数据结构入门

南京外国语学校　许昊然

为什么要学数据结构

为什么要学数据结构

* 数据结构是 OI 竞赛中十分重要的考察点

为什么要学数据结构

* 数据结构是 OI 竞赛中十分重要的考察点
* 数据结构题在近年省选、NOI 试题中频繁出现

为什么要学数据结构

* 数据结构是 OI 竞赛中十分重要的考察点
* 数据结构题在近年省选、NOI 试题中频繁出现
* 许多非数据结构题，也需要使用数据结构优化算法

为什么要学数据结构

* 数据结构是 OI 竞赛中十分重要的考察点
* 数据结构题在近年省选、NOI 试题中频繁出现
* 许多非数据结构题，也需要使用数据结构优化算法
* 因此熟练运用数据结构是目标在 NOI 或更高的同学必须掌握的能力

要求同学们预习的知识

课前已经要求同学们预习以下知识：

* 线段树
* ST 表

同学们是否都完成了呢？

好了，下面开始讲课。

从 RMQ 问题谈起

从 RMQ 问题谈起

* 给定一个长度 n 的数列

从 RMQ 问题谈起

* 给定一个长度 n 的数列
* 有很多询问，每次询问数列第 l 个到第 r 个元素的最大值

从 RMQ 问题谈起

* 直接一个元素一个元素地找最大值？每次询问复杂度 $O(n)$，太慢了……

从 RMQ 问题谈起

* 直接一个元素一个元素地找最大值？每次询问复杂度 $O(n)$，太慢了……
* 为什么慢？

从 RMQ 问题谈起

* 直接一个元素一个元素地找最大值？每次询问复杂度 $O(n)$，太慢了……
* 为什么慢？
* 冗余计算太多!

从 RMQ 问题谈起

* 直接一个元素一个元素地找最大值？每次询问复杂度 $O(n)$，太慢了……
* 为什么慢？
* 冗余计算太多!
* 比如有两个询问，分别是 [2,5] 和 [2,7]

从 RMQ 问题谈起

* 直接一个元素一个元素地找最大值？每次询问复杂度 $O(n)$，太慢了……
* 为什么慢？
* 冗余计算太多!
* 比如有两个询问，分别是 [2,5] 和 [2,7]
* 刚才提到的暴力做法中，做 [2,5] 时会看：第 2 个元素不如第 3 个元素大，第 3 个元素比第 4 个元素大，第 3 个元素也比第 5 个元素大，哦，第 3 个元素最大。

从 RMQ 问题谈起

* 但做 [2,7] 时候，如果是人在做，肯定会想，[2,5] 已经看过了，第 3 个元素最大。

从 RMQ 问题谈起

* 但做 [2,7] 时候，如果是人在做，肯定会想，[2,5] 已经看过了，第 3 个元素最大。
* 那询问 [2,7] 只要比较一下第 3 个元素、第 6 个元素、第 7 个元素中最大的就行了。

从 RMQ 问题谈起

* 但做 [2,7] 时候，如果是人在做，肯定会想，[2,5] 已经看过了，第 3 个元素最大。
* 那询问 [2,7] 只要比较一下第 3 个元素、第 6 个元素、第 7 个元素中最大的就行了。
* 可是程序并不能意识到这一点，它还是会依次判断：第 2 个元素不如第 3 个元素大，第 3 个元素比第 4 个元素大……等等

从 RMQ 问题谈起

* 这就是暴力很慢的原因

从 RMQ 问题谈起

* 这就是暴力很慢的原因
* 题目中潜藏的很多性质暴力都没有利用，而是直接抛弃了

从 RMQ 问题谈起

* 这就是暴力很慢的原因
* 题目中潜藏的很多性质暴力都没有利用，而是直接抛弃了
* 比如上述例子中，人脑机智地利用了最大值的“可合并”性质，也就是，用 [2,5] 的最大值和 [6,7] 的最大值能直接拼出 [2,7] 的最大值。

从 RMQ 问题谈起

* 这就是暴力很慢的原因
* 题目中潜藏的很多性质暴力都没有利用，而是直接抛弃了
* 比如上述例子中，人脑机智地利用了最大值的“可合并”性质，也就是，用 [2,5] 的最大值和 [6,7] 的最大值能直接拼出 [2,7] 的最大值。
* 而暴力程序却没有利用这个性质，从而不得不做了大量没有必要的重复计算。

从 RMQ 问题谈起

* 于是，怎么利用这些隐藏性质呢？

从 RMQ 问题谈起

* 于是，怎么利用这些隐藏性质呢？
* 其实预习过线段树或 ST 表的同学们应该已经知道了。

从 RMQ 问题谈起

* 于是，怎么利用这些隐藏性质呢？
* 其实预习过线段树或 ST 表的同学们应该已经知道了。
* 我们可以通过适当地预处理一些区间的最大值，让每个询问区间都可以用很少的预处理过的区间拼出来。

从 RMQ 问题谈起

* 于是，怎么利用这些隐藏性质呢？
* 其实预习过线段树或 ST 表的同学们应该已经知道了。
* 我们可以通过适当地预处理一些区间的最大值，让每个询问区间都可以用很少的预处理过的区间拼出来。
* 请同学们思考，对这个问题，线段树利用了哪些隐藏的性质来完成复杂度的优化的？

从 RMQ 问题谈起

* 于是，怎么利用这些隐藏性质呢？
* 其实预习过线段树或 ST 表的同学们应该已经知道了。
* 我们可以通过适当地预处理一些区间的最大值，让每个询问区间都可以用很少的预处理过的区间拼出来。
* 请同学们思考，对这个问题，线段树利用了哪些隐藏的性质来完成复杂度的优化的？
* ST 表又利用了哪些隐藏的性质来完成复杂度的优化呢？

从 RMQ 问题谈起

我个人认为，OI 竞赛最核心的部分就是观察和利用性质。数据结构当然也不例外。

从 RMQ 问题谈起

我个人认为，OI 竞赛最核心的部分就是观察和利用性质。数据结构当然也不例外。

下面提几个下文需要用到的性质的定义（非严格定义，仅适用于本堂课）：

从 RMQ 问题谈起

我个人认为，OI 竞赛最核心的部分就是观察和利用性质。数据结构当然也不例外。

下面提几个下文需要用到的性质的定义（非严格定义，仅适用于本堂课）：

* 可加性：给出 A 和 B，我们能快速算出 $A + B$ 的值

从 RMQ 问题谈起

我个人认为，OI 竞赛最核心的部分就是观察和利用性质。数据结构当然也不例外。

下面提几个下文需要用到的性质的定义（非严格定义，仅适用于本堂课）：

* 可加性：给出 A 和 B，我们能快速算出 $A + B$ 的值
* 可减性：给出 A 和 B，我们能快速算出 $A - B$ 的值

从 RMQ 问题谈起

我个人认为，OI 竞赛最核心的部分就是观察和利用性质。数据结构当然也不例外。

下面提几个下文需要用到的性质的定义（非严格定义，仅适用于本堂课）：

* 可加性：给出 A 和 B，我们能快速算出 $A+B$ 的值
* 可减性：给出 A 和 B，我们能快速算出 $A-B$ 的值
* 可逆性：给出 A 和 B，始终有 $A+B-B=A$

从 RMQ 问题谈起

我个人认为，OI 竞赛最核心的部分就是观察和利用性质。数据结构当然也不例外。

下面提几个下文需要用到的性质的定义（非严格定义，仅适用于本堂课）：

* 可加性：给出 A 和 B，我们能快速算出 $A+B$ 的值
* 可减性：给出 A 和 B，我们能快速算出 $A-B$ 的值
* 可逆性：给出 A 和 B，始终有 $A+B-B=A$
* 结合律：$(A+B)+C=A+(B+C)$

Part I. 序列上的数据结构

线段树/平衡树所利用的性质

什么样的问题可以用线段树/平衡树去维护。

线段树/平衡树所利用的性质

什么样的问题可以用线段树/平衡树去维护。

* 答案的可加性：支持把两个子问题 A 和 B 的答案快速合并，得到 $A + B$ 的答案

线段树/平衡树所利用的性质

什么样的问题可以用线段树/平衡树去维护。

* 答案的可加性：支持把两个子问题 A 和 B 的答案快速合并，得到 $A + B$ 的答案
* 合并运算满足结合律：$(A + B) + C = A + (B + C)$

线段树/平衡树所利用的性质

如果有区间修改呢？

线段树/平衡树所利用的性质

如果有区间修改呢？

预习过线段树的同学们应该知道，用懒标记就可以了。

线段树/平衡树所利用的性质

如果有区间修改呢？

预习过线段树的同学们应该知道，用懒标记就可以了。

请同学们思考，具有什么样性质的问题可以使用懒标记来支持区间修改？

线段树/平衡树所利用的性质

如果有区间修改呢？

预习过线段树的同学们应该知道，用懒标记就可以了。

请同学们思考，具有什么样性质的问题可以使用懒标记来支持区间修改？

* 懒标记与懒标记具有可加性，且满足结合律
* 懒标记与答案也具有可加性，且满足结合律

例题

请同学们判断下面的问题

能否使用线段树/平衡树来维护

例题 1

有一个序列，要求支持：

* 修改一个元素
* 查询区间和

例题 1

有一个序列，要求支持：

* 修改一个元素
* 查询区间和

可以，“子序列和”这个问题显然具有可加性和结合律

例题 2

有一个序列，要求支持：

* 修改一个元素
* 查询区间前缀最大和

例题 2

有一个序列，要求支持：

* 修改一个元素
* 查询区间前缀最大和

可以，维护和以及前缀最大和这两个值即可支持合并。

例题 3

有一个序列，要求支持：

* 区间整体加一个值
* 查询区间和

例题 3

有一个序列，要求支持：

* 区间整体加一个值
* 查询区间和

可以，以“这个区间被加了多少”作为懒标记，显然懒标记内部、懒标记和答案之间都具有可加性和结合律

例题 4

有一个序列，要求支持：

* 区间整体加一个值
* 查询区间前缀最大和

例题 4

有一个序列，要求支持：

* 区间整体加一个值
* 查询区间前缀最大和

不可以，难以快速合并答案。

例题 5

有一个序列，要求支持：

* 修改一个元素
* 查询区间平方和

例题 5

有一个序列，要求支持：

* 修改一个元素
* 查询区间平方和

可以，“子序列的平方和”显然满足可加性和结合律。

例题 6

有一个序列，要求支持：

* 区间整体加一个值
* 查询区间平方和

例题 6

有一个序列，要求支持：

* 区间整体加一个值
* 查询区间平方和

可以。维护“子序列的平方和”和“子序列和”这两个值，可以发现这时满足懒标记的应用条件。

例题 7

有一个序列，要求支持：

* 修改一个元素
* 查询区间最大公约数

例题 7

有一个序列，要求支持：

* 修改一个元素
* 查询区间最大公约数

可以，“子序列的最大公约数”显然满足可加性和结合律。

例题 8

有一个序列，要求支持：

* 区间整体加一个值
* 查询区间最大公约数

例题 8

有一个序列，要求支持：

* 区间整体加一个值
* 查询区间最大公约数

可以。虽然直接维护的话，懒标记确实是无法快速更新答案的；但如果我们维护邻项作差后的序列，那么区间修改就变成了单点修改，而邻项作差是不会影响最大公约数的结果的。

例题 9

有一个序列，要求支持：

* 查询区间第 k 大数

例题 9

有一个序列，要求支持：

* 查询区间第 k 大数

可以。虽然直接做很困难，但只需先二分答案，问题便被转化为了“一个区间中有多少个数大于某个值”，这个问题只需记录区间中的数排序后的结果即可快速支持查询。

例题 10

有一个序列，要求支持：

* 修改一个元素
* 查询区间第 k 大数

例题 10

有一个序列，要求支持：

* 修改一个元素
* 查询区间第 k 大数

可以。与上题类似，先二分答案，问题便被转化为了“一个区间中有多少个数大于某个值”，我们需要维护区间中的数排序后的结果，并支持修改一个元素。因此用平衡树维护区间中数排序后的结果即可。

例题 11

有一个序列，要求支持：

* 区间整体加一个值
* 查询区间第 k 大数

例题 11

有一个序列，要求支持：

* 区间整体加一个值
* 查询区间第 k 大数

不可以，难以快速合并答案。

例题 12

有一个序列，要求支持：

* 查询区间众数

例题 12

有一个序列，要求支持：

* 查询区间众数

不可以，难以快速合并答案。

没有可加性（无法快速合并答案）怎么办？

没有可加性（无法快速合并答案）怎么办？

分块结构、离线处理等转化。具体请看附件中的文章（也就是我当时在江苏省队时候写的文章《数据结构漫谈》）。

Part II. 树上的数据结构

预备知识

* 和树有关的操作一般都有啥？

预备知识

* 和树有关的操作一般都有啥？
* 首先我们要有一棵树……（一般是有根的）

预备知识

* 和树有关的操作一般都有啥？
* 首先我们要有一棵树……（一般是有根的）
* 操作可以要求查询/修改 某个结点 的一些东西（单点操作）

预备知识

* 和树有关的操作一般都有啥？
* 首先我们要有一棵树……（一般是有根的）
* 操作可以要求查询/修改 某个结点 的一些东西（单点操作）
* 也可能要求查询/修改 某个子树 的一些东西（子树操作）

预备知识

* 和树有关的操作一般都有啥？
* 首先我们要有一棵树……（一般是有根的）
* 操作可以要求查询/修改 某个结点 的一些东西（单点操作）
* 也可能要求查询/修改 某个子树 的一些东西（子树操作）
* 或者查询/修改 某条链 的一些东西（路径操作）

预备知识

* 和树有关的操作一般都有啥？
* 首先我们要有一棵树……（一般是有根的）
* 操作可以要求查询/修改 某个结点 的一些东西（单点操作）
* 也可能要求查询/修改 某个子树 的一些东西（子树操作）
* 或者查询/修改 某条链 的一些东西（路径操作）
* 当然也可以有更复杂一点的操作……

预备知识

* 和树有关的操作一般都有啥？
* 首先我们要有一棵树……（一般是有根的）
* 操作可以要求查询/修改 某个结点 的一些东西（单点操作）
* 也可能要求查询/修改 某个子树 的一些东西（子树操作）
* 或者查询/修改 某条链 的一些东西（路径操作）
* 当然也可以有更复杂一点的操作……
* 比如，在某个位置添加一个叶子……（添加结点）

预备知识

* 和树有关的操作一般都有啥？
* 首先我们要有一棵树……（一般是有根的）
* 操作可以要求查询/修改 某个结点 的一些东西（单点操作）
* 也可能要求查询/修改 某个子树 的一些东西（子树操作）
* 或者查询/修改 某条链 的一些东西（路径操作）
* 当然也可以有更复杂一点的操作……
* 比如，在某个位置添加一个叶子……（添加结点）
* 或者把某个子树砍下来接到另外一个地方去……（砍树）

预备知识

* 和树有关的操作一般都有啥？
* 首先我们要有一棵树……（一般是有根的）
* 操作可以要求查询/修改 某个结点 的一些东西（单点操作）
* 也可能要求查询/修改 某个子树 的一些东西（子树操作）
* 或者查询/修改 某条链 的一些东西（路径操作）
* 当然也可以有更复杂一点的操作……
* 比如，在某个位置添加一个叶子……（添加结点）
* 或者把某个子树砍下来接到另外一个地方去……（砍树）
* 或者把树根结点换成另外一个……（换根）

预备知识

* 和树有关的操作一般都有啥？
* 首先我们要有一棵树……（一般是有根的）
* 操作可以要求查询/修改 某个结点 的一些东西（单点操作）
* 也可能要求查询/修改 某个子树 的一些东西（子树操作）
* 或者查询/修改 某条链 的一些东西（路径操作）
* 当然也可以有更复杂一点的操作……
* 比如，在某个位置添加一个叶子……（添加结点）
* 或者把某个子树砍下来接到另外一个地方去……（砍树）
* 或者把树根结点换成另外一个……（换根）
* 等等等等……

预备知识

为方便叙述，除非特殊说明

* 本部分中提到的查询内容都具有可加性和结合律
* 本部分中提到的整体修改操作都满足懒标记应用条件

静态路径查询

给定一棵树，点上有权值，要求支持：

静态路径查询

给定一棵树，点上有权值，要求支持：

* 路径查询

静态路径查询

给定一棵树，点上有权值，要求支持：

* 路径查询
* 没有修改操作

静态路径查询

给定一棵树，点上有权值，要求支持：

* 路径查询
* 没有修改操作

请同学们开动脑筋，利用之前讲线段树/ST 表时提到思想解决这个问题。

静态路径查询

解答：

任一路径 (a, b) 都可以拆成 (a, p) 与 (p, b) 两段（其中 p 为 a 与 b 的最近公共祖先）

静态路径查询

解答：

任一路径 (a, b) 都可以拆成 (a, p) 与 (p, b) 两段（其中 p 为 a 与 b 的最近公共祖先）

因此我们只考虑路径一个端点是另一端点祖先的情况。

静态路径查询

很容易想到，预处理出一些路径，使得每个路径都可以利用很少的预处理过得路径拼出来。

静态路径查询

很容易想到，预处理出一些路径，使得每个路径都可以利用很少的预处理过得路径拼出来。之前我们已经将询问路径我们不妨处理每个点向上延伸长度为 2^k 的各个路径。这样，询问时，每一步我们都选择尽可能长的预处理过的路径。

静态路径查询

很容易想到，预处理出一些路径，使得每个路径都可以利用很少的预处理过得路径拼出来。之前我们已经将询问路径我们不妨处理每个点向上延伸长度为 2^k 的各个路径。这样，询问时，每一步我们都选择尽可能长的预处理过的路径。

其实相当于每次去掉当前剩余长度二进制表示的最高位。因此每个询问都能用 $O(\log n)$ 个预处理过的区间拼出来。

静态路径查询

很容易想到，预处理出一些路径，使得每个路径都可以利用很少的预处理过得路径拼出来。之前我们已经将询问路径我们不妨处理每个点向上延伸长度为 2^k 的各个路径。这样，询问时，每一步我们都选择尽可能长的预处理过的路径。

其实相当于每次去掉当前剩余长度二进制表示的最高位。因此每个询问都能用 $O(\log n)$ 个预处理过的区间拼出来。

这个十分常见的技巧叫做“倍增祖先”。

子树查询 + 子树修改

给定一棵树，点上有权值，要求支持：

子树查询 + 子树修改

给定一棵树，点上有权值，要求支持：

* 子树查询

子树查询 + 子树修改

给定一棵树，点上有权值，要求支持：

* 子树查询
* 子树修改

子树查询 + 子树修改

为了解决这个问题，我们需要引入 DFS 序 这个东西。

子树查询 + 子树修改

为了解决这个问题，我们需要引入 DFS 序 这个东西。

DFS 序是 dfs 一棵有根树时依次访问的结点 组成的序列。

子树查询 + 子树修改

为了解决这个问题，我们需要引入 DFS 序 这个东西。

DFS 序是 dfs 一棵有根树时依次访问的结点 组成的序列。

DFS 序具有一个非常优秀的性质：

* 任意一个子树的结点集合在 DFS 序中都对应着一个区间

请同学们思考一下这个性质的利用方法。

子树查询 + 子树修改

利用 DFS 序，我们把树上的子树操作转变为了序列上的区间操作。这样就转化成了上个部分讨论的序列操作问题，直接使用数据结构维护这个序列即可。

路径查询 + 单点修改

给定一棵树，点上有权值，要求支持：

路径查询 + 单点修改

给定一棵树，点上有权值，要求支持：

* 路径查询

路径查询 + 单点修改

给定一棵树，点上有权值，要求支持：

* 路径查询
* 单点修改

路径查询 + 单点修改

给定一棵树，点上有权值，要求支持：

* 路径查询
* 单点修改
* 查询、修改内容具有可减性和可逆性

路径查询 + 单点修改

给定一棵树，点上有权值，要求支持：

* 路径查询
* 单点修改
* 查询、修改内容具有可减性和可逆性

这个问题中，倍增祖先算法就没办法使用了。请同学们思考为什么。

路径查询 + 单点修改

给定一棵树，点上有权值，要求支持：

* 路径查询
* 单点修改
* 查询、修改内容具有可减性和可逆性

这个问题中，倍增祖先算法就没办法使用了。请同学们思考为什么。

请同学们开动脑筋，利用之前讲的“子树操作”处理方法的思想解决这道问题。

路径查询 + 单点修改

解答：

查询内容具有可减性和可逆性，这也意味着可以使用类似“部分和”的方法来简化问题。

容易发现，路径 (a, b) 的答案是和 $(root, a) + (root, b) - (root, p) - (root, parent(p))$ 的答案相同的（p 为 a 与 b 的最近公共祖先）。

因此我们只需考虑回答从根出发到某个点的路径即可。

路径查询 + 单点修改

我们不妨考虑一个修改，这个修改能影响到什么样的询问呢？

路径查询 + 单点修改

我们不妨考虑一个修改，这个修改能影响到什么样的询问呢？

显然，只有当询问的端点到根的路径覆盖了这个修改结点时，这个修改才对这个询问有贡献。

路径查询 + 单点修改

我们不妨考虑一个修改，这个修改能影响到什么样的询问呢？

显然，只有当询问的端点到根的路径覆盖了这个修改结点时，这个修改才对这个询问有贡献。

也就是，一个修改对一个询问有贡献，当且仅当这个询问的端点位于这个修改的子树中。

路径查询 + 单点修改

我们不妨考虑一个修改，这个修改能影响到什么样的询问呢？

显然，只有当询问的端点到根的路径覆盖了这个修改结点时，这个修改才对这个询问有贡献。

也就是，一个修改对一个询问有贡献，当且仅当这个询问的端点位于这个修改的子树中。

因此，对于每个单点修改，我们实际执行子树修改；这样每个点所存储的答案实际就是它到根的路径的答案了。

路径查询 + 单点修改

我们不妨考虑一个修改，这个修改能影响到什么样的询问呢？

显然，只有当询问的端点到根的路径覆盖了这个修改结点时，这个修改才对这个询问有贡献。

也就是，一个修改对一个询问有贡献，当且仅当这个询问的端点位于这个修改的子树中。

因此，对于每个单点修改，我们实际执行子树修改；这样每个点所存储的答案实际就是它到根的路径的答案了。

于是，通过计算贡献，我们将“路径查询，单点修改”转化成了“单点查询，子树修改”，也就是前一个问题，直接套用维护 DFS 序的方法解决即可。

单点查询 + 路径修改

给定一棵树，点上有权值，要求支持：

单点查询 + 路径修改

给定一棵树，点上有权值，要求支持：

* 单点查询

单点查询 + 路径修改

给定一棵树，点上有权值，要求支持：

* 单点查询
* 路径修改

单点查询 + 路径修改

给定一棵树，点上有权值，要求支持：

* 单点查询
* 路径修改
* 查询、修改内容具有可减性和可逆性

单点查询 + 路径修改

给定一棵树，点上有权值，要求支持：

* 单点查询
* 路径修改
* 查询、修改内容具有可减性和可逆性

请同学们开动脑筋，利用上个问题“单点修改，路径查询”处理方法的思想解决这个问题。

单点查询 + 路径修改

解答：

解决方法很类似，首先利用可减性和可逆性，每个路径修改都能等价转化为若干从某个点到根的路径修改。

接下来，计算贡献，问题被转化为“单点修改，子树查询”，维护 DFS 序即可解决。

子树查询 + 路径修改

给定一棵树，点上有权值，要求支持：

子树查询 + 路径修改

给定一棵树，点上有权值，要求支持：

* 子树查询

子树查询 + 路径修改

给定一棵树，点上有权值，要求支持：

* 子树查询
* 路径修改

子树查询 + 路径修改

给定一棵树，点上有权值，要求支持：

* 子树查询
* 路径修改
* 查询、修改内容具有可减性和可逆性

子树查询 + 路径修改

给定一棵树，点上有权值，要求支持：

* 子树查询
* 路径修改
* 查询、修改内容具有可减性和可逆性

请同学们开动脑筋，利用之前使用的“计算贡献”思想解决这个问题。

子树查询 + 路径修改

解答：

首先依然把路径操作都等价转化成若干从某个点到根的路径操作。

子树查询 + 路径修改

解答：

首先依然把路径操作都等价转化成若干从某个点到根的路径操作。

我们考虑一个修改对一个查询的贡献。显然，如果修改端点 A 在查询子树 B 内部，那么贡献是 $(depth[A] - depth[B] + 1) * S(A)$，其中 $depth$ 是深度，$S(A)$ 是那个修改操作；而如果 A 不在 B 内部，那么对 B 的答案没有任何贡献。

子树查询 + 路径修改

解答：

首先依然把路径操作都等价转化成若干从某个点到根的路径操作。

我们考虑一个修改对一个查询的贡献。显然，如果修改端点 A 在查询子树 B 内部，那么贡献是 $(depth[A] - depth[B] + 1) * S(A)$，其中 $depth$ 是深度，$S(A)$ 是那个修改操作；而如果 A 不在 B 内部，那么对 B 的答案没有任何贡献。

但我们发现，这个贡献是与 $depth[B]$ 相关的，也就是在没有给定查询之前，这个贡献是多少是未知的。我们无法直接维护。

子树查询 + 路径修改

我们对上面这个式子分离变量：

$$\begin{aligned} & (depth[A] - depth[B] + 1) * S(A) \\ = \quad & (depth[A] + 1) * S(A) - depth[B] * S(A) \end{aligned}$$

子树查询 + 路径修改

我们对上面这个式子分离变量：

$$\begin{aligned} & (depth[A] - depth[B] + 1) * S(A) \\ = \quad & (depth[A] + 1) * S(A) - depth[B] * S(A) \end{aligned}$$

所以，最终的答案是

$$\sum_{A\in\text{子树}B} ((depth[A] + 1) * S(A)) - depth[B] * \sum_{A\in\text{子树}B} S(A)$$

子树查询 + 路径修改

我们发现：

* 前一部分 $(depth[A]+1)*S(A)$ 与 B 无关，贡献是确定的，相当于“子树查询，单点修改”，可以维护。

子树查询 + 路径修改

我们发现：

* 前一部分 $(depth[A]+1)*S(A)$ 与 B 无关，贡献是确定的，相当于“子树查询，单点修改”，可以维护。
* 后一部分中 $S(A)$ 与 B 无关，贡献也是确定的，相当于“子树查询，单点修改”，可以维护。

子树查询 + 路径修改

我们发现：

* 前一部分 $(depth[A]+1)*S(A)$ 与 B 无关，贡献是确定的，相当于“子树查询，单点修改”，可以维护。
* 后一部分中 $S(A)$ 与 B 无关，贡献也是确定的，相当于“子树查询，单点修改”，可以维护。

维护了上述两部分后，只需在查询时将 $depth[B]$ 代入即可算出结果，这个问题便得到了解决。

路径查询 + 子树修改

给定一棵树，点上有权值，要求支持：

路径查询 + 子树修改

给定一棵树，点上有权值，要求支持：

* 路径查询

路径查询 + 子树修改

给定一棵树，点上有权值，要求支持：

* 路径查询
* 子树修改

路径查询 + 子树修改

给定一棵树，点上有权值，要求支持：

* 路径查询
* 子树修改
* 查询、修改内容具有可减性和可逆性

路径查询 + 子树修改

给定一棵树，点上有权值，要求支持：

* 路径查询
* 子树修改
* 查询、修改内容具有可减性和可逆性

请同学们开动脑筋，利用上一题中“分离变量”的技巧解决这道题目。

路径查询 + 路径修改

给定一棵树，点上有权值，要求支持：

路径查询 + 路径修改

给定一棵树，点上有权值，要求支持：

* 路径查询

路径查询 + 路径修改

给定一棵树，点上有权值，要求支持：

* 路径查询
* 路径修改

路径查询 + 路径修改

上文的方法对这个问题没有什么效果，原因是这里修改对查询的贡献不仅与查询的端点有关，而且与修改结点、查询结点的最近公共祖先有关，导致分离变量无法运用。

路径查询 + 路径修改

上文的方法对这个问题没有什么效果，原因是这里修改对查询的贡献不仅与查询的端点有关，而且与修改结点、查询结点的最近公共祖先有关，导致分离变量无法运用。

我们不妨回到最初的思路，试图预处理一些区间，使得每个查询路径都能用很少的预处理过的区间拼出来。

路径查询 + 路径修改

我们需要引入一个名为 树链剖分 的算法来解决这个问题。

路径查询 + 路径修改

我们需要引入一个名为 树链剖分 的算法来解决这个问题。

顾名思义，“树链剖分”就是把“树”分解为一条条的“链”。当然，胡乱分肯定是不太靠谱的。我们要采用比较科学的方法来进行“剖分”。

路径查询 + 路径修改

我们引入“轻边”与“重边”（发音 zhong）的概念。对每个非叶子结点，我们找到其包含最多结点的孩子（如有多个选任意一个）。连接它与这个孩子的边被称为“重边”，连接它与其他孩子的边都是“轻边”。

路径查询 + 路径修改

我们引入“轻边”与“重边”（发音 zhong）的概念。对每个非叶子结点，我们找到其 包含最多结点的孩子（如有多个选任意一个）。连接它与这个孩子的边被称为“重边”，连接它与其他孩子的边都是“轻边”。

完成这个步骤后，树的每条边要么是重边，要么是轻边；而且每个非叶结点 必定有且仅有一条重边连向它的某个孩子。

路径查询 + 路径修改

我们引入“轻边”与“重边”（发音 zhong）的概念。对每个非叶子结点，我们找到其 包含最多结点的孩子（如有多个选任意一个）。连接它与这个孩子的边被称为“重边”，连接它与其他孩子的边都是“轻边”。

完成这个步骤后，树的每条边要么是重边，要么是轻边；而且每个非叶结点 必定有且仅有一条重边连向它的某个孩子。

我们把 重边连接而成的链，称为“重链”。

路径查询 + 路径修改

我们引入“轻边”与“重边”（发音 zhong）的概念。对每个非叶子结点，我们找到其 包含最多结点的孩子（如有多个选任意一个）。连接它与这个孩子的边被称为“重边”，连接它与其他孩子的边都是“轻边”。

完成这个步骤后，树的每条边要么是重边，要么是轻边；而且每个非叶结点 必定有且仅有一条重边连向它的某个孩子。

我们把 重边连接而成的链，称为“重链”。

可以证明，树上任意一条简单路径必定只经过了不超过 $O(\log n)$ 条重链（或重链的一部分）和不超过 $O(\log n)$ 条轻边。

路径查询 + 路径修改

有了上述定理，这个问题就简单了。

路径查询 + 路径修改

有了上述定理，这个问题就简单了。

* 我们已经找到了一种方法，使得任意一条路径都能被用 $O(\log n)$ 个重链的一部分拼出来。

路径查询 + 路径修改

有了上述定理，这个问题就简单了。

* 我们已经找到了一种方法，使得任意一条路径都能被用 $O(\log n)$ 个重链的一部分拼出来。
* 因此，我们用线段树等数据结构维护重链，这样任意一条路径都可以由 $O(\log^2 n)$ 个线段树结点区间拼出来。

路径查询 + 路径修改

有了上述定理，这个问题就简单了。

* 我们已经找到了一种方法，使得任意一条路径都能被用 $O(\log n)$ 个重链的一部分拼出来。
* 因此，我们用线段树等数据结构维护重链，这样任意一条路径都可以由 $O(\log^2 n)$ 个线段树结点区间拼出来。

这个问题得到了解决。

一个神奇的方法

给定一棵树，点上有权值，要求支持：

一个神奇的方法

给定一棵树，点上有权值，要求支持：

* 子树查询

一个神奇的方法

给定一棵树，点上有权值，要求支持：

* 子树查询
* 路径查询

一个神奇的方法

给定一棵树，点上有权值，要求支持：

* 子树查询
* 路径查询
* 子树修改

一个神奇的方法

给定一棵树，点上有权值，要求支持：

* 子树查询
* 路径查询
* 子树修改
* 路径修改

一个神奇的方法

给定一棵树，点上有权值，要求支持：

* 子树查询
* 路径查询
* 子树修改
* 路径修改
* 只保证查询、修改内容满足可加性和结合律，不再有其他可利用性质

一个神奇的方法

给定一棵树，点上有权值，要求支持：

* 子树查询
* 路径查询
* 子树修改
* 路径修改
* 只保证查询、修改内容满足可加性和结合律，不再有其他可利用性质

上面所有方法对这道题目都失效了，但这题其实是可做的。

一个神奇的方法

我们分析一下这题的矛盾所在：

一个神奇的方法

我们分析一下这题的矛盾所在：

* DFS 序能高效处理子树操作，却无法直接高效处理路径操作

一个神奇的方法

我们分析一下这题的矛盾所在：

* DFS 序能高效处理子树操作，却无法直接高效处理路径操作
* 而且因为同时存在路径修改和路径查询，我们也无法通过各种技巧规避路径操作

一个神奇的方法

我们分析一下这题的矛盾所在：

* DFS 序能高效处理子树操作，却无法直接高效处理路径操作
* 而且因为同时存在路径修改和路径查询，我们也无法通过各种技巧规避路径操作
* 而能够高效地处理路径操作的树链剖分，却无法高效处理子树操作

一个神奇的方法

* 为什么 DFS 序无法高效处理路径操作？

一个神奇的方法

* 为什么 DFS 序无法高效处理路径操作？
* 因为一条连续的路径，对应到 DFS 序中不一定是连续的，而且也不可能做到始终连续。

一个神奇的方法

* 为什么 DFS 序无法高效处理路径操作？
* 因为一条连续的路径，对应到 DFS 序中不一定是连续的，而且也不可能做到始终连续。
* 为什么不可能做到始终连续？

一个神奇的方法

* 为什么 DFS 序无法高效处理路径操作？
* 因为一条连续的路径，对应到 DFS 序中不一定是连续的，而且也不可能做到始终连续。
* 为什么不可能做到始终连续？
* 因为如果某个结点有多个孩子，DFS 序显然只能排其中一个孩子在这个结点的下一个位置，这样到其他孩子的路径就不连续了。

一个神奇的方法

* 也就是说，对每个结点，DFS 序能且只能保证每个结点向下的一条路径是连续的。我们付出的代价是和询问路径被分割成的“连续路径”的个数成正比的。

一个神奇的方法

* 也就是说，对每个结点，DFS 序能且只能保证每个结点向下的一条路径是连续的。我们付出的代价是和询问路径被分割成的“连续路径”的个数成正比的。
* 换言之，对每个结点，我们可以在其连向其孩子的边中，选择一条边。然后，我们选中的边组成了很多链，这些链在 DFS 序中是连续的。

一个神奇的方法

* 也就是说，对每个结点，DFS 序能且只能保证每个结点向下的一条路径是连续的。我们付出的代价是和询问路径被分割成的“连续路径”的个数成正比的。
* 换言之，对每个结点，我们可以在其连向其孩子的边中，选择一条边。然后，我们选中的边组成了很多链，这些链在 DFS 序中是连续的。
* 我们的操作代价与路径中包含的“链”的数目成正比。

一个神奇的方法

* 也就是说，对每个结点，DFS 序能且只能保证每个结点向下的一条路径是连续的。我们付出的代价是和询问路径被分割成的“连续路径”的个数成正比的。
* 换言之，对每个结点，我们可以在其连向其孩子的边中，选择一条边。然后，我们选中的边组成了很多链，这些链在 DFS 序中是连续的。
* 我们的操作代价与路径中包含的“链”的数目成正比。
* 想到了什么？这个定义与树链剖分的定义一模一样!

一个神奇的方法

于是，我们依照树链剖分的策略，把“重边”作为我们选择的边。也就是，在求 DFS 序时，最优先 dfs 重边，然后再 dfs 轻边。

一个神奇的方法

于是，我们依照树链剖分的策略，把“重边”作为我们选择的边。也就是，在求 DFS 序时，最优先 dfs 重边，然后再 dfs 轻边。

此时树链剖分中的每条重链，在 DFS 序中都是有序的连续一段。

一个神奇的方法

于是，我们依照树链剖分的策略，把“重边”作为我们选择的边。也就是，在求 DFS 序时，最优先 dfs 重边，然后再 dfs 轻边。

此时树链剖分中的每条重链，在 DFS 序中都是有序的连续一段。

因为由之前的结论，任意一条路径都只会经过不超过 $O(\log n)$ 条重链，因此它们在 DFS 序中只会对应不超过 $O(\log n)$ 个连续区间，也就是 $O(\log^2 n)$ 个线段树结点。

一个神奇的方法

于是，我们依照树链剖分的策略，把“重边”作为我们选择的边。也就是，在求 DFS 序时，最优先 dfs 重边，然后再 dfs 轻边。

此时树链剖分中的每条重链，在 DFS 序中都是有序的连续一段。

因为由之前的结论，任意一条路径都只会经过不超过 $O(\log n)$ 条重链，因此它们在 DFS 序中只会对应不超过 $O(\log n)$ 个连续区间，也就是 $O(\log^2 n)$ 个线段树结点。

通过这个巧妙的处理，我们的 DFS 序也能支持链操作了。这个问题得到了完美解决。

一个神奇的方法

这个方法的强大之处在于，我们把树上的单点操作、子树操作、路径操作全部完美转化成了序列的区间操作问题，而且没有使用除必需的可加性、结合律外的任何性质。

一个神奇的方法

这个方法的强大之处在于，我们把树上的单点操作、子树操作、路径操作全部完美转化成了序列的区间操作问题，而且没有使用除必需的可加性、结合律外的任何性质。

而且，因为序列操作的灵活性，我们还可以完成很多更加困难的问题，比如之前提到的换根、路径权值翻转等，也可以非完美解决“增加/删除叶子”“砍树”等一般只能由动态树完成的操作。

对路径权值翻转操作的支持

线段树无法支持区间翻转的操作，但只需用 splay 树维护 DFS 序，要翻转路径时，先把整条路径提取出来拼接在一起，打上翻转标记，然后再切割成合适的长度放回原位就可以了。

因此，路径权值翻转操作是可以完美支持的。

对换根操作的支持

换根操作对路径操作没有影响，只会影响子树操作。

换根对子树的影响是，指定子树不一定是原树中的子树，也可能对应的是原树抠掉子树后剩余的部分。

到底是哪一种情况可以通过实际树根与操作结点在原树中的关系简单地判断出来，而“原树抠掉子树后剩余部分”实际在 DFS 序中就是对应 DFS 序列挖掉那个子树对应的区间后剩下的“两段”。

因此，换根操作也是可以完美支持的。

对添加/删除叶子操作的支持

添加/删除叶子会使得树的形态改变，相当于往 DFS 序中插入/删除元素。因此我们用 splay 树维护 dfs 序即可。

但为什么说是非完美呢？因为添加/删除叶子会使树的形态改变，进而可能影响到树链剖分方案，使得当前的剖分方案有退化的危险。

当然这个问题是可以勉强解决的，比如每隔若干次操作就重新树链剖分一次，重新建树。这样就比较难卡掉了。

对砍树操作的支持

砍下一棵子树接在其他地方，这个操作实际对应到 DFS 树中，就是把一段区间取出来，插入到序列另外一个地方。

同样，用 splay 树维护 dfs 序就可以支持了。当然也要面对树的形态改变导致当前剖分方案退化的问题。也可以通过隔段时间暴力重建勉强解决。

树形态改变的问题

从上可以看出，我们这种改进的 DFS 序维护方式对树形态不发生改变的问题的支持已经几乎完美了；而在树形态改变的问题中，虽然无法完美支持，但也可以凑活着用。

树形态改变的问题

从上可以看出，我们这种改进的 DFS 序维护方式对树形态不发生改变的问题的支持已经几乎完美了；而在树形态改变的问题中，虽然无法完美支持，但也可以凑活着用。

有没有方法完美支持树形态改变的问题呢？

树形态改变的问题

从上可以看出，我们这种改进的 DFS 序维护方式对树形态不发生改变的问题的支持已经几乎完美了；而在树形态改变的问题中，虽然无法完美支持，但也可以凑活着用。

有没有方法完美支持树形态改变的问题呢？

* 动态树、Top 树等

树形态改变的问题

从上可以看出，我们这种改进的 DFS 序维护方式对树形态不发生改变的问题的支持已经几乎完美了；而在树形态改变的问题中，虽然无法完美支持，但也可以凑活着用。

有没有方法完美支持树形态改变的问题呢？

* 动态树、Top 树等
* 但代码复杂、常数巨大、调试困难

树形态改变的问题

从上可以看出，我们这种改进的 DFS 序维护方式对树形态不发生改变的问题的支持已经几乎完美了；而在树形态改变的问题中，虽然无法完美支持，但也可以凑活着用。

有没有方法完美支持树形态改变的问题呢？

* 动态树、Top 树等
* 但代码复杂、常数巨大、调试困难
* 在目前 OI 比赛中应用较少

Part III. 数据结构在动态规划优化中的应用

数据结构在动态规划优化中的应用

数据结构不仅只出现在纯数据结构题中，在其他题目中，也常常需要数据结构优化算法。

数据结构在动态规划优化中的应用

数据结构不仅只出现在纯数据结构题中，在其他题目中，也常常需要数据结构优化算法。

因为时间原因，我们只简单介绍数据结构优化动态规划的几个经典例子。

例题 1

请优化以下 dp 转移方程（l, r, v 数组均已给出，$l_i \le r_i < i$）：

$$dp_i = max(dp_k \mid l_i \le k \le r_i) + v_i$$

例题 1

请优化以下 dp 转移方程（l, r, v 数组均已给出，$l_i \le r_i < i$）：

$$dp_i = max(dp_k \mid l_i \le k \le r_i) + v_i$$

这个方程中状态 $O(n)$，转移 $O(n)$，我们尝试用数据结构降低转移的复杂度。

例题 1

请优化以下 dp 转移方程（l, r, v 数组均已给出，$l_i \le r_i < i$）：

$$dp_i = max(dp_k \mid l_i \le k \le r_i) + v_i$$

这个方程中状态 $O(n)$，转移 $O(n)$，我们尝试用数据结构降低转移的复杂度。

解决方案很简单，我们考虑用数据结构维护 dp 数组，那么每个新的值都是已经求出的序列中的一个区间的最大值，而加入新求出的元素相当于修改序列。

例题 1

请优化以下 dp 转移方程（l, r, v 数组均已给出，$l_i \le r_i < i$）：

$$dp_i = max(dp_k \mid l_i \le k \le r_i) + v_i$$

这个方程中状态 $O(n)$，转移 $O(n)$，我们尝试用数据结构降低转移的复杂度。

解决方案很简单，我们考虑用数据结构维护 dp 数组，那么每个新的值都是已经求出的序列中的一个区间的最大值，而加入新求出的元素相当于修改序列。

“区间查询，单点修改”，直接用线段树维护即可。

例题 2

有一个长度为 n 的二元组序列 (p_i, q_i)，要求删掉若干元素，使得剩下的序列中，满足对于任意 $i < j$，均有 $q_i \geq p_j$。要求剩下的序列尽可能长。要求低于 $O(n^2)$ 算法。

例 2

首先我们考虑 $O(n^2)$ 的算法怎么做。

例 2

首先我们考虑 $O(n^2)$ 的算法怎么做。

首先这题的阶段性是很明显的，假设我们已经考虑了原序列的前 i 个元素，那么从这 i 个元素中选出的满足条件序列对后面的选择惟一的影响就是“选择的元素中 q_i 的最小值是多少”。

例题 2

因此我们用状态 $dp[i][s]$ 表示当前考虑了原序列的前 i 个元素，其中选择的子序列中最小的 q_i 是 s 时，这个子序列能多长。（我们可以预先将数值离散化掉，这样 s 的规模也是 $O(n)$ 的）

例题 2

因此我们用状态 $dp[i][s]$ 表示当前考虑了原序列的前 i 个元素，其中选择的子序列中最小的 q_i 是 s 时，这个子序列能多长。（我们可以预先将数值离散化掉，这样 s 的规模也是 $O(n)$ 的）

那么转移方程显然是（其中等号代表用右边更新左边最大值）

$$
\begin{aligned}
dp[i][s] &= dp[i-1][s] \\
dp[i][s] &= max(dp[i-1][k] \mid k \geq max(p_i, q_i)) + 1 \text{ 当 } s = q_i \\
dp[i][s] &= dp[i-1][s] + 1 \text{ 当 } p_i \leq s < q_i
\end{aligned}
$$

最终答案是 $max(dp[n][?])$。

这个算法状态 $O(n^2)$，转移 $O(1)$，总复杂度是 $O(n^2)$ 的。

例题 2

* 注意到这个问题中状态数目已经达到了 $O(n^2)$，光是状态数目就已经不可接受了。

例题 2

* 注意到这个问题中状态数目已经达到了 $O(n^2)$，光是状态数目就已经不可接受了。
* 如何优化？

例题 2

* 注意到这个问题中状态数目已经达到了 $O(n^2)$，光是状态数目就已经不可接受了。
* 如何优化？
* 更改状态表示？

例题 2

* 注意到这个问题中状态数目已经达到了 $O(n^2)$，光是状态数目就已经不可接受了。
* 如何优化？
* 更改状态表示？
* 确实是一个常见解决思路，但对这题似乎有难度……

例题 2

我们不妨换一个思路。我们重新观察这个方程。

$$
\begin{aligned}
dp[i][s] &= dp[i-1][s] \\
dp[i][s] &= max(dp[i-1][k] \mid k \geq max(p_i, q_i)) + 1 \text{ 当 } s = q_i \\
dp[i][s] &= dp[i-1][s] + 1 \text{ 当 } p_i \leq s < q_i
\end{aligned}
$$

例题 2

我们不妨换一个思路。我们重新观察这个方程。

$$
\begin{aligned}
dp[i][s] &= dp[i-1][s] \\
dp[i][s] &= max(dp[i-1][k] \mid k \geq max(p_i, q_i)) + 1 \text{ 当 } s = q_i \\
dp[i][s] &= dp[i-1][s] + 1 \text{ 当 } p_i \leq s < q_i
\end{aligned}
$$

我们发现，这个方程的转移是很有规律的。考虑 $dp[i][?]$ 的序列转移到 $dp[i+1][?]$ 后发生了哪些变化。

例题 2

* 转移 $dp[i][s] = dp[i-1][s]$

例题 2

* 转移 $dp[i][s] = dp[i-1][s]$
* 相当于把原序列复制到新序列的位置上，没有发生变化

例题 2

* 转移 $dp[i][s] = dp[i-1][s]$
* 相当于把原序列复制到新序列的位置上，没有发生变化
* 转移 $dp[i][s] = max(dp[i-1][k] \mid k \geq max(p_i, q_i)) + 1$ 当 $s = q_i$

例题 2

* 转移 $dp[i][s] = dp[i-1][s]$
* 相当于把原序列复制到新序列的位置上，没有发生变化
* 转移 $dp[i][s] = max(dp[i-1][k] \mid k \geq max(p_i, q_i)) + 1$ 当 $s = q_i$
* 相当于把新序列的第 q_i 个元素更新成了一个区间最大值

例题 2

* 转移 $dp[i][s] = dp[i-1][s]$
* 相当于把原序列复制到新序列的位置上，没有发生变化
* 转移 $dp[i][s] = max(dp[i-1][k] \mid k \geq max(p_i, q_i)) + 1$ 当 $s = q_i$
* 相当于把新序列的第 q_i 个元素更新成了一个区间最大值
* 转移 $dp[i][s] = dp[i-1][s] + 1$ 当 $p_i \leq s < q_i$

例题 2

* 转移 $dp[i][s] = dp[i-1][s]$
* 相当于把原序列复制到新序列的位置上，没有发生变化
* 转移 $dp[i][s] = max(dp[i-1][k] \mid k \geq max(p_i, q_i)) + 1$ 当 $s = q_i$
* 相当于把新序列的第 q_i 个元素更新成了一个区间最大值
* 转移 $dp[i][s] = dp[i-1][s] + 1$ 当 $p_i \leq s < q_i$
* 相当于对新序列的一个区间中的元素加 1

例题 2

想到了什么？这些操作都是区间操作!

例题 2

想到了什么？这些操作都是区间操作！

我们直接用线段树维护 $dp[i]$，然后从 $dp[i]$ 到 $dp[i+1]$ 的转移可以通过几次区间操作完成。

例题 2

想到了什么？这些操作都是区间操作！

我们直接用线段树维护 $dp[i]$，然后从 $dp[i]$ 到 $dp[i+1]$ 的转移可以通过几次区间操作完成。

通过整体转移的技巧，我们成功把时间复杂度降低到了 $O(n \log n)$。

例题 3

有一个长度为 n 的正整数序列 a，你要把它分成若干段，使得每段的和不超过一个给定的值 S，并且每段的最大的数的和尽可能小。

要求低于 $O(n^2)$ 算法。

例题 3

同样，我们先考虑 $O(n^2)$ 算法怎么做。

例题 3

同样，我们先考虑 $O(n^2)$ 算法怎么做。

这题也具有很明显的阶段性，我们不妨使用 dp_i 表示考虑到了原序列第 i 个数，且第 i 个数恰好是一段的结尾时，最小的答案是多少。

例题 3

同样，我们先考虑 $O(n^2)$ 算法怎么做。

这题也具有很明显的阶段性，我们不妨使用 dp_i 表示考虑到了原序列第 i 个数，且第 i 个数恰好是一段的结尾时，最小的答案是多少。

则显然有转移方程：

$$dp_i = min(dp_j + max(a_k \mid j < k \leq i)) \text{ 其中 } 0 \leq j < i$$

这个算法状态 $O(n)$，转移 $O(n)$，总复杂度是 $O(n^2)$ 的。

例题 3

我们考虑如何优化这个算法。

我们发现，这题难点在于，随着 i 往前推，$dp_j + max(a_k \mid j < k \leq i)$ 这个式子的值一直在变化，不方便我们用数据结构处理。（其实还是可以做的，只是会有点麻烦）

例题 3

这时，我们需要观察隐藏的性质，来方便我们的处理。

例题 3

这时，我们需要观察隐藏的性质，来方便我们的处理。

考虑固定 i，哪些决策 j 可能成为最优决策？

例题 3

这时，我们需要观察隐藏的性质，来方便我们的处理。

考虑固定 i，哪些决策 j 可能成为最优决策？

首先我们可以证明 dp 数组的不减性：

如果存在 $i_1 < i_2$ 使得 $dp_{i_1} > dp_{i_2}$，那么只需将 i_2 的方案直接删去 $(i_1, i_2]$ 这段元素，我们便得到了 i_1 的一组合法方案，而且代价不会增加，从而得到了 dp_{i_1} 更优的解。

例题 3

利用 dp 数组不减这条性质，我们可以发现更有用的性质：

> 如果对 $j_1 < j_2$，有 $max(a_k \mid j_1 < k \leq i) = max(a_k \mid j_2 < k \leq i)$，那么因为 $dp_{j_1} \leq dp_{j_2}$，最终决策 j_1 的答案一定不会劣于决策 j_2。

例题 3

利用 dp 数组不减这条性质，我们可以发现更有用的性质：

> 如果对 $j_1 < j_2$，有 $max(a_k \mid j_1 < k \leq i) = max(a_k \mid j_2 < k \leq i)$，
> 那么因为 $dp_{j_1} \leq dp_{j_2}$，最终决策 j_1 的答案一定不会劣于决策 j_2。

也就是说，我们不妨设 $f(j) = max(a_k \mid j < k \leq i)$，那么最优决策 j 一定有 $f(j-1) \neq f(j)$（当然前提是 $j-1$ 也是可行决策）。

例题 3

利用 dp 数组不减这条性质，我们可以发现更有用的性质：

> 如果对 $j_1 < j_2$，有 $max(a_k \mid j_1 < k \leq i) = max(a_k \mid j_2 < k \leq i)$，
> 那么因为 $dp_{j_1} \leq dp_{j_2}$，最终决策 j_1 的答案一定不会劣于决策 j_2。

也就是说，我们不妨设 $f(j) = max(a_k \mid j < k \leq i)$，那么最优决策 j 一定有 $f(j-1) \neq f(j)$（当然前提是 $j-1$ 也是可行决策）。
换言之，f 函数实际是一个下降的阶梯形，最优决策一定在这个阶梯形“台阶”的顶点上。

例题 3

我们考虑随着 i 的增加，f 函数的阶梯形“顶点”的变化。

例题 3

我们考虑随着 i 的增加，f 函数的阶梯形“顶点”的变化。

* 首先，因为新增加了一个元素，最老的决策受到“每段和不能超过 S”的限制，可能不再可行了。我们删掉不再可行的最老决策。

例题 3

我们考虑随着 i 的增加，f 函数的阶梯形“顶点”的变化。

* 首先，因为新增加了一个元素，最老的决策受到“每段和不能超过 S”的限制，可能不再可行了。我们删掉不再可行的最老决策。
* 新增一个元素后，观察后可以发现，所有值低于这个元素的“顶点”都不再存在了，将它们删掉。

例题 3

我们考虑随着 i 的增加，f 函数的阶梯形“顶点”的变化。

* 首先，因为新增加了一个元素，最老的决策受到“每段和不能超过 S”的限制，可能不再可行了。我们删掉不再可行的最老决策。
* 新增一个元素后，观察后可以发现，所有值低于这个元素的“顶点”都不再存在了，将它们删掉。
* 最后新元素一定自成一个顶点，将它加进来。

例题 3

我们考虑随着 i 的增加，f 函数的阶梯形“顶点”的变化。

* 首先，因为新增加了一个元素，最老的决策受到“每段和不能超过 S”的限制，可能不再可行了。我们删掉不再可行的最老决策。
* 新增一个元素后，观察后可以发现，所有值低于这个元素的“顶点”都不再存在了，将它们删掉。
* 最后新元素一定自成一个顶点，将它加进来。

我们发现，f 函数是单调的，因此我们维护的顶点也是单调的。因此所有的操作都是在序列开头或结尾进行，我们用一个队列就可以维护出当前可行决策了。

例题 3

因为每个决策最多入一次队出一次队，而且在决策存在于队列中期间，这个决策的答案显然是一直不变的。

例题 3

因为每个决策最多入一次队出一次队，而且在决策存在于队列中期间，这个决策的答案显然是一直不变的。

我们直接用平衡树维护当前所有可行决策的答案，在队列操作时同时更新平衡树，即可快速查找当前状态的最优决策。

例题 3

因为每个决策最多入一次队出一次队，而且在决策存在于队列中期间，这个决策的答案显然是一直不变的。

我们直接用平衡树维护当前所有可行决策的答案，在队列操作时同时更新平衡树，即可快速查找当前状态的最优决策。

我们通过观察隐藏性质，排除无用决策，为数据结构维护创造了条件，最终把复杂度降低至 $O(n \log n)$。

结束语

所以说到底，OI 考察的核心还是挖掘和利用性质的能力。

结束语

所以说到底，OI 考察的核心还是挖掘和利用性质的能力。

但是，除非你是天才，我觉得在考场上现场推性质想出 SAP 算法或者 dinic 算法之类的还是很不现实的。这也是我们必须坚持训练总结的原因。

结束语

所以说到底，OI 考察的核心还是挖掘和利用性质的能力。
但是，除非你是天才，我觉得在考场上现场推性质想出 SAP 算法或者 dinic 算法之类的还是很不现实的。这也是我们必须坚持训练总结的原因。

很多数据结构通过花代价在预处理上，来换取查询的速度，我们训练的目的也是一样的：

结束语

所以说到底，OI 考察的核心还是挖掘和利用性质的能力。
但是，除非你是天才，我觉得在考场上现场推性质想出 SAP 算法或者 dinic 算法之类的还是很不现实的。这也是我们必须坚持训练总结的原因。

很多数据结构通过花代价在预处理上，来换取查询的速度，我们训练的目的也是一样的：
你平时多预处理一点东西，到考试时，题目就能更快地用你在训练中掌握的技巧解出来了……

结束语

上文中讲的内容，只是博大精深的数据结构中很小的一部分，因为时间原因和自身水平的不足，我并没有能力把全部东西都讲出来，而且很多东西也无法讲出来，只能通过大量训练总结获得。

结束语

上文中讲的内容，只是博大精深的数据结构中很小的一部分，因为时间原因和自身水平的不足，我并没有能力把全部东西都讲出来，而且很多东西也无法讲出来，只能通过大量训练总结获得。

因此当李曙老师让我来讲课时，我一开始并不想来，因为我觉得不说三个小时，就算三十个小时也不一定讲得完数据结构的所有内容；就算填鸭式地讲完了，又如何做到熟练应用；题目无穷无尽，估计只要变个形，不会的还是不会。

结束语

但后来想了一下，或许我唯一能做的工作，就是给出一个学习导引，让同学们在学习过程中有目标，少走弯路。这也是我这堂课的目的。

结束语

但后来想了一下，或许我唯一能做的工作，就是给出一个学习导引，让同学们在学习过程中有目标，少走弯路。这也是我这堂课的目的。

本课件后面就是我做的一份简单的数据结构方面学习导引，希望同学们逐步完成。当然，只完成里面列出的几道题目自然是远远不够的。希望同学们努力训练、认真总结，相信努力一定能带来收获。

结束语

但后来想了一下，或许我唯一能做的工作，就是给出一个学习导引，让同学们在学习过程中有目标，少走弯路。这也是我这堂课的目的。

本课件后面就是我做的一份简单的数据结构方面学习导引，希望同学们逐步完成。当然，只完成里面列出的几道题目自然是远远不够的。希望同学们努力训练、认真总结，相信努力一定能带来收获。

OI 竞赛与众不同，光指望听课是会完蛋的，靠自己的努力才是王道。

结束语

但后来想了一下，或许我唯一能做的工作，就是给出一个学习导引，让同学们在学习过程中有目标，少走弯路。这也是我这堂课的目的。

本课件后面就是我做的一份简单的数据结构方面学习导引，希望同学们逐步完成。当然，只完成里面列出的几道题目自然是远远不够的。希望同学们努力训练、认真总结，相信努力一定能带来收获。

OI 竞赛与众不同，光指望听课是会完蛋的，靠自己的努力才是王道。学长只能帮你到这了。

结束语

但后来想了一下，或许我唯一能做的工作，就是给出一个学习导引，让同学们在学习过程中有目标，少走弯路。这也是我这堂课的目的。

本课件后面就是我做的一份简单的数据结构方面学习导引，希望同学们逐步完成。当然，只完成里面列出的几道题目自然是远远不够的。希望同学们努力训练、认真总结，相信努力一定能带来收获。

OI 竞赛与众不同，光指望听课是会完蛋的，靠自己的努力才是王道。学长只能帮你到这了。

谢谢大家

有疑问可以与我课后交流

常用 OJ 网址

* POJ（非常适合新手，也有不少好题，但无意义乱搞题实在太多，不推荐作为后期主要训练材料）：poj.org
* Codeforces（题目难易适中，强烈推荐）：www.codeforces.com
* BZOJ（大量省选原题，推荐）：www.lydsy.com/JudgeOnline
* Topcoder（思维训练的极好材料）：用客户端
* Project Euler（偏数学，有兴趣可以适度玩玩）：projecteuler.net
* 还有 SPOJ、SGU、URAL 等很著名的 OJ，但我没刷过，不作评论

课后要求同学们完成的内容

在线算法与离线算法：

* 不了解在线算法与离线算法区别的同学请用搜索引擎学习之
* 使用离线算法往往能简化题目算法。本课件中提到的题目都是用在线算法解决的。请思考其中有没有题目使用离线算法能得到更简单的解决方案。
* 注: 题目可能会用数据加密/交互式等方式强迫选手采用在线算法，因此在线算法还是有必要掌握的……

课后要求同学们完成的内容

与线段树相关的内容：

* GSS1: www.spoj.pl/problems/GSS1
* GSS3: www.spoj.pl/problems/GSS3
* GSS5: www.spoj.pl/problems/GSS5
* NOI2007 项链工厂
* Transformation: pkuteam.openjudge.cn/2013ioitest/C/
* (附加题) GSS2: www.spoj.pl/problems/GSS2

课后要求同学们完成的内容

与平衡树相关的内容：

* 阅读文章《The Magical Splay》作者 sqybi
* 阅读一篇介绍 treap 的文章
* GSS6: www.spoj.pl/problems/GSS6
* NOI2005 维护数列
* NOI2007 货币兑换

课后要求同学们完成的内容

与分块相关的内容

* 阅读论文《分块方法的应用》作者王子昱
* 阅读论文《浅谈分块思想在一类数据处理问题中的应用》作者罗剑桥
* 分块可以解决本课件第一部分中未能用线段树/平衡树解决的例题，请思考解决方案。

课后要求同学们完成的内容

与树的维护相关的内容：

* 有一棵有根树，点上有权，修改操作在单点修改、链修改、子树修改中任选一个; 查询操作在查询单点权值、链权值和、子树权值和中任选一个；组合起来可以得到 9 道题。除了链修改链查询需要树链剖分外，其他 8 个问题都可以简单的维护。请思考各个问题做法。
* UESTC 1653: acm.uestc.edu.cn/problem.php?pid=1653
* 以上问题建议不要使用万能方法做，以锻炼思维。

课后要求同学们完成的内容

与树链剖分、树分治相关的内容：

* 阅读论文《分治算法在树的路径问题中的应用》作者漆子超
* POJ 1741: poj.org/problem?id=1741
* QTREE: www.spoj.pl/problems/QTREE
* (附加题) QTREE4: www.spoj.pl/problems/QTREE4

课后要求同学们完成的内容

与字符串相关的内容：

* 善用搜索引擎，学习掌握 KMP 算法、AC 自动机、字典序 hash 算法、newlcp、后缀数组等字符串数据结构
* 阅读论文《后缀数组—处理字符串的有力工具》作者罗穗骞
* 字典序 hash 算法、后缀数组都是十分强大的利器，请务必彻底理解、熟练应用

启发式合并：

* 启发式合并是简单但很有用的技巧，请使用搜索引擎学习之
* SDOI2013 森林:
  www.lydsy.com/JudgeOnline/problem.php?id=3123

对时间分治及其扩展、整体二分：

* 阅读论文《从〈Cash〉谈一类分治算法的应用》作者陈丹琦
* 阅读论文《浅谈数据结构题的几个非经典解法》作者许昊然

课后要求同学们完成的内容

与可持久化数据结构相关的内容：

* 阅读文章《挖掘 treap 的潜力》作者范浩强
* 阅读论文《可持久化数据结构研究》作者陈立杰
* 思考本课件提到的题目中，有没有题目可以利用可持久化数据结构得到更简单的解决方案。