻到的路径最短，在每一次迭代过程中，都要对起始节点到所有遍历到的点之间的最短路径进行更新.

主要是通过贪心原则逐个遍历最小子节点，然后利用松弛方法去优化路径选择，最终将最优路径存放到可读列表当中，以此来解决最优路径规划问题.

1. A\*算法

A\*算法A\*算法是启发式搜索算法，启发式搜索即在搜索过程中建立启发式搜索规则，以此来衡量实时搜索位置和目标位置的距离关系，使搜索方向优先朝向目标点所处位置的方向，最终达到提高搜索效率的效果。f(x)=g(x)+h(x)其中g(x)是从起点到当前节点x的实际距离量度, h(x)是从节点x到终点的最小距离估计算法基本实现过程为．从起始点开始计算其每一个子节点的f值，从中选择f值最小的子节点作为搜索的下一点，往复迭代，直到下一子节点为目标点.

1. D\*算法

D\*算法是一种反向增量式搜索算法，反向即算法从目标点开始向起点逐步搜索；增量式搜索，即算法在搜索过程中会计算每一个节点的距离度量信息H(x)，在动态环境中若出现障碍物无法继续沿预先路径搜索，算法会根据原先已经得到的每个点的距离度量信息在当前状态点进行路径再规划，无需从目标点进行重新规划。

1. LPA\*算法

LPA\*算法实现原理：搜索起始点为所设起点（正向搜索），按照Key值的大小作为搜索前进的原则，迭代到目标点为下一搜索点时完成规划；Key值中包含启发式函数h项作为启发原则来影响搜索方向；处于动态环境时，LPA\*可以适应环境中障碍物的变化而无需重新计算整个环境，方法是在当前搜索期间二次利用先前搜索得到的g 值，以便重新规划路径。

1. D\* Lite算法

D\* Lite算法是先在给定的地图集中逆向搜索并找到一条最优路径。在其接近目标点的过程中，通过在局部范围的搜索去应对动态障碍点的出现。增量式算法的优势在于：各个点的路径搜索己经完成，在遇到障碍点无法继续按照原路径进行逼近时，通过增量搜索的数据再利用直接在受阻碍的当前位置重新规划岀一条最优路径，然后继续前进。

* 1. 地图软件功能

像 Google 地图、百度地图、高德地图这样的地图软件。如果想从家开车到公司，你只需要输入起始、结束地址，地图就会给你规划一条最优出行路线。这里的最优，有很多种定义，比如最短路线、最少用时路线、最少红绿灯路线等等。

* 1. 实际应用及难点

在研究路径选择和流量分配等交通问题时，常常会用到最短路算法。

用最短路算法解决交通问题存在两个难点：

一、算法的选择和程序的编写。最短路算法有很多种改进算法和启发式算法，这些算法的效率不同，适用的网络也不相同。

二、构建一个算例网络很简单，但由于实际路网具有高度复杂性，因此将真实的拓扑路网导入最短路算法变得困难。

* 1. 解决方式

明确一点，基本的图搜索算法dijkstra是无法满足互联网地图检索实时响应这种性能要求，所以各家公司都有各自的预处理方法：分层或者预计算。具体采用何种方式，这取决于采取的加速算法相关。在2008年前后，以KIT为主的研究院产出了多个路径规划加速算法，其中以contraction hierarchies 和 highway hierarchies较出名，加之微软研究院提出的Customizable Route Planning，与传统的A-star，基本上支撑了目前工业界地图产品的路径规划服务。

1. **contraction hierarchies**

当谈论到两点之间的路径规划算法，我们首先想到的是Dijkstra算法，为了提升效率又改进出现了Bidirectional Dijkstra算法。但对于国家，大陆级别的道路网规模，查询效率还是太慢，甚至超过2s，很难达到ms级的响应。

Contraction Hierarchies算法即是在原始Graph先进行预处理，提前先算好一些点与点之间的最短路径距离(Shortcuts)，简化Graph的Edge个数，再利用改进的Bidirection Dijkstra找到最短路径。

1. **highway hierarchies**

highway hierarchies是一种新的加速最短路径查询方法。算法使用线性的存储需求在几个小时内预处理了千万数量级结点数的美国和西欧地图，进而可以在毫秒级时间内完成最短路径查询，这比简单使用Dijkstra算法快大约2000倍。

1. **customizable route planning**

crp 算法由微软硅谷研究院三名工程师于2013年提出。

其设计目标为：

* 应用于大陆级别的实际路网数据
* 支持任意类型metric(可以理解为cost function)
* 响应时间满足实时查询的需求
* 快速的路况更新以及定制化metric更新

1. Alpha Star - 进化算法和机器学习
   1. 成就

在Alpha Zero取得围棋领域的胜利之后，最近的大新闻是： AI又下一城，拿下了重磅即时战略游戏星际争霸。

Alpha Star, 在星际争霸这样级别的游戏击败人类， 应该说这是一个了不起的胜利。首先， 星际争霸， 是一个与围棋有着本质不同的游戏-那就是星际争霸是非完全信息下的博弈游戏。 这里的难点一个是不完全信息， 其次是需要远期计划，另外就是实时性与多主体博弈。 因为星际争霸这个游戏比围棋要接近真实世界很多， 信息是局部的，远方的世界笼罩在黑雾里， 当下和未来是连续的， 战略需要跨越很多时间尺度。 这些问题， 需要比阿法狗更加复杂的接近方法。

* + 1. 非完全信息

何为非完全信息， 你看所有的即时战略游戏， 从红警到星际，你没有刚刚到达过的区域都被一团战争迷雾所笼罩， 那里可能有对方的军队在采矿， 或者大批小兵集结， 这可能与我此刻的决策关系很大， 但是却不为我所知。 这与围棋这样的游戏有着本质的区别， 因为围棋这样的游戏， 即使策略在复杂， 你方和我方的情况都是一目了然的， 围棋游戏的复杂体现在策略空间的巨大导致的维度灾难， 然而始终都是你知我知的。

* + 1. 多体博弈

多体博弈， 体现了策略建模的复杂性， 我方的最佳策略取决于有方策略，对方策略，以及盟友的策略，还有兵种的相克，组合带来的1+1大于2的过等等，游戏尽可能的模拟了真实的战争差。星际争霸的挑战还有超大的动作空间（可能性超过围棋全部组合数百个数量级），游戏长度很长， 而且游戏初期做出的决定会影响最终的成败。但最关键的，星际相对围棋最大的技术挑战在于非完全信息博弈， 也是它和真实世界最接近和最有价值的地方。

* 1. Alpha Star介绍

Alpha Star以一个具有记忆的神经网络LSTM为基础， 然后用到的学习方法， 正是进化算法加机器学习（强化+监督学习）。 应该说， 这才是这套方法的灵魂。它的思想简直是对1994文章的升华， 把拉马克进化真正用起来。 也就是说， 我们把一个最外层的代系间的进化过程， 和内层的持续不断的强化学习结合起来。 内层的学习会影响外层的进化。 更重要的是， 在这里， 我们通过进化引入了不同参数的网络（智能体）间的博弈。 那些从一个根基上产生的稍有不同的网络， 会通过学习来改变自己， 并在战斗中一决雌雄， 当一轮结束， 优胜者将改写失败者的基因， 但失败者不会马上消失。

AlphaStar的目标是训练出一个可以战胜所有可能的策略的最强策略网络。作为一个强化学习的问题，其之所谓困难， 至少有以下几个原因：

1. 鲁棒性要求高：这是一个有很强的策略与反策略的游戏，学习打败某种策略相对容易，学习一个可以应对多种战术的策略非常难，这件事情无法直接通过简单的self-play 来解决；
2. 复杂的感知任务：智能体需要感知的信息有三维游戏世界地图信息、大量的军事单位、每一个军事单位和建筑的属性信息以及自身的一些资源属性信息；
3. 复杂的动作和决策空间：游戏动作维度本身很高，观测信息是部分可知的（我们无法完全知道对手目前的状态），且一场游戏的决策步数非常多，且策略过程非常复杂。
4. 词法分析程序（IDE等） - Thompson算法
   1. Thompson算法介绍

汤普森构造法是C语言&Unix之父之一的肯·汤普森(Ken Thompson)提出的构造识别正则表达式ε-NFA的方法，其原理非常简单，先构造识别子表达式的ε-NFA，再通过几个简单的规则将ε-NFA合并，最终得到识别完整正则表达式的ε-NFA。汤普森构造法的优点是构造速度快，且构造的ε-NFA状态数较少

Thompson构造法在计算机科学中是指一个能将正则表达式转化为一个与之等价的非确定有限状态自动机（NFA）的算法。算法得到的NFA可以在编程中用于匹配一个正则表达式，这也是正则表达式引擎实现的基本思路之一。

* 1. 正则表达式和非确定有限状态自动机

正则表达式和非确定有限状态自动机是形式语言的两种不同的抽象表达方式。在诸如文本编辑器的高级“查找和替换”以及许多编程语言中，人们都习惯使用正则表达式来表示字符串的匹配模式。然而，当计算机执行匹配程序时，NFA却是更加适合的一种格式。因此，Thompson构造法有着重要的应用价值，它实际上可以视作正则表达式到NFA的一个编译器。而从理论角度上来说，该算法实际上是正则表达式和NFA等价性证明的一部分——事实上，这两种表述形式本质上都对应着相同的语言，即正则语言。

在计算理论中，非确定有限状态自动机或非确定有限自动机（NFA）是对每个状态和输入符号对可以有多个可能的下一个状态的有限状态自动机。

非确定有限自动机是Michael O. Rabin和Dana Scott在1959年介入的，他们证明了它与确定自动机的等价性。

* 1. IDE等应用实现

在应用中，算法得到的NFA可以再次通过幂集构造和最小化的过程得到一个对应的最简的确定有限状态自动机（DFA），进而用于匹配正则表达式。但是有些情况下也会直接使用对应的NFA。

整个语法树的构建过程中需要一个词法分析器Lex，词法分析器从左到右逐个字符地扫描正则表达式，根据遇到的字符返回正确的Token给语法树构建器，对于不合法的正则表达式给出报错信息（例如转义字符\后面跟的不是特殊字符）。

语法树构建器拿到词法分析器返回的词法Token后，开始进行自下而上的建树过程，在不考虑括号的情况下，正确的正则表达式的第一个词法Token应该是一个非运算符，它被包装为语法树节点结构然后被压入语法树构建器的语法树节点栈中。之后第二个词法Token可能是一个运算符也可能是一个非运算符，如果是非运算符，则需要添加一个表示连接的cat运算符到运算符栈中，并将得到的操作数Token包装为语法树节点压入语法树节点栈中。每次向运算符栈中压入新的运算符new之前，都需要查看当前运算符栈顶的运算符old，和new谁的优先级更高，如果old的优先级较高，则先处理old运算符（会用掉语法树节点栈中的节点，运算得到的节点再压回语法树节点栈），old被处理完后，old出栈，接下来的栈顶元素成为old，再次和new进行比较，重复这个过程，直到old的运算符优先级低于new，再将new运算符压栈。如果遇到了左括号，则先将左括号压入运算符栈中，在遇到右括号时需要将运算符栈中的节点从栈顶开始处理，直到处理到最靠近栈顶的左括号为止。当正则表达式处理完后，最后再处理运算符栈中剩余的运算符。正确的结果应该是运算符栈为空，语法树节点栈中有一个节点，这个节点就是整个语法树的根节点。

1. 大数据推送
   1. 大数据推送介绍

大数据的精准推送是新媒体根据对大数据的信息化处理从而将人们想看的新闻、视频、段子、商品等置于优先位置的一种算法。算法推送即是一种编码程序，它可以对大数据时代泛滥的新闻信息进行分类、标签、整合、排序，然后通过特定的运算把输入数据转化为输出结果，再以特定的需求，有针对性地给相关用户进行推送。现在新闻传播及社交媒体之中存在的个性化推荐机制便是在算法推送的基础上，通过对受众的兴趣爱好特性，以及用户之间的社交关系进行多重分析，深度解析用户的需求，从而进行个性化的推送。

* 1. 推荐算法介绍

推荐算法顾名思义，就是通过一些数学算法，推断出用户可能喜欢的东西。

推荐算法的研究起源于20世纪90年代，由美国明尼苏达大学GroupLens研究小组最先开始研究，他们想要制作一个名为Movielens的电影推荐系统，从而实现对用户进行电影个性化推荐。如今推荐算法主要应用于网络，如我们熟知的bilibili，淘宝，抖音，微博等平台。在提高用户满意度的同时，增加了用户粘性。

* 1. 多种推荐算法

1. 协同过滤推荐

协同过滤是推荐算法中最经典的一种，包括在线的协同和离线的过滤两部分。所谓在线协同，就是通过在线数据找到用户可能喜欢的物品，而离线过滤，则是指过滤掉一些不值得推荐的数据，如评价较差的物品和用户已经购买过的物品。该方法的优点在于易于实现，不需要特定领域的知识。常分为基于用户的协同过滤，基于物品的协同过滤和基于模型的协同过滤。

1. 内容推荐

内容推荐是信息过滤技术的延伸和发展，它是建立在项目的内容信息之上所做出来的推荐。凭借机器学习对内容的识别和分析处理，对内容的特征按照一定的标准进行分类，从而获得用户得兴趣特征，进而扩展性的推荐。

如何通过基于内容的推荐，来对求职者A进行职位推荐：找到用户A感兴趣的职位集合，如程序员。找到职位集合的具体化内容，如所在城市，薪资，学历要求等。将内容中的共性列出来，如城市均为北京，薪资6000+等 。根据共性内容查找符合条件的职位，推送给用户A。

1. 相似性推荐

协同过滤（和内容推荐都）是建立在用户有一定的历史行为数据针对用户进行个性化推荐。如果平台没有用户数据积累的话，那么如何实施推荐呢？

这时相似性推荐就有了极大的用途。对于新用户，在他首页进行随机性推荐，根据他在不同场景停留时间等行为，分析停留时间较长的场景，将该场景类似的集合推荐给用户。

在给新用户推荐电影的场景中，相似性推荐算法是如何工作的：新用户A 看了一个导演A导演的由演员B、C、D演的电影1。根据导演和演员获取其他由他们导演或者参演的电影。将电影推荐给新用户。

1. 关联规则推荐

“把啤酒放在尿布旁，有助于提升啤酒销售量”是关联规则推荐的经典案例。

关联规则是数据挖掘中的概念，通过分析数据，找到数据之间的关联。常应用于电商中用来分析购买物品之间的关联性。例如，“购买尿布的用户，有大概率购买啤酒”，这就是一个关联规则。在实际应用中，线上在用户将尿布放入购物车后推荐啤酒，线下将尿布和啤酒放在一起。

如何给买啤酒的人推荐其他的东西：从大量的订单交易信息中找到人们在买啤酒时还买了其他什么东西。在这些东西中，找出被买次数最多的东西。将这些东西推给其他买啤酒的人。

1. 混合推荐

由于各种推荐方法各有优缺点，所以在实际使用中，常通过将多种推荐方式相组合的方法进行混合推荐。其中用到最多的是协同过滤与内容推荐的组合。

其中最为简单的方法就是分别用基于内容推荐的方法和协同过滤推荐方法产生一个推荐预测结果。然后在用某种方式对结果进行组合。这种方法的优点就在于将不同推荐算法结合，使结果更加准确。

在组合方式方面，主要有七种思路：

1. 加权(Weight) : 加权多种推荐技术结果。
2. 变换(Switch) : 根据问题背景和实际情况或要求决定变换采用不同的推荐技术。
3. 混合(Mixed) : 同时采用多种推荐技术给出多种推荐结果，为用户提供参考。
4. 特征组合(Feature Combination) : 组合来自不同推荐数据源的特征被另一种推荐算法所采用。
5. 层叠(Cascade) : 先用一种推荐技术产生一种粗糙的推荐结果，第二种推荐技术在此推荐结果的基础上进一步做出更精确的推荐。
6. 特征扩充(Feature Augmentation) : 将一种技术产生附加的特征信息嵌入另一种推荐技术的特征输入中。
7. 元级别(Meta-Ievel) : 用一种推荐方法产生的模型作为另一种推荐方法的输入。
8. 推荐算法的延伸

平台通过对用户的订单数据，历史行为进行分析，抽象用户的特征形成标签。针对不同标签的用户进行不同的价格定位，优惠补贴，营销策略等。

例如：

平台可以从日志中分析出用户A的历史特征是：

1. 有优惠券也不使用

2）等待30秒没人接单就加价

3）等待60秒没人接单就打专车

可以分析出：用户A是土豪，对接单时间敏感，对价格不敏感。

从日志中分析出用户B的历史行为特征是：

1. 没有优惠券就不下单
2. 宁愿等待30分钟也不加价
3. 从不打专车，也绝不在高峰期(价格会加倍)打车

可以分析出：用户B对价格敏感，以省钱为优先

于是，用户A和用户B同时打车，可能出现：时间敏感的用户A贵、价格敏感的用户B便宜

1. 广告点击率预估(click through rate,CTR)
   1. CTR介绍

谈到CTR，都多多少少有些了解，尤其在互联网广告这块，简而言之，就是给某个网络服务使用者推送一个广告，该广告被点击的概率，这个问题难度简单到街边算命随口告诉你今天适不适合娶亲、适不适合搬迁一样，也可以复杂到拿到各种诸如龟壳、铜钱等等家伙事，在沐浴更衣、净手煴香后，最后一通预测，发现完全扯淡，被人暴打一顿，更有甚者，在以前关系国家危亡、异或争国本这种情况时，也通常会算上一卦，国家的兴衰、。其实CTR和这个一样，以前经常和小伙伴吐槽，其实做机器学习、无论是推荐还是计算广告，都和以前的算命先生没什么差别，做的好的官至国师，不好的吃不了饱饭也是有的。

* 1. CTR包括

主要包括两大部分：离线部分、在线部分，其中离线部分目标主要是训练出可用模型，而在线部分则考虑模型上线后，性能可能随时间而出现下降，如果出现这种情况，可选择使用Online-Learning来在线更新模型。

1. 离线部分：
2. 数据收集：主要收集和业务相关的数据，通常会有专门的同事在app位置进行埋点，拿到业务数据；
3. 预处理：对埋点拿到的业务数据进行去脏去重；
4. 构造数据集：经过预处理的业务数据，构造数据集，在切分训练、测试、验证集时应该合理根据业务逻辑来进行切分；
5. 特征工程：对原始数据进行基本的特征处理，包括去除相关性大的特征，离散变量one-hot，连续特征离散化等等;
6. 模型选择：选择合理的机器学习模型来完成相应工作，原则是先从简入深，先找到baseline，然后逐步优化；
7. 超参选择：利用gridsearch、randomsearch或者hyperopt来进行超参选择，选择在离线数据集中性能最好的超参组合；
8. 在线A/B Test：选择优化过后的模型和原先模型（如baseline）进行A/B Test，若性能有提升则替换原先模型；
9. 在线部分
10. Cache & Logic：设定简单过滤规则，过滤异常数据；
11. 模型更新：当Cache & Logic 收集到合适大小数据时，对模型进行pretrain+finetuning，若在测试集上比原始模型性能高，则更新model server的模型参数；
12. Model Server：接受数据请求，返回预测结果；
13. 算法改进建议
14. 任务分配的右剪枝

课上所说的任务分配只有左剪枝优化，这里提出右剪枝：

rw表示之后的总的最低任务成本，如果rw+当前成本大于mincost就剪枝，用mc[i]表示第i人的任务成本中最低的，rw初始为mc的总和，每次迭代，如果选第i人那么rw就减去mc[i]。

1. 斐波那契数列

课上所用的解决斐波那契数列的办法为递归，即

|  |
| --- |
| int f1(int n)  {  if (n < 1)  {  return 0;  }  else if (n == 1 || n == 2)  {  return 1;  }    return f1(n - 1) + f1(n - 2);  } |

直接递归，利用斐波那契的公式，时间复杂度为O（）

现介绍三种改进措施，降低时间复杂度：

1. 动态规划

斐波那契存在递推关系，因此可以用动态规划求解转移方程即为上述递推关系，边界条件为和。

**优化：**看出求解第n个斐波那契只需要其前两个数，于是无需创建dp数组，降低了复杂度。

**代码如下：**

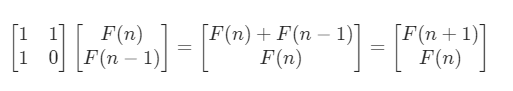
|  |
| --- |
| int fib2(int n)  {  if (n < 1)  {  return 0;  }  if (n == 1 || n == 2)  {  return 1;  }  int res = 1;  int pre = 1;  int tmp = 0;  for (int i = 3; i <= n; i++)  {  tmp = res;  res = res + pre;  pre = tmp;  }  return res;  } |

**复杂度分析：**

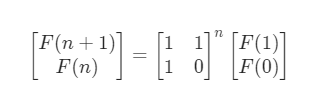
* 时间复杂度为：
* 空间复杂度：

1. 矩阵快速幂

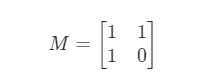
使用矩阵快速幂的方法可以降低时间复杂度。

首先构造递推关系：

之后化简得：



最后得出：

，可以看出，只要能够能计算矩阵 M的n次幂，就能得到的值。

**优化：**如果直接求，时间复杂度为。可以定义矩阵乘法，用矩阵快速幂加速运算。

代码如下：

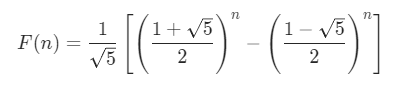
|  |
| --- |
| struct Matrix  {  int mat[2][2];  };    struct Matrix matrixMultiply(struct Matrix \*a, struct Matrix \*b)  {  struct Matrix c;  for (int i = 0; i < 2; i++)  {  for (int j = 0; j < 2; j++)  {  c.mat[i][j] = (\*a).mat[i][0] \* (\*b).mat[0][j] + (\*a).mat[i][1] \* (\*b).mat[1][j];  }  }  return c;  }    struct Matrix matrixPow(struct Matrix a, int n)  {  struct Matrix ret;  ret.mat[0][0] = ret.mat[1][1] = 1;  ret.mat[0][1] = ret.mat[1][0] = 0;  while (n > 0)  {  if (n & 1)  {  ret = matrixMultiply(&ret, &a);  }  n >>= 1;  a = matrixMultiply(&a, &a);  }  return ret;  }    int fib3(int n)  {  if (n < 2)  {  return n;  }  struct Matrix q;  q.mat[0][0] = q.mat[0][1] = q.mat[1][0] = 1;  q.mat[1][1] = 0;  struct Matrix res = matrixPow(q, n - 1);  return res.mat[0][0];  } |

**复杂度分析：**

* 时间复杂度为：
* 空间复杂度：

1. 通项公式

运用数学知识可以看出斐波那契数是齐次线性递推，根据递推方程，可以写出特征方程，求解得出通项公式为



得到通项公式后，就能根据通项公式求得斐波那契的第n项。

代码如下：

|  |
| --- |
| int fib4(int n)  {  double sqrt5 = sqrt(5);  double fibN = pow((1 + sqrt5) / 2, n) - pow((1 - sqrt5) / 2, n);  return round(fibN / sqrt5);  } |

**复杂度分析：**

pow函数的时空复杂度与CPU支持的指令集相关。

1. 判断素数

课上所用的判断素数的方法为蛮力法及其优化，即从2到n开始枚举i，如果没有i被n整除，那么n就是素数，否则不是。优化过后不需从2枚举到n而是枚举到n/2，或者sqrt(n)

代码为：

|  |
| --- |
| bool prime\_1(int n)  {  for (int i = 2; i < n; i++)  {  if (n % i == 0)  return false; // 不是素数  }    return true; // 是素数  }    bool prime\_2(int n)  {  for (int i = 2; i < sqrt(n); i++)  {  if (n % i == 0)  return false; // 不是素数  }  return true; // 是素数  }    bool prime\_3(int n)  {  for (int i = 2; i < n / 2; i++)  {  if (n % i == 0)  return false; // 不是素数  }  return true; // 是素数  } |

**复杂度分析：**

* 时间复杂度为：优化前 优化后
* 空间复杂度：

上述方法，在判断某个小素数时还是有效的，但是到了判断许多大素数时便显得有点慢。先介绍两种方法，来应对这种情况：

1. 欧拉筛法求素数

埃氏筛法的基本思想 ：从2开始，将每个质数的倍数都标记成合数，以达到筛选素数的目的。

欧拉筛法的基本思想 ：在埃氏筛法的基础上，让每个合数只被它的最小质因子筛选一次，以达到不重复的目的。

更高效地素数判断方法应该是将素数预先保存到一个素数表中，当判断一个数是否为素数时，直接查表即可。利用欧拉筛可以快速得到素数表。

代码如下：

|  |
| --- |
| const int N= 1000010;    int primes[N], cnt;  bool st[N];    void get\_prime(int x) {      for(int i = 2; i <= x; i++) {          if(!st[i]) prime[cnt++] = i;          for(int j = 0; prime[j] <= x / i; j++) {              st[prime[j]\*i] = true;              if(i % prime[j] == 0) break；          }      }  } |

**复杂度分析：**

* 时间复杂度为：构建表查询
* 空间复杂度：

1. 素数测试求素数

当遇到特别大的素数时，常规算法耗时都很长。这时需要用素数测试来判断素数。

基本思想：对N进行m次测试，如果有一次测试成功，则回答N是合数；如果m次测试均失败，则回答N是素数。回答N是合数时，答案100%正确；回答N是素数时，答案正确的概率是1 - 2^(-m)。

代码如下：

|  |
| --- |
| //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // Miller\_Rabin 算法进行素数测试  //速度快，而且可以判断 <2^63的数  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  const int S=20;//随机算法判定次数，S越大，判错概率越小      //计算 (a\*b)%c. a,b都是long long的数，直接相乘可能溢出的  // a,b,c <2^63  long long mult\_mod(long long a,long long b,long long c)  {  a%=c;  b%=c;  long long ret=0;  while(b)  {  if(b&1){ret+=a;ret%=c;}  a<<=1;  if(a>=c)a%=c;  b>>=1;  }  return ret;  }        //计算 x^n %c  long long pow\_mod(long long x,long long n,long long mod)//x^n%c  {  if(n==1)return x%mod;  x%=mod;  long long tmp=x;  long long ret=1;  while(n)  {  if(n&1) ret=mult\_mod(ret,tmp,mod);  tmp=mult\_mod(tmp,tmp,mod);  n>>=1;  }  return ret;  }    //以a为基,n-1=x\*2^t a^(n-1)=1(mod n) 验证n是不是合数  //一定是合数返回true,不一定返回false  bool check(long long a,long long n,long long x,long long t)  {  long long ret=pow\_mod(a,x,n);  long long last=ret;  for(int i=1;i<=t;i++)  {  ret=mult\_mod(ret,ret,n);  if(ret==1&&last!=1&&last!=n-1) return true;//合数  last=ret;  }  if(ret!=1) return true;  return false;  }    // Miller\_Rabin()算法素数判定  //是素数返回true.(可能是伪素数，但概率极小)  //合数返回false;    bool Miller\_Rabin(long long n)  {  if(n<2)return false;  if(n==2)return true;  if((n&1)==0) return false;//偶数  long long x=n-1;  long long t=0;  while((x&1)==0){x>>=1;t++;}  for(int i=0;i<S;i++)  {  long long a=rand()%(n-1)+1;//rand()需要stdlib.h头文件  if(check(a,n,x,t))  return false;//合数  }  return true;  }      //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //pollard\_rho 算法进行质因数分解  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  long long factor[100];//质因数分解结果（刚返回时是无序的）  int tol;//质因数的个数。数组小标从0开始    long long gcd(long long a,long long b)  {  if(a==0)return 1;//???????  if(a<0) return gcd(-a,b);  while(b)  {  long long t=a%b;  a=b;  b=t;  }  return a;  }    long long Pollard\_rho(long long x,long long c)  {  long long i=1,k=2;  long long x0=rand()%x;  long long y=x0;  while(1)  {  i++;  x0=(mult\_mod(x0,x0,x)+c)%x;  long long d=gcd(y-x0,x);  if(d!=1&&d!=x) return d;  if(y==x0) return x;  if(i==k){y=x0;k+=k;}  }  }  //对n进行素因子分解  void findfac(long long n)  {  if(Miller\_Rabin(n))//素数  {  factor[tol++]=n;  return;  }  long long p=n;  while(p>=n)p=Pollard\_rho(p,rand()%(n-1)+1);  findfac(p);  findfac(n/p);  }    int main()  {  srand(time(NULL));  long long n;  while(scanf("%I64d",&n)!=EOF)  {  tol=0;  findfac(n);  for(int i=0;i<tol;i++)printf("%I64d ",factor[i]);  printf("**\n**");  if(Miller\_Rabin(n))printf("Yes**\n**");  else printf("No**\n**");  }  return 0;  } |

**复杂度分析：**

* 时间复杂度为：
* 空间复杂度：

1. KMP算法

KMP串匹配算法是Knuth-Morris-Pratt算法的简称，KMP算法的思想就是当模式串的某一位置失配时，能不能将更前面的位置与查找串的该位置对齐，并且直接从该位置开始比较。按照这个思路走，问题叫变成了：当模式串的某一位置失配时要找到一个更前面的位置与查找串的该位置对齐。

代码如下：

|  |
| --- |
| void get\_next(const string & M,vector<int> & next)  {  //按模式串生成vector<int> next(M.size(),-1);  //这里的串的第1个元素下标是0  int i = -1, j = 0;  int M\_len = M.size()-1;  do  {  if((i < 0) || (M[i] == M[j]))  {  i++; j++;  next[j] = i;  }  else  i = next[i];  }while( j < M\_len);  }        int KMP\_match(const string & S,const string & M,int pos)  {  int j = pos, i = 0;//这里的串的第1个元素下标是0  int S\_len = S.size();  int M\_len = M.size();    if((S\_len-pos) < M\_len)  return -1;    vector<int> next(M.size(),-1);    get\_next(M,next);      while (i<M\_len && j<S\_len)  {  if (i < 0 || S[j]==M[i])  {  ++i;  ++j;  }  else i = next[i];//j不变,i跳动  }    if (i == M\_len) return j-i;//匹配成功  else return -1;  } |

**复杂度分析：**

* 时间复杂度为：
* 空间复杂度：

KMP算法的时间复杂度足够低，处理某些问题已经够了，但在某些极端问题上，有时会显得很慢。比如当字符串为aaaaab时，从第一个a到b编号为1-6，如果5号不匹配那么他会用4号与来与主串比较，但是因为4，5号元素都是a，所以这种比较是没有意义的，之后更会与1，2，3号比较，这种循环是十分耗时且无用的。

**下面提出改进措施：**

对next[i]：

这样就避免了重复的比较。

代码如下：

|  |
| --- |
| void get\_next(const string & M,vector<int> & next)  {  int i = -1, j = 0;  int M\_inx = M.size() - 1;    do  {  if((i < 0) || (M[i] == M[j]))  {  i++; j++;  if(M[i] != M[j])  next[j] = i;  else next[j] = next[i];  }  else  i = next[i];  }while( j < M\_inx);  } |

**复杂度分析：**

* 时间复杂度为：
* 空间复杂度：