

基础内容简介 (DRAFT)

GUANGWEI ZHANG(AUGPATH) NONGYU DI(BLUESKY007)

Zhengzhou No. 1 Middle School, China University of Geosciences

Website

GitHub

Zhengzhou No. 1 Middle School, Lanzhou University

Website

GitHub

`gwzhang@cug.edu.cn, diny20@lzu.edu.cn`

目录

| | | |
|-----|-------------|----|
| I | C 语言回顾 | 1 |
| II | 递归问题 | 2 |
| 1 | 问题的简化和递归的过程 | 2 |
| 2 | 递归的结构 | 3 |
| III | 一些有趣的思想 | 8 |
| 3 | 二分法 | 8 |
| 4 | 前缀和与差分 | 10 |
| 5 | 贪心算法 | 12 |
| 6 | 倍增 | 13 |
| 7 | 更多的练习与思考 | 13 |
| 8 | 并查集简介 | 14 |
| IV | 树与图 | 15 |
| 9 | 图的关键要素和存储 | 15 |
| 10 | 常见图算法 | 17 |
| V | 简单计数问题 | 21 |
| 11 | 数学基础 | 21 |
| 12 | 计数原理 | 23 |

| | |
|-----------------------|-----------|
| VI 动态规划简介 | 31 |
| 13 初步的问题 | 31 |
| 14 背包问题 | 35 |
| 15 关于区间的问题 | 38 |
| 16 树形 DP | 40 |
| VII 数论简介 | 44 |
| 17 整数, Euclid 算法 | 44 |
| 18 同余 | 51 |
| VIII 组合数学与概率简介 | 56 |
| 19 二项式系数 | 56 |
| 20 容斥原理 | 58 |
| 21 概率问题简介 | 59 |
| IX 从树状数组到线段树 | 61 |
| 22 树状数组简介 | 61 |
| 23 线段树简介 | 64 |

C 语言回顾

PART

I


有了计算机, 我们可以把很多重复的工作交给计算机完成. 这样, 人们就可以把更重要的经历放在更主要的事情上面去了. 其中, 程序设计语言担当了我们人类世界与机器世界沟通的桥梁, 我们只有通过程序设计语言 (如 C 语言), 计算机才会按照我们希望的方法工作. 当然, 我们对于计算机的期望很多时候是失败的这时候, 我们只有通过一些外部工具, 来证明或者否定我们对于计算机内部一些事情工作的原理.

Axiom 1 机器永远是对的.

TBD: 一个简单的参考, 列举常用工具而不用关心其逻辑实现. 假设已经十分熟悉了.

模拟
介绍

程序执行的状态

 P3592. `◇C++P3592.cpp`

闲聊与练习

核心指导原则 Don't Panic. (不要慌)

——The Hitchhiker's Guide to the Galaxy

如果你还没有入门, 仍然感到恐惧, 请记住: 坚持住, 进入未知领域, 从简单的、能理解的东西试起, 投入时间, 就有收获.

掉在队伍之后的同学, 即便是仅有一定的编程基础, 努力过的同学也一定能通过 (Yes!)

— 蒋炎岩, 在南京大学操作系统课前的提示

递归问题

PART

II

SECTION 1

问题的简化和递归的过程

我们常说：“大事化小，小事化了”。比如，你在做数学计算的 $3+2+4+5$ 这个表达式的时候，你可能会自动先计算 $3+2=5$ ，然后再继续计算 $5+4+5$ 。这样一来，我们距离结果就更进一步了。也就是问题变得更“小”了，或者更“容易”解决了。有些时候，我们甚至允许把整个过程用抽象的方法盖住了。比如，你做了一个摄氏度转华氏度的转换器，你可以用一个函数把它抽象，这样一来，下次使用的时候就直接调用就行了。

我们来看一些我们如何简化问题的一些例子：

Example 要计算一个正整数 n 的阶乘，如果它不是 1 或者 0，那么计算 $n * (n-1)$ 的阶乘。用数学的语言来写就是 $f(x) = \begin{cases} x \cdot f(x-1) & x > 1 \\ 1 & x = 1 \end{cases}$ 这就意味着，我们每一次计算的阶乘都比原来的更靠近答案。如果我们实际写一下 $f(5)$ ，我们就会有如下过程（为了方便起见，我们使用 $f(x)$ 表示 x 的阶乘）：

$$f(5) = 5 \times f(4) = 5 \times 4 \times f(3) = 5 \times 4 \times 3 \times f(2) = 5 \times 4 \times 3 \times 2 \times f(1) = 5 \times 4 \times 3 \times 2 \times 1.$$

Example 要学习知识，首先要认真理解课本的内容，然后自己进行思考，最后对于一些问题进行提问。

实际上，有一类问题它比较特殊。你会发现，如果能够把小问题解决好了，那么原来的大问题就自然而然地解决好了。这种情形，我们一般认为是递归的问题。

Definition 1 (递归) 递归的问题是这样解决的：

- 如果给定的问题大小可以直接解决，那么就直接解决了；
- 否则，把它转化为这个问题的更简单（通常会更小）的问题


一开始，这样自己提及自己的内容的确让人困惑。但是，有一个有趣的方法，就是假想有一个小精灵帮助你解决问题 - 我喜欢称他为递归精灵。你唯一的任务是简化原来的问题，或者在不必要或不可能简化的情况下直接解决它。递归精灵将使用与你无关的方法为你解决所有更简单的子问题。

这样说来确实很困惑。但是我们来看下面的几个例子：

Example 归并排序：要排序一系列数，我们可以将待排序的序列分成两个子序列，然后分别对这两个子序列进行递归排序，最后将两个有序的子序列合并成一个有序的序列。¹ 这时候，我们将问题分解成若干个相同或相似的相似的小问题来解决，然后再将子问题的解合

¹ 在合并的过程中，我们还可以计算逆序对。

并起来, 得到原问题的解. `◇C++mg-sort.cpp`

 **P1908 逆序对**. 我们可以借鉴快速排序的思想的合并过程中作为合并的时候统计. 如果左边有 m 个逆序对, 右边有 n 个逆序对, 那么就会多出 $mid - i + 1$ 个. 见代码 `◇C++inversion-pair.cpp`.

Example | TBD: Hanoi 塔问题

从这里开始, 我们对于程序的执行的理解似乎就感觉有点模糊了. 不过我们总是可以使用正确的工具来让我们了解更多. 具体地, 我们可以使用调试功能. 调试器可以帮助我们窥探程序现在在执行哪一行, 执行的内容是什么. 以及执行到这一步里面的变量有什么, 是什么值. 我们可以使用 `gdb` 来解答这个问题.

我们可以把每一次这样的函数调用想象是一个状态. 所谓状态, 就是相当于给这时候程序里面的内容拍了个照. 研究状态是如何变化的会让我们思路更加清晰. 我们首先从 Hanoi 塔开始看起:

TBD

SECTION 2

递归的结构

引用
重要的操作

可能发生过这样的情形: 假设你是一个课代表, 你希望去某一个老师那里抱作业. 但是你不知道这位老师住在哪. 现在, 有同学告诉你老师在 309 办公室. 这下, 你就可以使用知道的“309”来找到老师了.

看似这是一个不起眼的例子, 实际上, 这包含着计算机科学里面比较重要的内容: 使用一种间接的方式来获得我们要的东西.


链表
单向

链表是它由一系列节点组成, 每个节点包含两部分: 数据 (存储我们需要的信息) 和指针 (指向下一个节点的地址). 链表中的节点可以分散存储在内存中, 不像数组那样需要一段连续的内存空间. 这就可以方便地在中间插入与删除元素. 但是我们的随机查找的功能就不能很快地完成了. 因为它们在内存里面不是连续的.

链表可以被看作是一个递归结构, 即链表的每个节点都可以看作是一个小型的链表. 每个节点包含数据和指向下一个节点的指针, 而下一个节点又包含数据和指向更下一个节点的指针, 以此类推. 原来即使在一些结构里面, 递归的问题也是大有帮助的.

 **UVA11998 Broken Keyboard**. TBD

链表
双向

当然有些问题我们还可以记录双向的信息, 也就是它的上一个和下一个. 这里有一个经典的问题. 我们可能需要很长时间才能够把它调试正确.  **UVA12657 移动盒子**.

树状结构
二叉树

我们窗外应该就有很多树. 我们现在来看树状的结构.

二叉树是一种特殊的树结构, 其中每个节点最多有两个子节点, 通常称为左子节点和右子节点. 这个定义是递归的, 因为二叉树的每个子树也是一个二叉树.

具体地说, 二叉树可以为空, 也就是没有节点的情况, 这被称为空二叉树或空树. 如果二叉树不为空, 则它由一个根节点组成, 以及两个分别为根节点的左子树和右子树的二叉树.

递归定义的核心是，二叉树的每个子树仍然是一个二叉树，它们也遵循相同的定义：最多有两个子节点，每个节点都可以有自己的左子节点和右子节点，或者为空。这种递归定义使得我们可以在处理二叉树时使用递归的思想，对于每个节点，我们可以递归地处理它的左子树和右子树。递归在二叉树的许多问题中都是非常常见和有效的解决方法。

一个问题是，如何储存二叉树呢？一个想法是，我们假设考虑一个最满的二叉树，应该是如下图所示的：

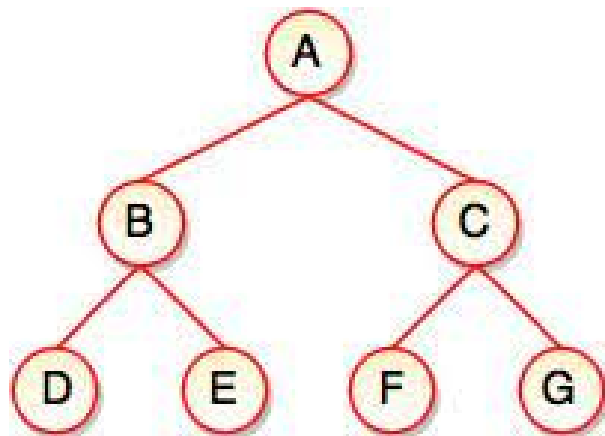


Fig. Full Binary Tree

自然，我们就可以为他们天然指定一个序号。我们采用这样的方式：对于一个节点编号为 o ，左孩子我们可以使用 $o \times 2$ 来表示，右孩子我们可以使用 $o \times 2 + 1$ 来表示。这样子，我们就可以来储存这棵树了。

但问题又来了：我们现在是按照满的样子去精简的。假设我们这个是一条链，也就是层数很深，但是节点数很稀疏，这时候如何是好？这时候，我们用到这样的一个技术了：我们主要维护一个新节点的“池子”，里面的每一个表示一个新节点的编号。每次我们要新建一个节点的话，就可以通过让这个池子的数据加 1 的方式。这时候，我们可以使用编号的方式去更新与原来的关系。实际上，用一个“池子”这样的思路其实是使用了数组模拟指针的思想。现在使用指针可能会造成一些误解，所以先使用数组模拟的指针做大致了解。

TBD: 添加数组模拟指针的代码...

既然树是递归的结构，那么，自然我们更希望使用递归的方法去处理它。不过，因为它有两个子节点，我们才感觉它和一般的线性的数据有一些不同。

一个递归函数，当然可能要决定什么时候递归，什么时候对我们当前的节点做操作。时机是很重要的。

前序遍历的过程是，先输出当前的节点，再递归地前序遍历左边的孩子，再递归地前序遍历右边的孩子。我们对于样例的树模拟一下。

```
void pre_order(int o){
    if(o does not exist) return;
    print(o);
    pre_order(o's left son);
}
```

```

    pre_order(o's right son);
}

```

二叉树的三
种遍历模式
中序遍历

中序遍历的过程是, 先递归地中序遍历左边的孩子, 再输出根节点, 再递归地遍历右边的孩子.

```

void in_order(int o){
    if(o does not exist) return;
    in_order(o's left son);
    print(o);
    in_order(o's right son);
}

```

二叉树的三
种遍历模式
后序遍历

后序遍历的过程是, 先递归地中序遍历左边的孩子, 再递归地遍历右边的孩子, 最后输出根节点.

```

void in_order(int o){
    if(o does not exist) return;
    post_order(o's left son);
    post_order(o's right son);
    print(o);
}

```

这三个内容看起来就是一些奇怪的交换顺序, 实际上, 我们有很多的有趣的性质可以从中观察出来. 比如, 前序遍历的一个内容总是根. 那么, 能不能通过前序遍历的序列和中序遍历的序列还原整个二叉树呢? 其实是可以的. 既然前序遍历的一个内容总是根, 中序遍历只要找到这个根在哪就可以了. 中间的就是子树. 子树也可以按照同样的方法做.

🔗**P1030 求先序排列.** 这个问题只是变了一下形式, 后序遍历了. 但是后序遍历根节点在最后输出. 和上面的讨论是一致的.

我们之所以能够按照这种方法遍历, 说到底还是用好了递归的结构定义. 下面我们来看一看有什么有趣的遍历方法.

🔗**UVA10562 Underdraw the trees.** 题目大意: 你的任务是把多叉树写成括号的表示法. 每个节点处了 “-”, “|”, “ ”(空格) 用其他字符表示, 每个非叶子节点下方总会有一个 “|” 字符, 下方是一排 “-” 字符, 恰好覆盖在所有的子节点上面. 单独的一行 “#” 作为结束标记.

我们可以定义函数 `dfs(r,c)` 表示 `r` 行 `c` 列开始的内容. 下面有子树的条件是下一行的这一列有 `|` 记号 (注意不要越界). 然后我们就可以寻找 `-` 的左边界, 顺着 `-` 走, 一旦发现下面有字符就继续递归下去.

刚刚的问题甚至不是二叉树, 但是我们运用我们的方法照样可以继续下去.

🔗**UVA297 Quadtrees.** 这是一个四叉树的问题. 但是解决问题的思路也是类似的. 这时候我们把两个内容都画出来就好了. 这样子模拟一遍就完成了.

🔗**UVA806 Spatial Structures.**

接下来我们来看求最短路的一些方法. 除了一路走到底, 我们能不能边走边看呢? 我们可以使用 BFS 的方法. 具体而言, 我们的策略如下:


```

queue=[起点]
while(queue 不是空的){
    node = 队列的第一个元素;
    输出 node;
    把所有与 node 能够到达的没有访问过的边放进来;
}

```

我们来看一个迷宫游戏, 并且在上面运用一下我们的 BFS 技术. TBD.

我们发现我们搜到的结果其实就是一棵树.

其实, 我们刚刚发现的树, 有一些类似的妙用. 不过我们要加上允许环的形成. 也就是, 现在它是一些点和一些边的集合. 它甚至可以帮助我们理解我们的程序. 其实, 每一个程序都可以被抽象为一个执行的图.

到底什么是程序? 我们看上去我们会认为是我们的 C 代码, 经过下面的内容, 希望大家可以对于“什么是程序”有一个不一样的答案. 我们会用刚刚我们了解到的“图”的知识, 构建一个“状态机模型”. [Jiang]

“到底什么是程序”这样的问题是比较深刻的. 理论计算机学家深刻地研究了程序语言应该有的语义, 执行的过程等等. 但是我们从一个更加简化的角度来看, **程序就是状态机**. 每一个状态就相当于一个节点里的一些数据, 不同状态之间经过程序语句进行转移. 一个粗浅的理解是: 状态就是“堆 + 栈”(存放着我们的变量等), 初始状态就是“main 的第一条语句”, 迁移就是“执行一条简单语句”. 因为任何一个 C 程序都可以写成一个非复合语句的 C 代码, 并且的确有**这样的工具和解释器!**

这样的过程会对我们的调试代码带来好处. 比如, 我们可以使用 gdb 来检查我们感兴趣的输出, 同时我们可以使用 printf 指令向我们输出感兴趣的调试信息.

闲聊与练习

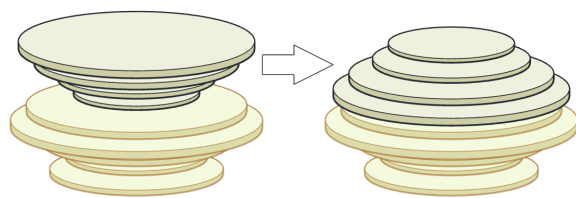
孫子曰：凡治眾如治寡，分數是也。鬥眾如鬥寡，形名是也。三軍之眾，可使必受敵而無敗者，奇正是也。兵之所加，如以礮投卵者，虛實是也。

Sunzi said: The control of a large force is the same principle as the control of a few men: it is merely a question of dividing up their numbers. Fighting with a large army under your command is nowise different from fighting with a small one: it is merely a question of instituting signs and signals. To ensure that your whole host may withstand the brunt of the enemy's attack and remain unshaken - this is effected by maneuvers direct and indirect. That the impact of your army may be like a grindstone dashed against an egg - this is effected by the science of weak points and strong.

— 《孫子兵法·兵勢》

Exercise 2.1: 翻炒煎饼: 选自 [Erickson] 第一章问题 4

假设你得到一堆 n 个不同大小的煎饼。你想把薄煎饼排个序, 这样小煎饼就在大煎饼的上面。你唯一能做的就是翻转——在顶部的 $k(k = 1, 2, \dots, n)$ 个煎饼下面插入一把刀然后将它们全部翻转。



Flipping the top four pancakes.

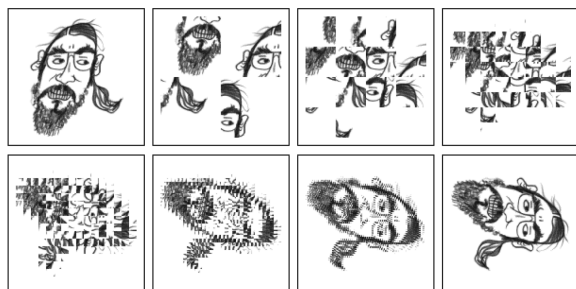
(1) 描述一种算法, 使用尽可能少的翻转对一堆任意的 n 个煎饼进行排序。在最坏的情况下, 你的算法到底执行了多少次翻转?

(2) 现在假设每个煎饼的一面都烧焦了。描述一种算法, 对任意堆叠的 n 个煎饼进行排序, 使每个煎饼烧焦的一面朝下, 同样保证翻转次数尽可能少。在最坏的情况下, 你的算法到底执行了多少次翻转?

(3) 使用你刚刚思考的结果, 完成 [UVA120 Stacks of Flapjacks..](#)

Exercise 2.2: 图像旋转: 选自 [Erickson] 第一章问题 9

假设我们想要将一个 $n \times n$ 的像素地图顺时针旋转 90° (n 是 2 的若干次幂)。一种方法是将像素地图分成四个 $\frac{n}{2} \times \frac{n}{2}$ 的块, 使用五次块传送将每个块移动到其正确的位置, 然后递归地旋转每个块。(为什么是五次? 和汉诺塔问题需要第三个柱子的原因一样。) 另一种方法是首先递归地旋转块, 然后将它们放到正确的位置。



The first rotation algorithm (blit then recurse) in action.

Exercise 2.3: k-d 树: 选自 [Erickson] 第一章问题 25

假设我们有 n 个散布在二维的盒子内的点。“k-d 树” 通过将这些点划分用递归的方式如下: 首先, 我们使用一条垂直的线将盒子分成两个较小的盒子, 然后使用水平线将每个较小的盒子再次划分, 如此反复进行, 始终在水平和垂直划分之间交替。每次划分盒子时, 划分线会通过盒子内的一个中位点 (不在边界上) 尽可能均匀地划分剩余的内部点。如果一个盒子不包含任何点, 我们就不再继续划分它; 这些最终为空的盒子称为单元 (cells)。

(1) 最后由多少个单元? 用 n 表示。

(2) 在最坏的情况下, 一条水平线到底能穿过多少个单元? 用 n 表示。

一些有趣的思想

PART

III

SECTION 3

二分法

我们在解决问题的时候, 常常采用一些有趣的方法来进行. 比如, 我们有逻辑推理能力, 来解答各种各样的问题. 我们先来考虑二分法.

我们先考虑这样的一个猜数字游戏: 假设有一个人选定了一个秘密数字, 并让你来猜这个数字是多少. 这个秘密数字是在一个已知范围内的整数. 你可以每次猜一个数字, 然后得到一个提示: 告诉你该数字是猜测的秘密数字的偏大还是偏小, 或者是猜中了. 基于这个提示, 你要做的是继续猜测直到猜中为止. 你的目标是用最少的猜测次数找到秘密数字.

在上面的问题中, 我们可以找到某个性质的边界, 其中分别是小于这个数的和大于等于这个数的. 也就是说, 我们要二分一个问题, 就是看一看这个边界是不是能够找到.

在这一部分中, 我们首先会叙述这个的一般原理, 然后观察几个基本的问题以及几个写代码的范式 - 很多时候写二分有关的代码是很容易犯错的. 结果就是无尽地死循环. 但是幸运的是, 我们可以避免这件事情发生.


整数二分
原理

我们的目标是找个性质的边界. 例如, 我们有如下的边界: 并且有一个命题 P , 左边的红色的部分是不满足 P 的, 右边的是满足 P 的.

那么, 要找到红色的最右边的那个, 就 (1) 首先要找到一个中间值 $mid = (l+r+1) \gg 1$, (2) 判断中间值是不是满足性质 P , 也就是 $check(mid)$. (2.1) 如果 P 满足, 那么 $l=mid$; (2.2) 如果 P 不满足, 那么 $r=mid-1$. 返回到 (1), 重复执行, 直到 $r \geq l$.

如果要找到绿颜色最左边的那一个, 和上面的问题相仿, 还是 (1) 首先要找到一个中间值 $mid = (l+r) \gg 1$, (2) 判断中间值是不是满足性质 P , 也就是 $check(mid)$. (2.1) 如果 P 满足, 那么 $l=mid$; (2.2) 如果 P 不满足, 那么 $r=mid+1$. 返回到 (1), 重复执行, 直到 $r \geq l$.

我们发现上述只是在取 mid 的时候和修改 l, r 的时候发生了一点小问题. 这是因为 C 中的数组的舍去问题. 如果不这样做, 有时候会发生死循环 - 就是说在锁定只有两个的时候, 不额外加一的时候, 可能会导致 l 在执行 $(l+r)/2$ 之后还是 l . 这样就相当于什么都没有更新. 肯定不是我们想要的.

 **P1163 银行贷款.** 个人认为这个题面似乎有点表述不清. 我们采用另一个更严谨的题目: 给出 n, m, k , 求贷款者向银行支付的利率 p , 使得:

$$n = \frac{m}{1+p} + \frac{m}{(1+p)^2} + \frac{m}{(1+p)^3} + \cdots + \frac{m}{(1+p)^k}$$

其中 p 保留 0.1%. ²

² 果然使用数学公式是很容易表达的

Idea1. 我们来“猜测” p , 然后根据我们的猜测根据公式计算, 看一看它到底还的多还是还的少. 如果多了, 就稍微把 p 往下调一点, 少了就把 p 往上调一点. 不过这道题也有够坑的 – 有的利率答案居然高达 300%! 所以二分的边界需要设置为 300% 才行. 我这里只让它执行了 10000 次二分操作 – 毕竟最后的精度不高. `◇C++P1163.cpp`

Remark 注意保留精度! 使用 `pow` 进行求和可能会扩大误差, 达到最后会差大约 200 元.

Idea2. 如果学习过了一些数学, 这个问题还可以使用数学的方法推演. 形如这样的叫做等比数列, 意思是后一项除以前一项, 结果总是一个常数. 大家耳熟能详的 2, 4, 8, 16, 32, ... 这一串数列就是一个典型的等比数列, 其中通向是 2^n . 其中 n 是第几项 (从 1 开始编号). 也就是说, 我要想知道第二项是多少, 就要带入 $n = 2$, 结果就是 $2^2 = 4$.

等比数列
求和


等比数列如何求和? 这就需要一些技巧: 我们假设等比数列的通项是 $a[n] = a_1 q^{n-1}$, 那么 $S_n = a_1 + a_2 q + a_3 q^2 + \dots + a_n q^{n-1}$.

我们发现这里面有很多的东西, 所以我们得想个办法把它们消掉. 采取两边同乘以 q , 两式相减, 就有神奇的效果. 这个方法也叫做错位相减法.


TBD: 公式推导

好了, 经过上面的推导, 我们就可以得到等比数列的求和: TBD

但是我们发现这个东西并不好解答... 确实, 我们并不能一味地通过一种方法解答问题. 当我们遇到困难的时候就要多换角度.

 **P2249 查找.** 这个就是最基本的内容了. 直接参考代码就可以了! 注意刚刚说过的一个问题: 到底是左端点还是右端点.


Remark 整数相关的二分的算法 bug 是比较隐蔽的. Java 标准库中一个类似的查找函数使用了类似的二分方法. 但是它使用了 `int mid = (low+high)/2`; 导致了问题. 这个 Bug 在 Java 的数组标准库里面待了 9 年. [这里是原创文章.](#)

 **P1676 Aggressive Cows G.** 这个是最小值最大的问题, 意味着我们一般使用按照答案二分的策略. 我们首先猜一个答案, 然后去施展我们应该有的构造, 最后来看一看这个是不是太小了.


我们可以假设牛棚都是空的, `check` 时如果当前牛棚与上一个住上牛的牛棚之间的距离 `dis >= mid`, 我们就可以让这个牛棚里住上牛, 反之向更远的距离寻找牛棚. 这是个贪心算法. 如果最后能安排的牛总数小于总的牛数, 那么就可以扩大需求. (`r=mid`) 反之, 就要缩小 (`l=mid+1`).

Question 1 为什么这个贪心算法是对的?

我们说: 按照上面的构造, 一定是“最省”的. 并且我们只要能说明只要不按照这样做不一定是最省的就可以了. 也就是, 最小值可能会变得更小.

 **P2678 跳石头.** 这个仍然是最小值最大的问题. 和上一个是类似的. 自己试着感受一下吧!

 **P3853 路标设置.** 这个和上面的问题也是一样的. 自己动手试一试吧!

 **P1314 聪明的质检员.** 这个虽然标号的颜色是绿色的, 但是仍然逃不过二分答案的区间. 不过, 这里面可能有些符号难以阅读. 我们来简单阅读一下:

求和符号
简介

求和记号是一大堆连加记号的缩写. 简单来说, 只是一个省略而已, 并没有万能的公式可以求和任何事情.

Iverson 的
括号
简介

Iverson 记号写作 $[..]$ 其中, 里面的 $..$ 是一个布尔表达式. 当里面的结果是真的时候, 值为 1, 否则值为 0.

Iverson 括号可以和求和一起搭配使用, 来达到简化求和记号的作用. 比如, 我们要交换两个求和记号的时候, 更好的想法可能是用这样的方法: TBD

介绍了刚刚的内容, 我们来简单梳理一下这个问题.

我们要得到 $\min |s - y|$, 就必须找到合适的 W , 进而得到对应的 y . 并且另一个观察是: y 越大, W 越小. 当 $y < s$ 时, y 偏小, 我们就要减小 W ; 当 $y = s$ 的时候, 我们就得到了我们想要的结果. 当 $y > s$ 时, y 偏小, 我们就要增大 W .

我们求 y 的过程满足单调性, 因此使用二分的方法即可. 到这里, 我们能够得到 70 分. 查询的部分有个双重的 for 循环. 这部分使用前缀和优化一下就好. 我们马上会提及.

SECTION 4

前缀和与差分

前缀和
普通版本

现在有一个数组, 请问 $\sum_{i=l}^r a_i$ 等于多少? 我们很容易用 for 循环实现. 但是, 如果这样的事情会发生多达 10^5 , 应该怎么办? 一个好的想法是我们可以把他们累加起来.

Definition 2 如果一个数组 a , 它的前缀和数组 s 的通项为 $s_i = a_1 + \dots + a_i$.

这时候要想求 $l \sim r$ 的和就求 $s_r - s_l$ 即可.

Question 2 既然有前缀和, 那么你认为什么操作下积可以被前缀吗? 你觉得能够前缀的问题有哪些特征?

我们发现上述的前缀和问题能够胜任查询问题, 但是对于修改操作并没有办法很好的胜任因为单点进行修改之后, 其之后的前缀和都要发生变化.

前缀和
何必要前缀
“和”?

事实上, 前缀和刻画了“连续进行若干次操作, 产生的一个综合影响可以通过某种手段撤销.”比如, 我们如果连着加他们, 到最后可以使用减法把影响的区间消除. 减法在数学中称为加法的“逆 (inverse) 运算”. 普通乘法的逆运算是除法.

事实上, 运算这件事情可以被定义得很广泛. 比如, 你可以在正方形纸片上面定义一个运算, 叫做“向右旋转 90 度”. 它的逆运算可以是“向左旋转 90 度”, 或者说“连续做 3 次向右旋转 90 度”.

下面我们来看一个比较奇怪的, 但是也能用上述的思想做的内容.

Example 现在有编号为 $0 \sim 10$ 一共 10 个球, 我们现在有若干个区间的对换. 具体地, 对于区间 $[l..r]$ 的对换之后, 如果原来这方面的球的编号是 $\dots, a_l, a_{l+1}, \dots, a_r, \dots$, 那么经过这次对换之后, 这个区间的球的顺序就变成了 $\dots, a_{l+1}, a_{l+2}, \dots, a_r, a_l, \dots$. 现在你有 n 条操作规则, 每条操作规则就是两个数 l, r . 现在, 我们想知道你连续执行编号 a 到编号 b 的操作规则之后, 得到的内容是多少. 注意有 m 次查询. 数据范围: $1 \leq n, m \leq 10^5, 0 \leq a, b \leq 9, 1 \leq l \leq r \leq n$.

我们如果这时候把“交换”当做一个运算, 运算的“数”就是你现在交换的区间左端点和右端点, 这样子就和刚刚加法减法的前缀和类似了. 事实上, 这样的对换在后续学习中是很重要的.

Remark 重要的对换: 如果你之后学习了 Polya 定理, 其中有一个重要的结论是任何一个置换都可以分解成若干个对换的复合. 这会对于你计数带有对称性的内容带来很大的帮助.

另外, 在数学中, 抽象代数中的群也有类似的刻画. 同样也有更加一般化的结论和内容. 不过要是学习这个, 必须有足够扎实的数学基础和对于许多内容的熟练掌握 (如数学分析, 高等代数等基础课程) 在这里我们不做讨论.

当然, 上述的内容只是一个简单的例子. 当你学习了更多的结构的时候, 很多结构天然地满足这个性质. 到时候请多加留意.

差分
普通版本 我们发现, 前缀和让我们拥有在 $\mathcal{O}(1)$ 时间查询的能力. 但是如果修改起来可能就麻烦了. 这里, 我们介绍一种方法, 使得我们可以在 $\mathcal{O}(1)$ 时间内修改, 并且能够 $\mathcal{O}(n)$ 查询出来单点的值.

我们现在的的问题是有一个数组 a , 每一次, 我要向 $l..r$ 的区间内的元素加上一个值 d . 最后只有一次询问, 问我现在第几个元素被改成几了. 这样的修改会发生很多次, 因此我们不能使用 for 循环来做.

我们发现, 在对于区间一整个加的操作中, 我们在这一个区间加和的过程中, 区间内部的两个数之间的差一直不变. 于是我们试着引入差分的定义:

Definition 3 对于一个数组 a , 我们定义 $d_i = a_i - a_{i-1}$, 那么 d 数组为原数组的差分 (difference) 数组.

TBD: 具体的操作: 一个点 $+x$, 另一个点 $-x$ 就可以. 详细描述之

挺有趣: 刚刚使用了累加, 我们才能得到了一个可以胜任区间求和, 但是做不了区间修改的东西. 现在我们让每一个内容是它减去它前面的内容, 居然可以胜任修改, 但是无法胜任区间的求和.

那么, 我们的原数组 d , 这个数组 a , 以及前缀和数组之间 s 有什么关系呢? 经过不复杂的数学推导, 我们可以发现:

TBD 插入一个他们之间关系的图

Remark 这个关系, 在你上了高中, 接触到了路程, 速度, 加速度的关系的时候, 会发现它们是出奇的一致的. 为什么? 路程, 速度, 加速度的关系就似乎是这里的 x, v, a 的关系. 完整的知识在大学才能揭晓 – 那时候你会学习数学分析, 更进一步地看一看在连续的情形下, 我们是如何做“前缀和”的.

差分
加一个等差数列? 如果我们要在之间加一个等差数列, 那该怎么办? 比如原数列是 1, 2, 3, 4, 5, 在区间 [1..3] 加上等差数列 2, 4, 6, 最后的结果是 3, 6, 9, 4, 5.

我们发现, 我们让原来的差分数组再差分一次不就好了! 等差数列再次差分, 就只要在前面加一个数, 在后面减掉一个数了, 就像刚才一样. 这是差分的一个重要的性质.

在练习中, 你会看到有哪些差分做起来是好做的. 你同时也会发现很多奇妙的公式.

差分
加一个平方数列? 这次我们使劲差分, 差分到三次, 你就会发现, 他们就会奇迹般地出现出来 0 的样式了.

Question 3 为什么是差分三次?

事实上, 我们会发现每次差分之后, 得到的内容就会消掉一次. 也就是从二次变到一次, 再到 0 次. 在 0 次的情形, 就是我们最开心的情况了. 如果下次要加上一些单项式的组合, 其实同样的方法也是适用的.

TBD: 阐述二维前缀和的内容. 并且给出基本的例子简单介绍容斥原理, 为后续做准备.

SECTION 5


贪心算法


贪心是指在最优化问题的决策过程中, 每次选择当前局面的最优决策. 不过需要指出的是, 当前局面最优不一定能得到全局最优. 通常, 我们要使用贪心算法, 至少要思考一下如何说明一下它的正确性.

有点有趣的是, 在推荐给大家的 Jeff Erickson 的 Algorithm 书中, 作者风趣地写道: “Greedy algorithms never work! Use dynamic programming instead!”.


这显示了使用贪心算法的副作用 – 没有说明胡乱贪心有时候不可取. 其中的 Dynamic Programming 是动态规划的意思, 现在可以认为是聪明的搜索 – 使用记忆的方法避免求解了一些重复的子问题. 我们会在后面简单了解.

我们来看若干个问题:

 **P1056 排座椅.** 对于单独的某邻近两列, 如果有 x 对爱唠嗑的同学, 选择拆散这一列, 就拆散了 x 对同学, 邻近两行也是同理的. 另外, 对于任意情况, 我们都应该拆散邻近两列或两行爱唠嗑同学对数最多的那两行或两列. 不然, 我们本可以拆散更多的同学. 我们刚刚的论证用了问题的描述以及反证法 (“不然...”). 虽然思路很好想, 但是注意输出的时候是按照编号输出的. `◇C++P1056.cpp`.

 **P1016 旅行家的预算.** 我们可以在油便宜的时候必须要买油, 只要比当前油价便宜就好. 如果在油箱的油消耗完之前不能到达比当前油价还便宜的地方, 就在这里把油箱加满油. 如果能到达比当前油价便宜的地方, 那就加油到刚好能跑到那个地方. `◇C++P1016.cpp`

不过要注意的是, 贪心可能很难, 贪心的结果也可能非常有趣. 我们下面来看一个有趣的脑筋急转弯.

 **Addition and Substraction Hard.** 这个题目的意思很简单: 给你一个只包含 ‘+’、‘-’、正整数的式子, 你需要在式子中添加一些括号, 使运算结果最大, 输出最大的结果.

首先我们看到我们必须在减号的后面加括号. 因为减号的后面才能使得符号发生变化. 在这个第一个括号里面, 我们就需要闭合的时候这个值尽可能的小了. 那么如何让这第一个括号里面的值尽可能小呢? 首先, 这个第一个括号的闭合肯定在式子的最末尾. 其次, 我们可能还要在减号的后面继续加括号, 但是要满足让原式的结果尽量大, 第二层括号里面的值的要求是也是尽可能最大 – 也就是让第二个括号里面的加号最多. 我们只需要把所有的减号后面的连加符号都括起来即可.

这就是我们的贪心思路了. 我们可能会说: 为什么不会有嵌套三层 (往上) 的情况? 其实, 我们注意到, 任何一个嵌套括号到了 3 层 (或往上), 一般形态为 $x_0 - (x_1 - (x_2 -$

$(x_3)))$ 其实可以被组合成为 $x_0 - (x_1 - x_2) - (x_3)$, 保留了原来的减号. 但是枚举每一个括号计算的时间是 $\mathcal{O}(n^2)$ 的, 难以应对数据量. 我们使用前, 后缀和的技巧来优化我们的计算.

要观察出这个思路需要相当对算术运算的体会. 官方给出的题解使用了动态规划的思路. 正如我们刚刚提到的, 这是一个“聪明地搜索状态空间”的算法. 我们可能会在未来重新回顾官方题解的做法.

SECTION 6

倍增


介绍
引例 1

不知道大家有没有在无聊³的时候玩过算 2 的几次幂的游戏: $2^1 = 2, 2^2 = 4, 2^3 = 8, \dots$. 许多同学在各种评论区分享了他们算过的最大值. 但是, 我们发现, 我们可以这样来算得更多一些: $2 \times 2 = 4, 4 \times 4 = 16, 16 \times 16 = 256, 256 \times 256 = 65536 \dots$.

如果记得 `unsigned int` 的最大值是 $2^{32} - 1$, 即 4294967295, 那么你可能应该可以很快地计算出 2^{64} , 甚至 2^{128} 了. 为了好玩, 我们给出 2^{256} 这个 78 位数:


115792089237316195423570985008687907853269984665640564039457584007913129639936

并且我们发现, 我们如果要任意的一个幂次, 就都可以用上面的一些内容表示出来. 比如, $2^3 = 2^2 \times 2^1$. 因为 $3 = (101)_2$. 由此, 我们就可以发明“快速幂”的算法. 我们只要 $\log_2 b$ 次来计算 a^b (不溢出的情况下).

 **P1226 【模板】快速幂 | 取余运算**. 这里只是多了一个取余数的运算. 我们发现 $(a \times b) \bmod c = ((a \bmod c) \times (b \bmod c)) \bmod c$. 用这个内容写代码就好了. `◇C++qpow.cpp`

ST 表
介绍

我们刚刚使用二进制去拼凑一个整数的方法能不能用于其他的问题呢? 其实, “可重复贡献的问题”就是我们可以用这样的方式做的. 我们可以预处理出 $f[i][j]$ 表示序列上起点为 i , 长度为 2^j 的区间的答案, 查询的时候使用拼凑的方式把我们的答案拼凑出来就可以了. 比如快速查询区间最值, 区间按位或, 区间按位和, 区间最大公约数等等. 他们都满足一个性质: $f[a..c] = f[a..b] \text{OP} f[b..c]$. 我们有 $\mathcal{O}(n \log n)$ 的时间预处理, $\mathcal{O}(1)$ 时间查询.

 **P3865 ST 表**. 我们可以用上述的方法来完成这道问题. `◇C++st.cpp`

有些时候我们还会在树上的最近公共祖先中遇到这样的倍增的思想. 具体我们可以到时候再了解.

SECTION 7

更多的练习与思考

TBD: 构造答案, 问题分解, 带有顺序的枚举法, 滑动窗口

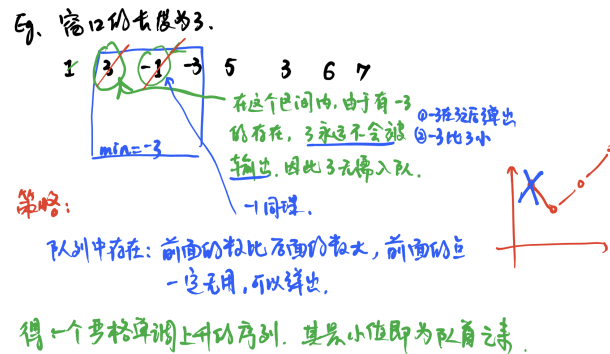
除此之外, 我们还会有很多很有趣的思考问题的方法. 我们下面举出几个例子:

滑动窗口
介绍

³ 没有无聊的时候? 相信我到时候一定会有. 比如河南省的会考. 如果当前情况维持不变的话, 会考理科的试题是非常充足的. 当时大概所有理科作答时间 (数学, 物理, 化学, 生物) 加起来总共用了一个小时左右, 所以在那个时候就可以轻松体会这个游戏了.

我们在有些问题的时候, 要求一个固定长度的区间内部的最大值和最小值都输出出来. 我们有如果我们使用暴力的做法, 可以这样做: 可以先用一个队列来维护窗口, 保证每次这个窗口里面存的是当前的所有元素. 遍历所有元素, 得到时间的复杂度是 $O(nk)$. 但是我们想一想这个真的需要这么复杂吗?

考虑优化, 有些元素似乎是用不了的. 具体如下图所示:



SECTION 8

并查集简介

并查集本身是一种数据结构, 主要用于解决元素的所属关系, 也就是看一看两个内容是不是在同一类中. 具体地, 就有如下的两点要求:

- 将两个集合合并
- 判定两个元素是不是在一个集合中

要达到这个目的, 我们原理: 每个集合用一棵树来表示, 树根的编号就是整个集合的编号, 每个节点储存着他的父节点的编号, $p[x]$ 表示 x 的父节点.

那么我们的几个操作就可以这样表示了:

- 判断树根: $p[x] == x$
- 求 x 集合所在的编号: $\text{while}(p[x] != x) \ x = p[x]$
- 合并两个集合 $px, py, x \neq y. \ p[x] = y.$

树与图

PART

IV

拿着郑州市的地图, 在所有两条或多条街道的汇合处或一条街道的尽头都画上一个黑点, 这样就有了一个组合数学中的图的例子. 在这个城市中, 某些街道是单行道, 只允许单向行驶, 因此你可以在每一条单行道的行进方向上画一个箭头 \rightarrow , 在所有的双行道上画一个双箭头 \leftrightarrow . 考虑栖息在你喜爱城市里的所有的动物和植物, 如果一个物种捕食另一个物种, 就在它们之间画上一个箭头, 这样你又得到了一个有向图. 得到了我们学习的食物网.

上述的例子说明, 图和有向图为相关对象可以让我们理解事情更加清晰. 图在计算机科学中也是一种极为有用的模型, 因为在计算机科学中出现的许多问题, 都能够很容易地通过图的算法去刻画. 在这之前, 我们要来看一看一些简单的内容:

SECTION 9

图的关键要素和存储

正如我们刚刚看到的, 我们发现图无非是一些节点和一些边组成的.

Definition 4

一个图 G (也叫做简单图) 是由两类对象构成的. 它有一个被称为顶点 (有时也叫做结点的元素的集合

$$V = \{a, b, c, \dots\}$$

和一个被称为边的不同顶点对的有限集合 E , 我们用

$$G = (V, E)$$

表示以 V 为顶点, E 为边的图.

Example

一个有 5 个顶点的图 G 的顶点是

$$V = \{a, b, c, d, e\}$$

以及边是

$$E = \{\langle a, b \rangle, \langle b, c \rangle, \langle c, d \rangle, \langle d, a \rangle, \langle e, a \rangle, \langle e, b \rangle, \langle e, d \rangle\}.$$

所以, 关键是把握住如何向计算机里面塞入这些数据. 我们有两种做法:

枚举每一个点对之间有没有边. 这样子, 我们可以开一个二维数组 $g[i][j]$. 其中 $g[a][b]$ 如果是 0, 那么说明这两个点之间没有边, 反之认为两个点之间有边. 上面的可以用如下的二维数组表示: TBD

在每一个节点上维护一个它可以到的相邻节点的列表. 这个列表的实现方式可能有很多种, 比如“链式前向星”就使用一个类似于链表的结构记录了这个列表. 有些时候, 可以使用另一个数组 `vector` 来进行存储.

TBD 加一个图

那么如果想到达相邻节点的相邻节点怎么办? 我们又没有存储它. 没关系, 我们只要先到一个相邻节点, 再在另一个相邻节点的位置上找到那个相邻的节点就可以了. 也就是说, 我们只要能够维护一个相邻节点的关系, 我们就可以在整个图上到达我们可以到达的节点了.

🔗* 通过手动模拟理解链式前向星的加边代码. 有些时候我们会使用一个链表来进行加班的操作. 我们来看代码:

```
int cnt=0; // 记录当前是第几条边
int head[MAXN]; //head[u]表示最后一条从u出发的边
struct Edge{
    int to,next,val;
    //to:从u出发到的节点
    //next: 前一条从u出发的边的cnt的值
    //val: 当前边边权
}e[MAXN];
```

我们来看一个问题的练习:

🔗B3643 图的存储. 这个是基本的问题. 让我们用不同的方法来进行书写. `◇C++graph-store-1.cpp`

由于这个问题需要我们排序输出, 所以我们还是不要使用链表的方式了. 因为排序的时候会很麻烦.

树的节点除了可以存当前的节点编号, 还可以维护一些辅助的信息; 边除了边的权重, 还可以加一些辅助的信息在边上. 这就需要具体看问题具体分析了.

下面我们来看一个可以用刚刚说的链式前向星进行的例子: 当然这个可能复杂了一些.

🔗P3916 图的遍历. 一个朴素的算法是从每个点搜索能到达的点, 再找出最大的. 但是这样的内容显然是无法处理 $O(n^2)$ 的数据的. 我们需要一些优化. 由于是最大的, 我们可以提前指定一个序关系, 使得我们从答案出发, 看一看哪些点会最终到达这个答案. 这就要求我们建立一个反向边就行了.

所以, 我们的策略是: 应该按编号从大到小 DFS 每个节点, 这样能保证一个点在被第一次访问的时候一定是能够到达最大的值的. 这样, 每个点就要访问 1 次, 从的时间复杂度是 $O(n)$, 可以做到. `◇C++P3916.cpp`

如果你有时候觉得中括号太多了, 可以把 `to`, `next` 从结构体里面提出来写成数组, 这样一来 `e[i].to` 就可以直接写作 `to[i]`. 当然是只有一个图的时候这样子也是可以的.

我们发现运用图来解决问题很多时候是很有趣的, 我们会简单说明有些常用的算法, 并做一点简单的应用.

拓扑排序 介绍

想象一下,你身为一个小组的学生们计划一个综合项目.每个学生都有自己的任务,但某些任务必须要在其他任务之前完成.我们要使用拓扑排序来确定任务的顺序.

假设我们的任务有: A - 搜集信息, B - 分析数据, C - 编写报告, D - 进行演示. 现在让我们来看看每个任务之间的依赖关系.

A 不依赖其他任务, 所以它排在第一位. B 依赖于 A, 所以它排在 A 之后. C 依赖于 B, 所以它排在 B 之后. D 依赖于 C, 所以它排在 C 之后.

所以最终的任务顺序是: $A \rightarrow B \rightarrow C \rightarrow D$.

要进行拓扑排序, 我们 (1) 首先找到所有没有前置依赖的顶点 (入度为 0 的顶点). 这些顶点可以作为排序序列的起点. (2) 然后, 从上一步得到的顶点中选择一个作为当前的顶点, 并将其添加到排序序列中. (3) 将当前顶点从图中移除, 并更新与其相关的顶点的入度. 具体地说, 对于每个与当前顶点相邻的顶点, 将其入度减 1. (4) 重复步骤 2 和 3, 直到所有的顶点都被处理和移除. 如果在这个过程中存在入度为 0 的顶点, 就继续选择一个添加到排序序列中. 最终, 得到的排序序列就是一个满足依赖关系的顶点顺序, 表明了任务的执行顺序的序列.

在 2020 年的 NOIP 中, 我们确实需要写这样一个内容, 不过需要加上高精度. 不加上高精度可以得到部分的分.

P7113 排水系统. 这是一个拓扑排序的问题, 这里大家只要简单模拟就行了. 当然注意分数通分的时候应该先加再乘. 当年也是因为这个丢掉了很多的分数.

P1038 神经网络. 我们必须保证前面的点都已经算过了, 我们才可以计算这个点. 所以, 我们就必须按照拓扑顺序来执行这些内容.

P3243 菜肴制作. 这是一个拓扑排序的例子. 但是会发现一个问题, 这里的要求是尽量先吃到质量高的菜肴. 那么应该怎么办? 可以让小的菜编号为 a , 大的为 b . 我们尽量想让 a 往前靠, 但是这个难以计算. 我们可以尝试让 b 尽量往后靠. 这样不论 b 在哪, 都能保证 a 在前面, 可以反向拓扑排序. (这时候还需要使用优先队列⁴进行维护)

⁴ 优先队列是可以让最大值出队列的一种数据结构.

最短路算法 介绍

最短路算法是图论中的一个经典问题, 它用于寻找图中两个顶点之间最短路径的问题. 最短路算法的核心过程是通过遍历图中的边和顶点, 找到连接起点和终点的最短路径.

最短路算法 Floyd 算法

给定一个带有边权的有向图, 我们的目标是找出图中任意两个顶点之间的最短路径长度. 这个问题被称为全源最短路问题, 因为我们需要找出从图中每一个顶点到其他所有顶点的最短路径.

首先一个基本的问题是, 从 u 到 v , 如果我们找到了一个节点 k , 使得 $\text{dis}(u, v) \geq \text{dis}(u, k) + \text{dis}(k, v)$, 那么我们就可以通过 $u \rightarrow k \rightarrow v$ 的方法经过路径.


算法的流程如下: 首先, 我们需要构建一个二维数组 `dist`, 用来存储每对顶点之间的最短路径长度. 如果两个顶点之间没有直接的边相连, 则将其距离设置为无穷大. 接下来, 我们初始化 `dist` 数组. 对于每一条边 (i, j) , 我们将 `dist[i][j]` 设置为边的权重值. 如果没有边直接相连, 则将 `dist[i][j]` 设置为无穷大.


然后, 我们使用三层循环遍历所有顶点, 并尝试通过第三个顶点 k 来优化从顶点 i 到顶点 j 的最短路径. 具体来说, 我们检查是否存在从 i 到 j 经过顶点 k 的路径长度比直接从 i 到 j 的路径更短. 如果是, 则更新 `dist[i][j]` 为更短的路径长度.

这个算法的证明, 如有兴趣请参考 TBD, 其中介绍了这个算法的历史. 我们可能需要在理解动态规划的思想之后再回来看它会比较好.


实际上, 这个 Floyd 算法可以用作另一种情形. 用于求“传递闭包 (transitive closure)”. 那么, 什么是传递闭包呢? 通俗的讲就是如果 $a \rightarrow b, b \rightarrow c$, 那么我们就建立一条 $a \rightarrow c$ 的边. 将所有能间接相连的点直接相连. 为什么这样做? 因为这个做就构造出了重要的“可达关系”. 我们就不用再枚举了.

```
void Floyd() {
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (g[i][k] && g[k][j]) g[i][j] = 1;
            }
        }
    }
}
```

 **B3611 【模板】传递闭包.** 我们可以用练习 B3611 来练习我们的这个算法.

 **POJ1975 Median Weight Bead.** 题目大意: 给出一些水滴之间的重量大小关系, 求有多少水滴的重量不可能是这些水滴重量的中位数. (水滴的数量保证为奇数)

这就可以建个图来转化问题. 假设 w_i 表示水滴的重量, i 到 j 有边代表 $w_i > w_j$. 求传递闭包, 对每个点都统计能到达多少个点、有多少个点能到达. 如果这两个数中有一个大于 $n/2$, 那它就不可能成为中位数.

 **找出最小权值的环.** 根据问题, 如果我们追踪 (起点终点为同一个点) 时, 就能获得从 i 到 i 的最小环, 因为如果没有环, 那么从 i 到 i 的路径长肯定还是无穷大, 如果有环, 则肯定从 i 到 i 的值肯定会被替换, 而且在 k 的循环中, 更小的环会替换掉旧的环, 所以, 通过这样, 可以获得环, 得到它的最小值就好了.

有 n 个点, m 条边, 除了起点, 最短路中最多还有 $n-1$ 个点, 所以最多对 m 条边进行 $n-1$ 轮松弛操作. 所谓“松弛”, 就是像 Floyd 算法那样, 如果找到了更小的, 就不断地更新 dis 的值为更小的那一个.

```
for(int i=1;i<n;i++){
    for(int j=1;j<=m;j++){
        dis[v]=min(dis[v],dis[u]+val[u][v]);
    }
}
```

当然, 这个算法是有一些慢的. 我们考虑加上一些优化:

优化 1. 减少不必要的循环. 我们可以添加 $flag$, 第 i 轮如果 $flag$ 的值没有改变, 直接跳出循环, 我们就可以结束了, 减少循环的次数.

```
int flag=1;
for(int i=1;i<n;i++){
    flag=1;
    for(int j=1;j<=m;j++){
```

```

        if(dis[v]>dis[u]+val[u][v]){
            dis[v]=dis[u]+val[u][v];
            flag=0;
        }
    }
    if(flag)break;
}

```

优化 2. 用 `queue` 存目前已经更新过的最短的路的节点。我们发现似乎没必要使用那么多的松弛操作。只有上一次被松弛的结点，所连接的边，才有可能引起下一次的松弛操作。这就启发我们用队列来维护哪些结点可能会引起松弛操作，就能只访问必要的边了。

🔗**P3371 单源最短路 (弱化版)**。请参看代码 `◇C++P3371.cpp`。

这份优化同样可以判断负环。同一条边入队次数大于等于 m (或者点更新的次数大于某个数，一般是 n)，说明存在一条路，满足只要一直走这一条路，这条路上节点的 `dis[]` 会不断减小。也就是，这条路是负环！

这种优化方法优化出来正是 SPFA，SPFA 可以用来判断负环是否存在。

🔗**P3385 判断负环**。请参考 `◇C++P3385.cpp`。

大家可能听闻了“卡 spfa”的声音。实际上，比如在网格图、菊花图之类的图上面它的复杂度是不稳定的。所以说在没有负权环的情况下慎用 spfa。

如果我们按照 spfa 的想法，在没有负边权的情况下，用优先队列优化，和 BFS 一样，我们就得到了 Dijkstra 算法。

算法的过程是，将结点分成两个集合：已确定最短路长度的点集（记为 S 集合）的和未确定最短路长度的点集（记为 T 集合）。一开始所有的点都属于 T 集合。初始化的时候 $dis(s) = 0$ ，其他点的 dis 均为 $+\infty$ 。

然后重复这些操作：(1) 从 T 集合中，选取一个最短路长度最小的结点，移到 S 集合中。对那些刚刚被加入 S 集合的结点的所有出边执行松弛操作。(2) 直到 T 集合为空，算法结束。

这里最小长度就展示我们用堆优化的方法来做这个问题了。直观来看，这个算法从已知点不断向未知点搜索，每个点只入队出队 1 次，所以从每个点出发的边只遍历 1 次，再加上优先队列入队的复杂度为 $O(m \log n)$ 。

这方面的代码可以这样写：

```

struct nodei{
    int dis,pos;
    bool operator <(const nodei &x)const{
        return x.dis <dis;
    }
};

void dijkstra(int s){ //s是起点
    std::priority_queue<nodei>q;
    dis[s]=0;
}

```

```

q.push((nodei){0,s});
while(!q.empty()){
    nodei tmp=q.top();
    q.pop();
    int x=tmp.pos ,d=tmp.dis ;
    if(vis[x])continue;
    vis[x]=1;
    for(int i=head[x];i;i=a[i].next ){
        int v=a[i].to ;
        if(dis[v]>dis[x]+a[i].val ){
            dis[v]=dis[x]+a[i].val ;
            if(!vis[v])//vis的值的含义： 0=这个点没有入队过 1=入队过
                q.push((nodei){dis[v],v});
        }
    }
}
}
}

```

我们可以使用数学归纳法来证明这个算法的确是正确的.

PROOF 我们要证, 在执行 1 操作时, 取出的结点 u 最短路均已经被确定, 即满足 $D(u) = dis(u)$.

初始时 $S = \emptyset$, 假设成立.

接下来用反证法.


设 u 点为算法中第一个在加入 S 集合时不满足 $D(u) = dis(u)$ 的点. 因为 s 点一定满足 $D(u) = dis(u) = 0$, 且它一定是第一个加入 S 集合的点, 因此将 u 加入 S 集合前, $S \neq \emptyset$, 如果不存在 s 到 u 的路径, 则 $D(u) = dis(u) = +\infty$, 与假设矛盾.

于是存在路径 $s \rightarrow x \rightarrow y \rightarrow u$, 其中 y 为 $s \rightarrow u$ 路径上第一个属于 T 集合的点, 而 x 为 y 的前驱结点 (显然 $x \in S$). 需要注意的是, 可能存在 $s = x$ 或 $y = u$ 的情况, 即 $s \rightarrow x$ 或 $y \rightarrow u$ 可能是空路径.

因为在 u 结点之前加入的结点都满足 $D(u) = dis(u)$, 所以在 x 点加入到 S 集合时, 有 $D(x) = dis(x)$, 此时边 (x, y) 会被松弛, 从而可以证明, 将 u 加入到 S 时, 一定有 $D(y) = dis(y)$.

下面证明 $D(u) = dis(u)$ 成立. 在路径 $s \rightarrow x \rightarrow y \rightarrow u$ 中, 因为图上所有边边权非负, 因此 $D(y) \leq D(u)$. 从而 $dis(y) \leq D(y) \leq D(u) \leq dis(u)$. 但是因为 u 结点在 1 过程中被取出 T 集合时, y 结点还没有被取出 T 集合, 因此此时有 $dis(u) \leq dis(y)$, 从而得到 $dis(y) = D(y) = D(u) = dis(u)$, 这与 $D(u) \neq dis(u)$ 的假设矛盾, 故假设不成立.

因此我们证明了, 1 操作每次取出的点, 其最短路均已经被确定. 命题得证. \square

 **P1462 通往奥格瑞玛的道路.** 我们发现, 求能到达终点路径上的收取的最大的费用的最小值, 我们可以用二分答案, 二分最大的费用. 如果最开始限制为 inf 依旧不能到达终点, 输出 AFK; 如果可以到达终点, 进行二分. `◇C++P1462.cpp`

简单计数问题

PART

V

SECTION 11

数学基础

在阅读数学相关的文献的时候, 我们可能因为数学的记号没有见到, 而产生恐惧. 现在, 我们就对常见的一些符号做一些简单的认识.

逻辑符号

数学家喜欢使用一些逻辑连接词来使他们描述的数学对象更加清晰. 符号 $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$ 分别表示逻辑上的“非”, “与”, “或”, “可以推出 (蕴含)”, “等价”.

联词与括号

Example $x^2 - 3x + 2 = 0$ 成立, 当 $x = 1$ 或 $x = 2$ 可以用这样的符号表示: $(x^2 - 3x + 2 = 0) \Leftrightarrow (x = 1) \vee (x = 2)$.

同时我们也需要区分必要性和充分性. 这时候我们可以使用真值表的方法来做事情. 另外, 如果把“若 P 则 Q ”的 P 和 Q 调换一下顺序, 再取反, 得到了 QP , 你会发现, 这两种说法是等价的.

并且为了好玩, 我们发现这些内容遵循如下的规律, 为了更清楚地说明, 可以用符号表示.

Theorem 1 (最基本的逻辑符号的运算规律) 我们发现有如下的最基本的运算规律, 以便于我们操作含字母的命题公式.

- 交换律:

$$(A \wedge B) \leftrightarrow (B \wedge A)$$

$$(A \vee B) \leftrightarrow (B \vee A)$$

- 结合律:

$$((A \wedge B) \wedge C) \leftrightarrow (A \wedge (B \wedge C))$$

$$((A \vee B) \vee C) \leftrightarrow (A \vee (B \vee C))$$

- 分配律:

$$(A \wedge (B \vee C)) \leftrightarrow ((A \wedge B) \vee (A \wedge C))$$

$$(A \vee (B \wedge C)) \leftrightarrow ((A \vee B) \wedge (A \vee C))$$

- De Morgan 律:

$$\neg(A \wedge B) \leftrightarrow (\neg A \vee \neg B)$$

$$\neg(A \vee B) \leftrightarrow (\neg A \wedge \neg B)$$

- 双重否定律:

$$\neg\neg A \leftrightarrow A$$

- 排中律:

$$A \vee (\neg A)$$

- 矛盾律:

$$\neg(A \wedge \neg A)$$

- 逆否命题:

$$(A \rightarrow B) \leftrightarrow (\neg B \rightarrow \neg A)$$

我们构造的证明,一般是形如 $A \Rightarrow B \Rightarrow C \Rightarrow \dots \Rightarrow G$. 其中 G 是我们的结论. 在数学中,每一个关系要么是公理,要么是由公理推导出来的命题.

集合
基本概念

我们总是希望把一堆东西放在一起加以研究. 这种趋势再 19 世纪末已经被明确提出来了. Cantor 等人提出了朴素集合论的思想. Cantor 说:“我们把集理解为若干个确定的,有充分区别的,具体的或者抽象的对象合并而成的一个整体”. 这种朴素的集合论的前提是 (1) 集合可以由任何有区别的对象构成; (2) 集合由其组成对象唯一确定, (3) 任何性质都可以确定一个具有该性质对象的集合.

我们可以用描述的方法表达这个集合的元素是什么. 比如如果我们要表示一个所有的 x , 满足 $x^2 - 1 = 0$, 我们就可以这样写: $\{x : x^2 - 1 = 0\}$. 这个冒号有时候也可以写作一个竖线⁵.

集合中 S 包含元素的个数用 $|S|$ 表示. 有时候称为该集合的“大小 (size)”. 很多计数问题都可以通过这样的一种方法得到更加清晰的表述. 我们发现, 这个想法和我们以往遇到的数学概念不同, 因为集合的给定方法在明确程度上可以由明显的不同. 例如“郑州一中的所有学生”, “集合的集合”, “Yanyan Jiang 中所有 a 的集合”, “所有集合的集合”.

甚至最后一个“所有集合的集合”, 干脆就是一个矛盾的概念!

PROOF | 对于集合 M , 设记号 $P(M)$ 表示 M 不是它本身的元素. ...

□

组成一个集合的对象叫做集合的元素, 如 x 是集合 X 的元素, 可以简单记作 $x \in X$ (或 $X \ni x$). 如果 x 不是集合 X 的元素, 可以简单记作 $x \notin X$ (或 $X \not\ni x$).

集合
包含关系

我们考察集合的交和集合的并, 以及集合的补.

Definition 5 两个集合的并的定义如下: $A \cup B = \{x : x \in A \vee x \in B\}$

Definition 6 两个集合的交的定义如下: $A \cap B = \{x : x \in A \wedge x \in B\}$.

这些就是集合的基本运算了. 我们可以通过这些内容来构造各种不同的东西.

映射可以认为是两个集合之间的对应关系. 这有点像送信:

映射
基本的定义

Definition 7 (映射) 设 A, B 是两个非空的集合, 如果按某一个确定的对应关系 f , 使对 A 中的任何一个元素 x , 在集合 B 中都有唯一确定的元素 y 与之对应, 那么就称对应 f 集合 A 到集合 B 的映射. 映射 f 也可记为 $f : A \rightarrow B$.

⁵ 我们高中学习的集合就是要写一个竖线. $\{x|x^2-1=0\}$ 但是这里由于不与 C 语言的“或”搞混, 先这样书写.

映射 映射有很多种类, 根据满足不同的条件, 我们可以将映射分为几种不同的类别:

映射的分类

Definition 8 如果映射 f 的定义域 A 中的每个元素都映射到 B 中的不同元素, 我们就说 f 是”一对一”或”单射”.

Definition 9 如果映射 f 的值域等于集合 B , 也就是说 B 中的每个元素都是 f 的某个元素的映像, 那么我们就说 f 是”映满”或”满射”.

Definition 10 如果映射既是单射又是满射, 那么我们就说它是”一一对应”或”双射”.

映射 那么映射能不能像一条链一样呢?

复合映射

其实是可以的. 它是通过将两个或更多的映射联结在一起形成的. 假设我们有两个映射, $f: A \rightarrow B$ 和 $g: B \rightarrow C$. 那么复合映射 $g \circ f: A \rightarrow C$ 就定义为对 A 中的每个元素 x , 首先应用映射 f 找到元素 $f(x)$, 然后应用映射 g 找到元素 $g(f(x))$. 这样便形成了从 A 到 C 的复合映射.

有时候这个就写作记号 $f \circ g$. 注意, 一般来讲 $f(g(x)) \neq g(f(x))$, 他们是不能交换的, 但是, 他们是可以结合的. 也就是以什么样的顺序算都是可以的.

SECTION 12

计数原理

基本的计数

原理

加法原理

“加法原理”是计数原理的一个基本策略. 如果我们有两个不相交 (即互不包含相同元素) 的事件, 其中第一个事件有 n_1 种方式发生, 第二个事件有 n_2 种方式发生, 那么这两个事件中的任一个发生的方法总数为 $n_1 + n_2$. 也是通常所说的“分步相加”. 局部的之和就是整体的. 下面是形式化的描述.

Principle 1 (加法原理) 如果 S 是一个集合, S 的一个有 m 个部分的划分 S_1, S_2, \dots, S_m , 满足不遗漏 ($S = S_1 \cup S_2 \cup S_3 \cup \dots \cup S_m$); 不重复 ($\forall i, j, S_i \cap S_j = \emptyset$), 那么 $|S| = |S_1| + |S_2| + \dots + |S_m|$.

⁶ 也就是不重复, 不遗漏地把这个集合分成若干部分

如果一个过程可以通过两个独立的步骤完成, 其中第一步有 m 种可能, 第二步有 n 种可能, 那么整个过程的总体完成事情的可能的方案数就是 $m \times n$. 如果我们用形式化地描述它们就可以这样说:

Principle 2 (乘法原理) 如果 S 是一个有序对的集合, 里面的元素形如 (a, b) . 两个有序对 $(a_1, b_1), (a_2, b_2)$ 相等当且仅当 $a_1 = a_2$ 且 $b_1 = b_2$. 这样. 如果 a 是从大小为 p 的集合里面抽出来的元素之一, b 是从大小为 q 的集合里面抽出来的元素之一, 那么 $|S| = p \times q$.

实际上, 我们可以把每一种选择当做一个“决策”, 平行的 (加法原理) 选择可以用分叉表示, 递进的 (乘法原理) 可以用层层深入表示. 例如下图: TBD.

下面有一些逆运算.

Principle 3 (减法原则) 如果 A 是一个集合, U 是一个包含 A 的更大的集合, 那么令 $\complement_U A = U \setminus A = \{x \in U : x \neq A\}$ 为 A 相对于 U 的补集. 那么 $|A| = |U| - |\complement_U A|$

Principle 4 (除法原则) 如果 S 是一个有限的集合, 划分成了 k 个部分, 每一个部分都有相同的元素, 那么划分的数量 k 就是

$$k = \frac{|S|}{\text{每一部分有多少个}}.$$

我们发现, 这些问题相当的基本, 基本到任何一个幼儿园小朋友在刚学习加减乘除的时候都会学习. 只不过现在我们就可以使用了更加有趣的方法来做了. 理解这些内容你可能知道了为什么幼儿园小朋友要掰手指头算数了.⁷

下面我们来考察一个由 Gian-Carlo Rota 提出的著名的组合问题. 它按照这个方法给我们的计数问题简单分了一个类.

⁷ 事实上, 幼儿园小朋友的这一行为体现的是自然数的递归地定义. 幼儿园小朋友还是很有智慧的.

Example (The Twelvelfold way) 我们来看著名的 “The Twelvelfold Way” 这个问题: 它包括了 12 种从有 n 个球放入有 k 个盒子里的方法. 每种方法具有独特的限制, 包括球和盒子是否是区分的及是否允许空盒子等.

| n 个球 | k 个盒子 | 想怎么放怎么放 | 每个盒子最多 1 个球 | 不允许有空盒子 | | | |
|----------|--|---------|-------------|---------|------|------|------|
| 不同的球 o0o | 不同的盒子 <table><tr><td>1</td><td>2</td><td>3</td></tr></table> | 1 | 2 | 3 | (1) | (2) | (3) |
| 1 | 2 | 3 | | | | | |
| 相同的球 ooo | 不同的盒子 <table><tr><td>1</td><td>2</td><td>3</td></tr></table> | 1 | 2 | 3 | (4) | (5) | (6) |
| 1 | 2 | 3 | | | | | |
| 不同的球 o0o | 相同的盒子 <table><tr><td></td><td></td><td></td></tr></table> | | | | (7) | (8) | (9) |
| | | | | | | | |
| 相同的球 ooo | 相同的盒子 <table><tr><td></td><td></td><td></td></tr></table> | | | | (10) | (11) | (12) |
| | | | | | | | |

我们下面来看这个问题.

问题 (1): n 个球, k 个盒子, 盒子和球都是不同的, 随便放. 我们希望做的事情是 “把 n 个球放入 k 个盒子”. 这时候, 我们对于第一个球的选择就随便选一个就好了. 因此有 k 种方法. 对于第二个球, 因为没有限制, 我们照样可以用 k 种方法... 一直到第 n 个球. 因此总共的方案是 k^n .

问题 (2): n 个球, k 个盒子, 盒子和球都是不同的, 每个盒子最多 1 个球. 我们假设盒子的个数多于球, 这样做的事情就会有意义一点. 我们希望做的事情是 “把 n 个球放入 k 个盒子, 每个盒子最多 1 个球”. 这时候, 我们对于第一个球的选择就随便选一个就好了. 因此有 k 种方法. 对于第二个球, 因为没有限制, 我们可以用 $k-1$ 种方法 (有一个已经占用了)... 一直到第 n 个球, 就有 $k-n+1$ 个. 因此总共的方案是 $k(k-1)(k-2)\cdots(k-n+1)$.

我们一般把这个叫做排列数, 因为它阐述的是从 k 个物品里面选择 n 个数的方法.⁸ 同时, 从 k 开始, 往下乘 n 个数也被称作下降幂 (falling power).

⁸ 注意这里的字母顺序可能和一般的教科书不同. 一般的教科书习惯写作 $A_n^k = n(n-1)\cdots(n-k+1)$. 这里的形式对于内容是没有影响的. 二者阐述的是同一件事情.

Definition 11 (排列数) 从 n 个物品里面选择 k 个数的方法数记作排列数. 记作 A_n^k . 计算方法为

$$A_n^k = k(k-1)(k-2)\cdots(k-n+1)$$

其中 $k(k-1)(k-2)\cdots(k-n+1)$ 可以被记作下降幂, 写作 $k^{\underline{n}}$.

问题 (3): n 个球, k 个盒子, 盒子和球都是不同的, 不允许有空盒子. 我们发现当我们的球的数量不少于盒子数量的时候这个内容才有意义.

既然我不允许有空盒子, 我先随便挑出来 k 个球去“压箱底”, 然后剩下的像刚刚一样随便放不就好了? 其实这个方法是不对的. 因为这样会算重复一些方案 – 你默认的要压箱底的和后来放的在这里是考虑次序的, 而原来的问题是不考虑次序的. 那我们该怎么做?

这事实上是集合的一个划分. 每一个划分正好对应一个集合. 我们如果能够把这个集合划分为 k 份, 然后再把每一个划分对应上一个盒子就好了. 第二步很简单, 直接乘上 $k!$ 即可.

关键是如何划分这个集合? 为了方便我们的符号书写, 我们先记 $\left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\}$ 为把 n 个集合划分为 k 个部分的个数. 这时候, 我们与其一口吃个胖子, 我们可以一步一步地考虑⁹

要把 $\{1, 2, \dots, n\}$ 划分为 k 份, 可以借助那些以往的状态可以把我们带到 $\left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\}$.

第一种情况是, 我们已经把集合 $\{1, 2, \dots, n-1\}$ 分为了 k 个部分, 现在的任务是把 n 放入任何这 k 部分的其中之一. 这就给了我们 $k \left\{ \begin{smallmatrix} n-1 \\ k \end{smallmatrix} \right\}$ 种方法达到这个目的.

第二种情况是, 我们已经把 $\{1, 2, \dots, n-1\}$ 分为了 $k-1$ 个部分, 并且让 $\{n\}$ 单独一份. 这样, 我们就有 $\left\{ \begin{smallmatrix} n-1 \\ k-1 \end{smallmatrix} \right\}$ 种方法.

这两种方法构建的分割是不同的: 因为在第一种方法中, n 始终位于一个大小 > 1 的划分部分中, 而在第二种方法中, $\{n\}$ 始终是一个单独的一部分. 因此这两种情况是不重叠的. 而对于任意一个 n 元素集合分割为 k 份, 必定可以通过这两种方法之一来构建. 因此根据求和法则:

$$\left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\} = k \left\{ \begin{smallmatrix} n-1 \\ k \end{smallmatrix} \right\} + \left\{ \begin{smallmatrix} n-1 \\ k-1 \end{smallmatrix} \right\}$$

成立.

要求得这个递归式的表达式是十分难的. 我们一般到此为止了. 事实上, 这个内容叫做**第二类 Stirling 数 (Stirling number of the second kind)**. 要计算第二类 Stirling 数, 我们有如下的公式:

Definition 12 (第二类 Stirling 数) 将一个大小为 n 的集合划分为 k 个部分的方案数被命名为第二类 Stirling 数. 记作 $\left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\}$.

Theorem 2 第二类 Stirling 数满足关系

$$\left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\} = k \left\{ \begin{smallmatrix} n-1 \\ k \end{smallmatrix} \right\} + \left\{ \begin{smallmatrix} n-1 \\ k-1 \end{smallmatrix} \right\}$$

于是, 我们一般使用递推计算的方式计算这个集合. 这个确实需要很多思考, 这就是为什么我们会用一个伟大数学家的名字命名它.

这下子, 我们就得到了第三个问题的答案: $k! \left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\}$.

问题 (5): n 个相同的球, k 个不同的盒子, 每个盒子顶多 1 个球. 这就要求我们搞清楚到底哪个可以有球, 哪个盒子里面没有球就行了. 所以我们要求从 k 个里面选取 n 个出来. 这个应该如何计算呢? 实际上, 我们可以先从排列数出发, 然后想一想把它们分

⁹ 这有点递归的意思了! 至此, 你应该能感受到为什么我们把递归问题放在第一节了.

成若干个组,也就是从小到大排个序. 这样子就是总共的组合数有 $A_k^n/k!$ 个. 为了方便起见,我们把这个定义做组合数.

Definition 13 (组合数) 从 n 个物品里面选取 k 个数的方案数为组合数, 记作 $\binom{n}{k}$, 或者 C_n^k . 定义为

$$\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k!}$$

问题 (4): n 个相同的球, k 个不同的盒子, 随便放. 由于每一个球是相同的, 所以我们需要关注每一个盒子里面被放了多少球. 因此, 我们就相当于要在这几个球的空档里面“插板”. 由于随意放置, 我们相当于要在 $n+k-1$ 个里面选出 k 个, 于是, 得到了

$$\binom{n+k-1}{k} = \frac{(n+k-1)!}{k!(n-1)!} = \frac{n(n+1)(n+2)\cdots(n+k-1)}{k!}.$$

我们把这个记作多重组合数的系数 (非标准官方译名):

Definition 14 (多重集合组合数) 多重集合的组合数定义为

$$\left(\binom{n}{k}\right) = \binom{n+k-1}{k} = \frac{(n+k-1)!}{k!(n-1)!} = \frac{n(n+1)(n+2)\cdots(n+k-1)}{k!}.$$

其中, $n(n+1)(n+2)\cdots(n+k-1)$ 这样的从 n 开始, 向上乘 k 个数这样的被称为上升幂. 方便起见记作 $n^{\bar{k}}$

于是, 我们得到了这个问题的答案: $\left(\binom{k}{n}\right)$.

问题 (6): n 个相同的球, k 个不同的盒子, 每个盒子不许空. 那么我们不妨首先把前几个球放到前几个球里面, 然后剩下的就得到了不受限制的状况了. 也就是我们这个问题的答案是 $\binom{n-1}{k-1}$.

问题 (7): n 个不同的球, k 个相同的盒子, 随便放. 我们可以把 $\{1, 2, \dots, n\}$ 划分进 i 个非空的盒子, 其中, $i \leq k$. 于是根据加法原理, 这个问题的答案是 $\sum_{i=1}^k \left\{ \binom{n}{i} \right\}$.

问题 (8): n 个不同的球, k 个相同的盒子, 每个盒子顶多一个球. 事实上, 如果 $n > k$, 那么不可能做到. 根据抽屉原理, 总有一个盒子要装两个球. 反之, 我们就可以做到. 于是这个问题的答案是

$$\begin{cases} 1 & \text{if } n \leq k \\ 0 & \text{if } n > k \end{cases}.$$

问题 (9): n 个不同的球, k 个相同的盒子, 不允许有空的盒子. 哈哈! 这不就是我们集合划分的定义吗? 这样, 我们就可以用 $\left\{ \binom{n}{k} \right\}$ 表示了.

问题 (12): n 个球, k 个盒子, 盒子和球都相同, 不能有空盒子. 其实这个是当我们把球放进盒子里面之后, 真正重要的是什么? 事实上, 我们发现我们只要关心每个盒子有几个球就好了, 并且我们不用关心有多少球的顺序. 等价地说, 就是把一个整数分拆. 比如 7 就可以这样分拆成 $1, 2, \dots, 7$ 部分:

| | |
|--|--------------|
| $\{7\}$ | $p_1(7) = 1$ |
| $\{1, 6\}, \{2, 5\}, \{3, 4\}$ | $p_2(7) = 3$ |
| $\{1, 1, 5\}, \{1, 2, 4\}, \{1, 3, 3\}, \{2, 2, 3\}$ | $p_3(7) = 4$ |
| $\{1, 1, 1, 4\}, \{1, 1, 2, 3\}, \{1, 2, 2, 2\}$ | $p_4(7) = 3$ |
| $\{1, 1, 1, 1, 3\}, \{1, 1, 1, 2, 2\}$ | $p_5(7) = 2$ |
| $\{1, 1, 1, 1, 1, 2\}$ | $p_6(7) = 1$ |
| $\{1, 1, 1, 1, 1, 1, 1\}$ | $p_7(7) = 1$ |

等价地说, 我们的要求是一个数 n 的 k 分拆, 分别记作 x_1, x_2, \dots, x_k , 满足如下的条件 (*):

- $x_1 \geq x_2 \geq \dots \geq x_k \geq 1$;
- $x_1 + x_2 + \dots + x_k = n$.

为了方便起见, 我们把整数 n 分拆成 k 部分记作 $p_k(n)$. 读作 “ n 的 k -分割” 下面我们同样用类似于递归的方法来求解这个问题:

假设 (x_1, \dots, x_k) 是 n 的一个 k -分割. 满足刚刚我们提到过的条件 (*).

我们对这个问题分类讨论: 第一种情况是, 如果 $x_k = 1$, 那么 (x_1, \dots, x_{k-1}) 是把 $n-1$ 分割成的一个不同的 $(k-1)$ -分割

第二种情况是, 如果 $x_k > 1$, 那么 $(x_1 - 1, \dots, x_k - 1)$ 是 $n-k$ 的一个不同的 k -分割. 并且每个 $n-k$ 的 k -分割都可以通过这种方式得到. 因此在这种情况下, n 的 k -分割数目为 $p_k(n-k)$.

由于所有的情况都已经讨论完毕, 因此, 我们可以使用加法原理, 把这两个部分加起来, 得到了 n 的 k -分割数目为 $p_{k-1}(n-1) + p_k(n-k)$, 即

$$p_k(n) = p_{k-1}(n-1) + p_k(n-k).$$

Definition 15

(分拆数) 定义分拆数 $p_k(n)$ 表示把一个正整数 n 分拆为 k 部分, 分别记作 x_1, x_2, \dots, x_k , 满足如下的条件的个数:

- $x_1 \geq x_2 \geq \dots \geq x_k \geq 1$;
- $x_1 + x_2 + \dots + x_k = n$.

Theorem 3

分拆数满足性质

$$p_k(n) = p_{k-1}(n-1) + p_k(n-k).$$

所以我们这个问题的答案就是 $p_n(k)$.

问题 (10): n 个球, k 个盒子, 盒子和球都相同, 随便放. 有了分拆数之后, 我们就可以决定到底要分拆多少个了, 于是答案就是 $\sum_{i=1}^k p_i(n)$.

问题 (11): n 个球, k 个盒子, 盒子和球都相同, 每个盒子顶多 1 个球. 它和第 (8) 问的情况类似. 同样要么能做, 要么不能做. 原理还是依照第八个问题一样.

这样我们就得到了整个表格的全貌:

| n 个球 | k 个盒子 | 想怎么放怎么放 | 每个盒子最多 1 个球 | 不允许有空盒子 | | | |
|----------|--|---------|-------------|---------|--|---|--|
| 不同的球 o0o | 不同的盒子 <table><tr><td>1</td><td>2</td><td>3</td></tr></table> | 1 | 2 | 3 | k^n | k^n | $n! \left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\}$ |
| 1 | 2 | 3 | | | | | |
| 相同的球 ooo | 不同的盒子 <table><tr><td>1</td><td>2</td><td>3</td></tr></table> | 1 | 2 | 3 | $\left(\left(\begin{smallmatrix} k \\ n \end{smallmatrix} \right) \right)$ | $\binom{k}{n}$ | $\left(\left(\begin{smallmatrix} k \\ n-k \end{smallmatrix} \right) \right)$ |
| 1 | 2 | 3 | | | | | |
| 不同的球 o0o | 相同的盒子 <table><tr><td></td><td></td><td></td></tr></table> | | | | $\sum_{i=1}^k \left\{ \begin{smallmatrix} n \\ i \end{smallmatrix} \right\}$ | $\begin{cases} 1 & \text{if } n \leq k \\ 0 & \text{if } n > k \end{cases}$ | $\left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\}$ |
| | | | | | | | |
| 相同的球 ooo | 相同的盒子 <table><tr><td></td><td></td><td></td></tr></table> | | | | $\sum_{i=1}^k p_i(n)$ | $\begin{cases} 1 & \text{if } n \leq k \\ 0 & \text{if } n > k \end{cases}$ | $p_k(n)$ |
| | | | | | | | |

不要担心这张表格看起来有些复杂. 其实, 这张表格没有记忆的必要. 现在我们只需要学习排列数和组合数就可以建立一个很好的模型了. 这些概念是非常有趣和实用的, 它们能够帮助我们解决很多有趣的问题.

上述材料里面的有时候我们还会遇到更加复杂的问题, 比如对于分拆数, 我们需要将一个数分拆成若干个部分, 并且考虑它们之间的顺序. 都可以通过一些递归的方法来解决. 我们只当做对大家的训练. 一个初学者当然需要看过足够多的例子, 加以大量的思考才能设计出比较好的这方面的内容. 大家完全不必着急.

假设我们有 n 个不同的球, k 个不同的盒子. 我们可以用一个映射的方式来描述不同的放置方法. 具体来说, 我们可以把每个盒子看作一个“投影”, 而每个球就是我们要放入的“元素”. 这样, 每一种放置方法就可以看作是一个特定的映射.

那么, 任意的映射就是我们刚刚的“随便放”; 单射就是我们的“每个盒子只放一个球”; 满射就是“每个盒子不能空”. 因此, 这个表格更为一般的情况你就能够看得懂了.

| N | M | 任何一个 $f : N \rightarrow M$ | 单射 $f : N \xrightarrow{1-1} M$ | 满射 $f : N \xrightarrow{\text{onto}} M$ | | | |
|----------|--|----------------------------|--------------------------------|--|--|---|--|
| 不同的球 o0o | 不同的盒子 <table><tr><td>1</td><td>2</td><td>3</td></tr></table> | 1 | 2 | 3 | k^n | k^n | $n! \left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\}$ |
| 1 | 2 | 3 | | | | | |
| 相同的球 ooo | 不同的盒子 <table><tr><td>1</td><td>2</td><td>3</td></tr></table> | 1 | 2 | 3 | $\left(\left(\begin{smallmatrix} k \\ n \end{smallmatrix} \right) \right)$ | $\binom{k}{n}$ | $\left(\left(\begin{smallmatrix} k \\ n-k \end{smallmatrix} \right) \right)$ |
| 1 | 2 | 3 | | | | | |
| 不同的球 o0o | 相同的盒子 <table><tr><td></td><td></td><td></td></tr></table> | | | | $\sum_{i=1}^k \left\{ \begin{smallmatrix} n \\ i \end{smallmatrix} \right\}$ | $\begin{cases} 1 & \text{if } n \leq k \\ 0 & \text{if } n > k \end{cases}$ | $\left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\}$ |
| | | | | | | | |
| 相同的球 ooo | 相同的盒子 <table><tr><td></td><td></td><td></td></tr></table> | | | | $\sum_{i=1}^k p_i(n)$ | $\begin{cases} 1 & \text{if } n \leq k \\ 0 & \text{if } n > k \end{cases}$ | $p_k(n)$ |
| | | | | | | | |

说了这么多的理论东西, 下面我们来看一看实践的内容吧!

P5520 青原樱. 一句话题意: 一共有 n 个位置, m 棵树, 两棵树之间要有空位, 问总共有多少种选法. 我们把这 m 棵树以及他们所占的位置拿出来, 那道路上就剩下 $n-m$ 个坑, 而这 $n-m$ 个坑有 $n-m+1$ 个空位, 我们要把带坑的树插进这 $n-m+1$ 个空位中, 一共的插法就有 A_{n-m+1}^m 了.

道理是很简单, 我们应该如何实现求排列数, 组合数的代码呢?

排列数可能是比较直接的. 直接一个 `for` 循环就好了.

第一个想法是我们干脆按照它说的模拟一遍不就好了? 但是问题在于, 有时候乘上去可能会超出数据的精度与大小, 从而得到错误的结果. 嗯, 我们需要一个优化. 其实,

排列数

组合数

递推计算

组合数有一个很著名的递推关系式,

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

这个递推关系的意义是, 从 n 个元素中选择 k 个元素的组合数等于两部分之和: 一部分是从 $n-1$ 个元素中选择 $k-1$ 个元素的组合数, 另一部分是从 $n-1$ 个元素中选择 k 个元素的组合数. 如果你只想递推某一行的, 那么你可以使用这个恒等式:


$$\binom{n}{k} = \frac{n-k+1}{k} \binom{n}{k-1}.$$

并且从 $\binom{n}{0=1}$ 开始从左往右开始做. 我们可以使用组合数的定义 $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ 进行证明.

 **CF817B Makes And The Product.** 题目大意: 给定一个长度大于 3 的数列, 多少个由 $(i, j, k) (i < j < k)$, 使得 $a_i \cdot a_j \cdot a_k$ 是最小的?

我们发现, 最小乘积是由数组的三个比较小元素相乘得到的. 因此首先排个序, 然后分一些情况来考虑, 分别按照最小的, 第二小的, 第三小的个数分类¹⁰.

如果你发现自己很难把问题分清楚, 那么我推荐**韩老师讲的这系列计数问题**(来自 Bilibili: BV1C34y1R7wf) 的视频. 虽然韩老师讲的是数学竞赛相关的内容, 题目类型可能不同, 但是对于思维的品质的要求可能是很相似的.¹¹

 **Cantor 展开.** 很多时候我们会枚举排列, 比如 abc 三个字母的排列有:

$$abc, acb, bac, bca, cab, cba$$

六种. 那么, 上述的内容是字典序排序过后的内容. 我们在搜索的时候或许希望得到他们的排名, 这样子我们就可以愉快的把一个字符串的状态表示变成一个整数. 再比如, 2 个 a , 1 个 b , 1 个 c 组成的所有的串按照字典序的编号可以如下:

$$aabc(1), aacb(2), \dots, cbaa(12).$$

我们比如输入一个字符串, 能不能输出它的编号呢?

首先我们要了解多重集合有多少个排列. 先把所有可能, 也就是全排列处理出来, 然后相同元素可以随意互换位置, 按照分组的想法“除掉”就行了。

$$\frac{(n_1 + n_2 + \dots + n_k)!}{n_1! n_2! \dots n_k!}$$

有兴趣的同学可以阅读严格证明.

下面直接求解一般情况的问题 (并不限定字母的种类和个数). 设输入串为 S , 记 $d(S)$ 为 S 的各个排列中, 字典序比 S 小的串的个数, 则可以用递推法求解 $d(S)$.

假设我们输入了 $caba$, 我们就有如下的一张图 TBD, 其中边上的字母表示“下一个字母”, $f(x)$ 表示多重集 x 的全排列个数. 例如, 根据第一个字母, 可以把字典序小于 $caba$ 的字符串分为 3 种: 以 a 开头的, 以 b 开头的, 以 c 开头的. 分别对应 $d(caba)$ 的 3 棵子树. 以 a 开头的所有串的字典序都小于 $caba$, 所以剩下的字符可以任意排列, 个

分类的时候一定要注意不重复, 不遗漏. 很多时候可以按照一个特定的顺序或逻辑执行.

¹¹ 今天的 *Twelvefold way* 只是做一个引入. 如果你想了解更多的内容和应用, 欢迎去追完整个系列.

数为 $f(cba)$; 同理, 以 b 开头的所有串的字典序也都小于 caba, 个数为 $f(caa)$; 以 c 开头的串字典序不一定小于 caba, 关键要看后 3 个字符, 因此这部分的个数为 $d(aba)$, 还需要继续往下分。感兴趣的同学可以参考 [🔗P3014 Cow Line S.](#) 只不过, 这里面的具体的例子。如果希望得到这个题的问题, 可以参考 [🔗P2518 计数.](#)

[🔗UVA1649 二项式系数](#). 题意: 给定 m , 输出所有的 (n, k) , 使得 $\binom{n}{k} = m$. TBD

动态规划简介

PART

VI

这一部分我们继续跟随状态机的模型，探求问题的状态，用一种比较聪明的方法来
说明如何比较聪明地遍历问题。

SECTION 13


初步的问题

数字三角形
介绍

TBD: 数字三角形的相关内容请参考

这是一个耳熟能详的问题。不过一个问题需要仔细地考虑：为什么方程 $d[i][j] = a[i][j] + \max(d[i+1][j], d[i+1][j+1])$ 的后半段直接可以取最大值？事实上，我们发现这是要求最大决定的——如果连“从 $(i+1, j)$ ”出发走到底部的和都不是最大的，加上 $a[i][j]$ 之后也肯定不是最大的。这个性质被称为**最优子结构 (optimal structure)**。有“全局的最优解包含着局部的最优解”的想法。具体如何进行，我们可以先使用搜索试试看，之后分析出状态转移的规律，就可以使用迭代的方式进行实现了。

我们来看下面的问题：

 **P1004 方格取数**。想法 1：我会搜索！我希望暴力枚举出所有可能的情况。

上述做法直接解决了一整个大问题。但是在解决的时候可能会出现一些重叠的子问题。并不太好。我们想一想可以如何称为若干个子问题。第一个想法是定义 $f[i][j]$ 表示从 $(0,0)$ 走到 (i,j) 的过程。这样可行吗？看上去不行，因为我们没有记录重复的数——重复的数是没有办法再取的。

那么走两次，我们可以这样设计： $f[i_1][j_1][i_2][j_2]$ 表示所有从 $(1,1), (1,1)$ 走到 $(i_1, j_1), (i_2, j_2)$ 的路径的最大值。如何处理同一个格子被取两遍的呢？只需要保证当前处理的时候不相同即可。

这里有 4 种情况，因为每一个都可以从上来和从左来。我们从最后一步考虑，有如下的四类情况 TBD：一个集合关系，下下，下右，右下，右右直接判定即可。 ◇C++1004-4D.cpp

我们还可以把这个优化：由于只能向下，向右走，不能走回头路，当 $i_1 + j_1 = i_2 + j_2$ 的时候，格子才可能重合。

充分条件和
必要条件
简介

这里面，我们说只有满足这个条件才可能重合，意味着只要重合了就一定会满足这个条件。但是， $i_1 + j_1 = i_2 + j_2$ 无法推出一定重合。我们就说他们“重合”是 $i_1 + j_1 = i_2 + j_2$ 的**充分 (sufficient) 条件**， $i_1 + j_1 = i_2 + j_2$ 是他们“重合”的**必要 (necessary) 条件**。或者用符号表示，是这样的：“ $(i_1 + j_1 = i_2 + j_2) \Leftarrow$ 重合”。

由于有一个等式了，我们可以“消掉”一个量。我们提出一种更加简化的方法：让 $k = i_1 + j_1 = i_2 + j_2$ 。 $f[k, i_1, i_2]$ 表示所有从 $(1,1), (1,1)$ 到 $(i_1, k-i_1), (i_2, k-i_2)$ 路径的最大值。

看上去这会让我们转移方程难以写。但经过分析，也是可以做到的，根据图，有如下的四类情况

- 下下: 从 $f[k-1][i_1-1][i_2-1]$, 重合加上 $w[i_1][j_1]$, 不重合加上 $w[i_1][j_1] + w[i_2][j_2]$.
- ...

其实也没什么大不了, 只是把刚刚的状态浓缩到了 k 里面. 下面我们来看代码 `◇C++1004-3D.cpp`


技巧

缩减编码复杂度

事实上, 调代码是非常折磨人的. 如果我们能写出易于检查的代码就好了. 这里面, 我们想把 `f[k][i1][i2]` 所减掉, 有没有什么办法呢? 其实有两种办法: 第一种是使用引用: 输入 `int &x = f[k][i1][i2]`; 这样下次使用的时候 `x` 就相当于 `f[k][x1][x2]` 了. 另外一个可以使用 `#define` 关键字. 不过记得使用 `#undef` 取消宏定义在使用结束的时候. 第一种情况用的很多.

Remark

编写易于理解, 不言自明的代码有些时候是保持思维逻辑清楚的很重要的一个习惯. 每当我们面临一个困难的问题的时候, 我们可以想一想有没有什么方法简化它. jyy 老师在这篇文章说过这样的一段话: “有个小朋友 Segmentation Fault 了也不知道哪里来的自信, 一口咬定是机器的问题. 给他换了机器, 并且教育了他机器永远是对的. 这个小插曲体现了编程的基础教育还有很大的缺憾, 使得竞赛选手大多都缺少真正的 ‘编程’ 训练, 我看他们对着那长得要命的 `if (...dp[a][b][c][d][e][f][n^1]...)` 调的真叫一个累. 让我不由得想起若干年前某 NOI 金牌选手在某题爆零后对着一行有 20 个括号的代码哭的场景.”

 **P1006 传纸条.** 传纸条和上一个问题基本是类似的. 双倍经验的时间来了.

DP 的多重

视角

状态集合的角度

我们可以用如下的检查单来思考一个 (可能的) 动态规划问题. 因此, 我们可以把在这个属性下具有相同特征的内容划分为若干个集合, 然后根据每一个划分, 找到相应的规律, 就可以得到对应的结果了.

Theorem 4

在思考动态规划问题的时候, 可以采用以下的检查单:

A. 状态表示:

- (1) 我状态表示归类的是哪一类的问题?
- (2) 要在这一类问题上体现哪些属性?

B. 状态计算

- (1) 当前状态可以由哪些状态得来?
- (2) 对于这些内容, 这个属性前后的关系是什么?

DP 的多重

视角

DFS 的视角

有时候, 如果我们的递推关系过于奇怪, 我们可以回到我们的老本行, 写出没有额外变量的 dfs 程序, 然后使用数组来递推. 由于我们的函数调用关系, 这个依赖关系是在调用的时候就能够轻松做出来的. 由于子问题有重叠, 每次我们只要把一个子问题计算一遍存起来就好了.

记忆化搜索和递推二者都确保了同一状态至多只被求解一次. 但是它们实现这一点的方式则略有不同: 递推通过设置明确的访问顺序来避免重复访问, 记忆化搜索虽然没有明确规定访问顺序, 但通过给已经访问过的状态打标记的方式, 同样达到了的目的.

与递推相比, 记忆化搜索因为不用明确规定访问顺序, 在实现难度上有时低于递推. 且能比较方便地处理边界情况. 但与此同时, 记忆化搜索难以使用一些更加聪明的优化方式, 我们在接下来的背包问题中可以看到一些.

接下来我们来看几个类似的问题.

最长上升子
序列问题
简介


我们现在考察最长上升子序列 (LCS) 的问题. 根据我们的检查单, 我们决定定义状态 $f[i]$ 表示集合 $a[i]$ 表示以 $a[i]$ 为结尾的严格单调上升子序列. 要维护的属性是最大值. 现在我们考虑所有到达了 $f[i]$ 的内容. 看看它可以从哪来:

TBD: 加一个图示

分析了上面的内容, 我们就可以发现状态转移方程为


$$f[i] = \max\{f[k]\} + 1, \forall k \in [1..i-1], f[k] < f[i].$$

上升子序列给我们的感受是往上升. 那么下面我们来看一个既有上升又有下降的内容.

 **登山**. 我们可以按照中间是哪个点是最高点分析. 先分为 $a[0], a[1], a[2], \dots, a[n-1], a[n]$ 是山峰这几类. 我们分别求出每一类的长度最大值就是整个的最大值. 不是一般性, 如果峰值是第 k 个的最大长度, 并且左边选哪些和右边的情况互不相干, 那么就在左边和右边分别跑一下 LCS 问题, 然后找到 \max 就行了.

在做模拟题的时候, 我们可能留意了“合唱队形 (NOIP)”这个问题. 其实, 这个是一个对偶. 去掉多少人就是总数减去留下多少人.

Remark 对偶问题. 我们说两个问题是对偶的, 感觉上就是两个问题表达的是一个问题的两个方面. 或者更直观的说, 有一种对称性. 例如这个问题和合唱队形的问题; 到未来大家学习最大流和最小割, 他们都具有对偶的感受.


 **P2782 友好城市**. 这里的要求是不交叉. 我们发现我们要求的序关系消失了. 我们考察所有合法的建桥方式和上升子序列之间的联系: 对于任何一个合法的建桥方式, 从一侧观察一边的点, 另一边都是严格上升的. 对于任意一个严格上升的子序列, 我们都能够找到合法的架桥方式. 也就是他们之间构成双射. 所以我们按照自变量大小进行排序看因变量的 LCS 就好了.

其实, 没有交叉意味着没有逆序对. 如果你曾经实现过归并排序, 你一定对这个不会陌生.

映射 将给定集合的每个元素与另一个集合的一个或多个元素相关联的一种思想. 我们在刚刚的问题里面发现了一对一的这样的情况, 因此可以断定两个问题的大小是一样的.

这次, 我们想要知道你挑选出来的上升子序列里面, 其和是多少. 你会发现, 最长上升子序列并不意味着最大的和. 我们又要按照刚刚的方法分析了. 状态 $f[i]$ 表示所有以 $a[i]$ 为结尾的上升子序列, 属性是和的最大值. 状态计算的划分是可以划分为上一个数字选的是空, $a[1], \dots, a[i-1]$. 于是, 我们就得出了状态转移方程:

$$f[i] = \max\{f[k] + a[i]\} +, \forall k \in [1..i-1], f[k] < f[i]$$

 **大盗阿福**. 直觉来看, 我们想要设置 $f(i)$ 代表当前抢劫到了第 i 个店铺的最大收益. 于是, 当前的状态被划分为两块: 抢劫第 i 家店铺, 得到 $f[i-2] + w[i]$, 以及不抢劫第 i 家店铺. 于是, 我们得到状态的转移方程为 $f[i] = \max(f[i-2] + w[i], f[i-1])$.

这个状态需要依赖上面两维的状态. 如果我们只希望依赖上面一维的状态, 我们还需要增加一维: 用 $f(i, 1)$ 表示上一家店铺被抢了, $f(i, 0)$ 表示上一家店铺没有抢. 因此, 我们就可以转移了.


映射
表达关系
最长上升子
序列
之和

这种转移有一些头疼. 于是, 我们可以使用一个特殊的方法 - 请看

TBD: 状态图

我们下面来正式把这个说一说: 定义 $f(i, 0)$ 表示当前站在第 i 个建筑前面, 当前状态位于 j 的所有走法, 得到的最大值. 下面决定状态转移方程. 考虑 $f[i][0]$, 有哪些走法可以走到 0? 其实, 我们可以从上一个 0 走到 0; 或者从 1 走到 0. 因此, 它们的最大值分别是 $f[i-1][0]$ 和 $f[i-1][1]$ - 毕竟没有选择这家店铺. 下面考虑 $f[i][1]$. 我们只能从 $f[i-1][0]$ 走过来. 这样子, 获得的收益是 $f[i-1][0] + w[i]$. 综合去取 \max 即可. 图示如下:


TBD: 图示

 **最长公共上升子序列**. 这个问题我们定义状态 $f[i][j]$ 为所有由第一个序列的前 i 个字母, 第二个序列的前 j 个字母构成的公共上升子序列, 属性是要求最长的. 但是我们现在在转移的时候因为缺少条件, 我们还需要知道现在结尾的数是多少, 以便于我们判断是不是可以向后增加. 具体地, 我们这样修改我们的定义: “状态 $f[i][j]$ 为所有由第一个序列的前 i 个字母, 第二个序列的前 j 个字母构成的公共上升子序列, 并且有 $b[j]$ 结尾”.

那么, 有哪些状态可以转移到了 $f[i][j]$ 呢? 我们可以包含两类: 所有包含 $a[i]$ 的公共上升子序列, 另外的是左右不包含 $a[i]$ 的公共上升子序列. 第二类里面, 由于它最后不包含第 i 个字母, 说明它只可能包含前 $i-1$ 个字母. 即从状态 $f[i-1][j]$ 转移来. 那第一类呢? 根据状态的定义, 由于同时包含 $a[i]$ 和 $b[j]$. 由于 $a[i]$ 是不确定发的, 我们需要继续细分, 就像刚刚的 LCS 问题一样. 我们考虑序列的倒数第二个数. 有可能是空, $b[1], b[2], \dots, b[j-1]$. 这样一来, 我们就从实际出发, 发现如果是 $b[k]$ 作为倒数第二个字符的话, 那么值应该是 $f[i][k] + 1$. 不过这个 DP 问题可能还需要对代码做等价变形, 我们来看一看: TBD

现在我们做代码的等价变形, 可以 TBD.

Remark 一个问题, 尤其是困难的问题, 搞清楚来龙去脉是重要的. 任何感觉到难的内容可能只是缺乏了前置应该了解的东西. 所以, 很多时候, 看一看它的历史, 你就能知道更加多样的东西. 甚至追寻着历史的规律, 有一天你也能为解决这一类问题添砖加瓦!


 **股票买卖**. 题目叙述: 给一个长度为 $N (1 \leq N \leq 10^5)$ 的数组, 数组中的第 i 数字表示给定股票在第 i 天的价格. 设计一个算法计算能获取的最大利润, 最多完成 k 笔交易. 你不能同时参与多笔交易 (你必须在再次购买前出售掉之前的股票). 一次买入卖出合为一笔交易. 第一行包含整数 $N, k (1 \leq k \leq 100)$, 表示数组长度和最大交易数, 第二行 N 个不超过 10000 的正整数, 表示完整的数组. 输出一个整数, 表示最大利润.

我们发现在例子的情况下, 我们能进行的操作是“买入”和“卖出”. 造成结果是“手中有股”和“手中无股票”. 这下子, 我们发现最好按照这样的划分方法, 才可以把原来的内容描述清楚. 如果我们手中有货, 我们在下一天到来的时候既可以继续持有, 或者卖出, 同时得到一定的收益 (得到 $w[i]$); 如果我们手中无货, 那么下一天到来的时候, 我们可以买入 (并付出 $w[i]$), 或者按兵不动.

我们效仿背包的情况: 假设现在进行到了第 i 天, 正在进行第 j 笔交易 (买入就算做这笔交易), 有 $f[i][j][0] = \max(f[i-1][j][0], f[i-1][j-1][1] + w[i])$. 同样的有 $f[i][j][1] = \max(f[i-1][j][1], f[i-1][j-1][0] - w[i])$.

Question 4 如果卖出的时候, 使用了这个会导致全局最大值不对吗?

我们会遍历所有的空间, 正如我们前面所说, 这是一个“聪明的搜索”, 所有的状态都会被计算到的.

 **股票买卖 2.** 我们这时候发现状态影响决策有手中有货, 手中无货的第一天, 以及手中无货大于等于第二天 (冷冻期).

TBD: 状态图手中有货转圈圈 (0), 有货-> 手中无货第一天 (+w[i]), 手中无货 1-> 无货的第二天 (0) 手中无货转圈圈 (0), 手中无货 2-> 手中有货 (-w[i]). 出口有两个

转移方程, 根据上图就有: $f[i][0] = \max(f[i-1][0], f[i-1][2] - w[i]); f[i][1] = f[i-1][0] + w[i]$, 以及 $f[i][2] = \max(f[i-1][1], f[i-1][2])$.

运用状态机的视角真不错. 我们在很多时候在处理很多问题的時候也可以这样做.

SECTION 14


背包问题

背包问题是一类很经典的问题. 我们首先介绍一些常见的策略, 然后仔细看一看“0-1 背包问题”. 背包问题选出的内容里面没有内在的关系. 有时候可以成为组合类的 DP.

假设你是一个背包客, 要去旅行, 但是你能只能带一只背包. 现在的问题是, 你面前有一些物品, 每个物品都有自己的重量和价值. 你希望在背包的承重范围内尽可能装入最有价值的物品.

然而, 你的背包有个限制: 它只能承受一定重量的物品, 超过这个限制它就会撑破了. 所以你必须仔细考虑, 该怎样选择物品放进背包.

这就是所谓的 0-1 背包问题: 你要在一系列物品中做出选择, 每个物品只有一个 (0 个或 1 个) 存在的机会. 你不能切割物品, 只能选择全部放入或不放入.

 **P1048 采药.** 这是一个最为普通的背包问题. 我们现在考虑如何设计方案, 以及有什么好的办法来做这件事情. 我们设计状态 $f[i][j]$ 表示在集合“考虑前 i 个物品, 总容量为 j ”的价值的最大值. 那么根据最后一步, 可以把状态表示转化为两大类 - 要么选择第 i 个要么不选第 i 个.

不选的方案的话, 那么是 $f[i-1][j]$, 如果选取的话, 那么有分为之前的加上第 i 个. $f[i-1][j-v[i]]+w[i]$. 我们只要找到他们的最大值就好了. 请看代码 `◇C++P1048-2D.cpp`.

下面我们来看他是如何转移的. 我们发现, 如果我们采用倒序的方式循环的话, 就可以被压缩到一维了. 请看代码参考 `◇C++P1048-1D.cpp`

假设我们所有的这些内容都有无限件可以选呢? 这时候我们就可以同样定义 $f[i][j]$ 为所有只从前 i 个物品中选, 且总体积不超过 j 的选法的集合. 要维护的属性同样是最大值. 下面我们来看状态的计算. 既然我们可以选 $0, 1, 2, \dots, s-1, s$ 个, 那么就需要分别对于这个进行转移了. 选 0 个就是 $f[i-1][j]$; 选 1 个就是 $f[i-1][j-v[i]]+w[i]$; 选 2 个就是 $f[i-1][j-2 \times v[i]]+2 \times w[i], \dots$. 按照道理来讲, 这个可以写作代码了. 我们可以第一层枚举物品, 第二层枚举体积, 第三层枚举选多少个. 如 `◇C++complete-bp-primary.cpp`

我们来看一看有没有优化的空间. 我们观察到 $f[i][j]$ 的表达式

$$f[i][j] = \max(f[i-1][j], f[i-1][j-v] + w, f[i-1][j-2v] + 2w, \dots, f[i-1][j-sv] + sw)$$

与 $f[i][j-v]$ 的表达式有时候类似, 把 j 换为 $j-v$ 即可:

$$f[i][j] = \max(f[i-1][j-v], f[i-1][j-2v]+w, f[i-1][j-3v]+2w, \dots, f[i-1][j-sv]+(s-1)w)$$

注意这里的 s 的表达式是不依赖于 v 的, 并且 $s = \lceil j/v \rceil$.

我们发现上面的项和下面的是可以对齐的, 他们之间有很多的性质. 比如, 下面的每一项比上面的每一项少了一个 w , 因此上面的最大值等于下面的最大值加上 w . 于是 $f[i][j]$ 的值就可以替换做

$$f[i][j] = \max(f[i][j], f[i][j-v] + w).$$

这就是我们得到的最终表达式. 同样可以优化为 1 维的, 只要把体积的枚举的顺序改为顺序的循环就行了. 事实上, 当空间优化为 1 维的时候, 只有完全背包由于无限的关系, 需要从小到大循环. 我们可以使用 `◇C++complete-bp-1d.cpp` 来看一看. 其中 $v[i]$ 是重量, $w[i]$ 是价值.

多重背包是每一个里面有有限个物品的问题. 遵循刚刚的, 如果某一个物品有 s 件, 那么转移方程就有

$$f[i][j] = \max(f[i-1][j], f[i-1][j-v] + w, f[i-1][j-2v] + 2w, \dots, f[i-1][j-sv] + sw).$$

同样的, 我们现在来看一看 $f[i][j-v]$ 是什么:

$$f[i][j-v] = \max(f[i-1][j-v], f[i-1][j-2v]+w, f[i-1][j-3v]+2w, \dots, f[i-1][j-(s+1)v]+sw).$$

唯一不同的地方是最后一项都是一样的, 并且上面的比下面的每一项多一个 w . 但是关键不同是最后多了一项. 我们的目标是求上面的最大值; 但是我们并不能根据下面的反推上面的最大值的. 那么, 我们再看一项:

$$f[i][j-2v] = \max(f[i-1][j-2v], f[i-1][j-3v] + w, f[i-1][j-4v] + 2w, \dots, f[i-1][j-(s+2)v] + sw)$$


$$f[i][j-3v] = \max(f[i-1][j-3v], f[i-1][j-4v] + w, f[i-1][j-5v] + 2w, \dots, f[i-1][j-(s+3)v] + sw)$$

...

注意到 $j-kv$ 是模 v 余数相同的, 于是考虑画一个数轴来看一看所有的情况. TBD 也就是每一个需要用到前面的几个状态的最大值, 这个就可以用滑动窗口解决了. 请看代码 `◇C++mulbp-deq.cpp`

下面的这个思路就比较简单了. 可以把它拆分成 0-1 背包问题, 再使用 0-1 背包的模板做就好了. 不过这里的划分也是有技巧的, 我们可以使用以前倍增介绍的一个方法, 按照二进制拆分. 如 `◇C++mulbp-bin.cpp`.

下面我们来看上面介绍的一些对应的习题.


 **P1049 装箱问题**. 这里面没有了“价值”. 怎么做? 其实体积同时也是价值, 然后用 0-1 背包求解就可以了. 请看 `◇C++P1049.cpp`


有时候我们的体积可能是二维的, 我们来看下面的例子:


多重背包问题
一个困难的
做法


多重背包问题
拆分为 0-1
背包

二维费用的
背包问题
简介

 **宠物小精灵之收服**. 这下子, 费用变为二维的了. 费用之一是精灵球数量, 之二是皮卡丘的体力值; 价值就是小精灵的数量. 首先考虑状态表示, 定义 $f[i][j][k]$ 表示从前 i 个物品中选, 且花费 1 不超过 j , 花费 2 不超过 k 的选法中, 最大的价值. 那么 $f[i][j][k] = \max\{f[i-1][j][k], f[i-1][j-v_1[i]][k-v_2[i]]+1\}$. 当我们把所有状态都算过了之后, 得到答案的时候就可以收服 $f[k][N][M]$ 个精灵. 最少耗费的体力就看 $f[K][N][m]$ 的 m 最小是多少, 使得我们可以等于最大值. `◇C++2dcost.cpp` (所有的体积维度都是倒着循环的, 可以发现)


 **潜水员**. 这个问题类似于上面的问题. 但是也有一些变动. 我们定义为 $f[i][j][k]$ 表示从前 i 个物品中选, 且花费 1 恰好是 j , 花费 2 恰好是 k 的选法中, 最小的价值. 那么 $f[i][j][k] = \max\{f[i-1][j][k], f[i-1][j-v_1[i]][k-v_2[i]]+1\}$. 之后, 枚举 $j \geq m, k \geq n$ 的里面找一个最小值就可以了. 对应到代码上面, 我们只需要按照实际的含义把 $f[0][0][0] = 0$, 其余是 $f[0][j][k] = +\infty$, 表示不合法 - 毕竟到不了. `◇C++U291791.cpp`

 **数字组合**. 我们定义状态表示 $f[i][j]$ 为所有只从前 i 个物品中选取, 且总体积恰好是 j 的方案数的集合. 属性是方案数. 那么考虑不包含物品 i 的所有选法, 为 $f[i-1][j]$, 以及包含物品 i 的所有选法, 就是 $f[i][j] = f[i-1][j] + f[i-1][j-v_i]$. `◇C++BL4004.cpp`


 **分组背包问题**. 这下, 每一次是能从一个物品组内最多选择一个物品. 剩下的就和原来的一样了. 这样子, 定义 $f[i][j]$ 定义为只考虑前 i 个组内总体积不超过 j 的, 划分的方案依据是从当前的这个内选哪个物品. 如第 0 个, 第 1 个, \dots , 第 s_i 个. 因此转移方程就是

$$f[i][j] = \max(f[i-1][j-v_{i,1}] + w_{i,1}, f[i-1][j-v_{i,2}] + w_{i,2} + \dots + f[i-1][j-v_{i,s_i}] + w_{i,s_i})$$

和 0-1 背包一样, 照样可以去掉第一维. 我们的做法如下: `◇C++grouping.cpp`

 **P1064 金明的预算方案**. 这个本质上是一个分组背包的问题, 把每一个主件和附件的组合当做一个组, 每一个组里面有一些选择: 购买主件; 购买主件和附件 (所有的排列); 这样就是说有若干个物品组, 每个物品组里面的每一个物品是当前的决策, 并且是互斥的. 然后使用分组背包的方法得到一个最大价值.

上面的问题如果可以由很多个子节点呢? 这就扩展为了树上的有依赖的背包问题了. 下面我们来看一例:

 **P2014**. 我们首先定义状态, 我们可以使用递归的思路来解决这个问题. 我们会想, 对于这个点而言不同的体积的时候得到的最大的价值是多少呢? 比如这个点是点 u , 要求以 u 为根的时候在不同的体积之下, 最大的价值是多少. 于是我们定义 $f[u][j]$ 表示所有以 u 为根节点的子树中选取, 且总体积不超过 j 的方案数中, 获得的最大的.

接着来看如何做. 肯定, 要想选子树, 这个当前的根节点一定是要选的. 但是它的子节点就不一样了. 对于 $f[u][j]$ 的任何一个方案, 可以分为方案: (1) 从第一棵子树中选取的方案, (2) 从第二棵子树中选取的方案, \dots , (m) 从第 m 棵子树中选取的方案每一棵子树每部也可以通过体积去划分. (不按照方案划分是因为代价太大了) 这样子, 我们就可以按照分组背包的方式循环一遍. 用这样的方法我们就把诸多的一类问题用一个状态表示了.

更通用的问题如下: `◇C++tree-bp.cpp`

背包的方案数
简介


分组背包
简介

树上的背包
简介

SECTION 15

关于区间的问题

有些动态规划问题, 我们设计状态需要考察一个区间. 我们从石子合并这个经典问题开始看起.

 **P1775 石子合并 (弱化版)**. 假设这时候我们认为这是在一条链上的情形. 也就是不能首尾合并. 这时候, 我们定义 $f[i][j]$ 表示所有从 i 到 j 合并的方案, 属性是最小值. 下面我们来考虑状态的计算问题. 我们来看一看哪个可以到达这个状态. 我们考虑合并两个区间, 会发现它的分界点不同. 所以这就启发我们使用不同的分界点去划分现在的集合. 假设分界线落在 k 和 $k+1$ 之间, 那么它需要的体力最小值就是 $f[l][k] + f[k][j] +$ 左右两边的和, 也就是先合并左边, 再合右边, 最后就把两堆合在一起. 最后的是所有的子集的最小值. 状态转移很好写, 但是**注意循环顺序!**


转移方程: $f[i][j] = \min\{f[i][k] + f[k+1][j] + \sum_{s=i}^j a[s]\}$. 其中 k 从 i 枚举到 $j-1$. 状态空间是 n^2 , 需要枚举起点, 有 $\mathcal{O}(n)$, 总共时间复杂度是 $\mathcal{O}(n^3)$. 计算 $300^3 = 2.7 \times 10^7$, 完全可以.

接下来我们来看代码: **请留意循环顺序!** 按照区间长度从小到大枚举. `◇C++P1775.cpp`

从上面的代码中, 一般而言, 区间 DP 可以首先循环长度, 然后循环左端点, 之后算右端点, 最后枚举分界点. 这样是使用循环去遍历状态. 正如我们前面所说, 我们也可以使用记忆化搜索的方法写这个内容, 当转移不明确的时候.

Question 5 如果每次允许合并相邻的 n 堆, 应该如何做? 说一说大致思路.


我们接下来的问题可以设置状态为前 i 个数成了 j 个的过程. 这相当于 DP 里面套了一层 DP. 我们这里不做讨论.


 **P1880 石子合并**. 下面我们来考虑环形的状况. 我们如何把环的情况展开成一条区间呢? 因为环形剪掉一条边就成了一个链, 一个朴素的想法是我们可以枚举缺口在哪. 就可以用区间 DP 的方法做了. 但这样的时间复杂度是 $\mathcal{O}(n^4)$, 难以接受. 下面介绍一种优化方式:

我们本质上是 n 个长度为 n 个链的式子合并问题. 我们可以这样做: TBD

这样一来, 我们使用长度为 $2n$ 的区间, 就能保证我们只处理 n 个区间就可以枚举到所有的情况了. 这样我们的复杂度是 $\mathcal{O}((2n)^3)$. 这样的方法可以处理大多数的环形 DP 问题.

请参看代码 `◇C++P1880.cpp`.


 **P1063 能量项链**. 我们现在断环为链, 像上一个问题一样. 对于一个链, 我们定义状态的表示 $f[i][j]$ 为所有将 $i..j$ 区间合并成为一个珠子的方式. 属性是维护最大值. 接着来看合并的时候状态的计算. 我们来看一看哪个可以到达这个状态, 根据最后的不同点来划分. 这个和上一个类似的: 有一个分界线 (在原来数组的视角下注意这时候是共用的). 根据这个我们可以把集合划分为若干个子集. 其中分界线分别为 $i+1, i+2, \dots, r-2, r-1$. 假设当前的分界线是 k 的话, 那么就会有将 $(i, k), (k, j)$ 最后将两个合并释放的能量. 用数学公式写出来就是 $f[i][k] + f[k][j] + w[i] \times w[k] \times w[r]$. 这就是我们使用线性的做法, 现在我们考虑环形的. 运用上一个问题的技巧, 在后面一个 $2n$ 的链上面做 DP 就可以了. `◇C++P1062.cpp`

 **LOJP10149. 凸多边形的划分.** 这个问题需要我们一定的观察与思考. 我们首先发现, 任意作一个三角形, 它就会把左边的和右边的三角形划分开. 因为题目中有一个重要的条件 – 互不相交. 这就保证了区间左右的独立性. 所有这样的方案把整个内容分为了独立的三部分. 这是区间问题里面很重要的一个特征. 我们只要在这个状态下左半边的划分, 右半边的划分和的最大值, 就可得到和上一个问题一样的想法. 也就是比如我要考虑从 1 到 n 的划分, 中间选了点 k 作为分界点, 就有 $f[1][k] + f[k][n] + w[1] \times w[k] \times w[n]$. 我们来求每一个它们的最小值就可以了. 这一个问题虽然和上一个问题构造非常不同, 但是其转移也非常的相似. 下面我们详细看一下这个应该如何正式化:

定义 $f[l][r]$ 维护集合所有将 $(l, l+1), (l+1, l+2), \dots, (r-1, r), (r, l)$ 划分为三角形的方案的值的最大值. 在进行状态计算的时候, 我们枚举 $l+1, l+2, \dots, r-2, r-1$, 就可以把问题分为若干类. 对于每一类, 其转移到当前的值为 $f[l][k] + f[k][r] + w[l] * w[k] * w[r]$.

很烦人的地方是, 这个问题需要写高精度. 因为 $(10^9)^3 \times 100$ 大概会有 30 位数. `int` 的最大值是 2147483647, 9 位数; `long long` 的最大值是 9223372036854775808, 19 位数. 我们应该秉持先做对, 再做好的原则进行. 也就是先做对, 把样例和小测试数据做好, 然后再用高精度写剩余的部分. 不加高精度的部分如 `◇C++LOJP10149-part.cpp` 所示.

下面加上高精度. 为了方便起见我们直接用这个数组存位数, 直接整合进 f 数组里面. 请看代码的 `add` 部分和 `mul` 部分. `◇C++LOJP10149.cpp`

 **P1040 加分二叉树.** 我们看到这个问题, 发现其计算公式很像区间 DP 的计算的方式: 分为三个独立的部分. 关键是, 这个中序遍历是不是具有这样的形式, 使得我们可以在上面做区间 DP 呢?


回顾: 现在有一棵树的中序遍历, 我们考察任意的一个子树, 可以发现其在序列里面一定是连续的一段. 这就让我们可以进行选取根节点进行中序遍历. 我们定义 $f[l][r]$ 为所有将 $l..r$ 区间构造成一个二叉树的情形. 属性是维护所有二叉树的最大值. 我们找到最后一个不同点, 根据这些类划分为不同的集合. 我们按照根节点的位置划分. 这样就划分为了若干类.

和上面的问题一样, 如果根节点在第 k 个点的话, 最大值应该如何求? 应该是 $f[l][k-1] \times f[k+1][r] + w[k]$. 下面考虑应该如何记录方案.

其实记录方案无非是决定最后在更新的时候再某个地方记上一笔: “节点 k 已经成为了这个子树的根.” 于是定义 $g[l][r]$ 表示 $l..r$ 区间的根节点选哪个. 在输出前序遍历的时候就先输出这里的根 ($g[1][n] =: R$)¹², 同时知道左子树的区间和右子树区间为 $1..R-1, R+1..R$, 反复进行这个过程就行了.

字典序最小的方案应该如何做? 实际上我们只要让根节点的值最小就好了. 也就是找到最靠左的一个分界点. 只有在小于当前答案的时候才更新, 并且记录. 如 `◇C++P1040.cpp`

我们看一看二维的区间 DP. 这时候区间就看上去有点奇怪了.

 **P5752 棋盘分割.** 这里面看上去有一个陌生的统计量均方差, 不过不用担心. 不过我们来看均方差的公式 $\sigma = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n}}$, 要是想要这个带根号的最小, 就意味着可以求 σ^2 最小. 简单变形就有:¹³

¹² $:=$ 表示 “定义做”. 冒号在被定义的表达式那一侧

¹³ 但其实我们可以不用变形的. 这里只是简单体会一下操纵求和记号.

$$\begin{aligned}
& \sum_{i=1}^n (x_i - \bar{x})^2 \\
&= \frac{1}{n} \sum_{i=1}^n (x_i^2 - 2x_i\bar{x} + \bar{x}^2) \\
&= \frac{1}{n} \left(\sum_{i=1}^n x_i^2 - \bar{x} \sum_{i=1}^n 2x_i + n\bar{x}^2 \right) \\
&= \frac{1}{n} \left(\sum_{i=1}^n x_i^2 - \bar{x} \cdot (2n\bar{x}) + n\bar{x}^2 \right) \\
&= \frac{\sum_{i=1}^n x_i^2}{n} - \bar{x}^2
\end{aligned}$$

Remark 这个推导在概率论中是比较常见的。

这就是我们试图最小化的东西。也就是所有部分平方和的最小值。好，下面我们来看动态规划部分。

定义 $f[x_1][y_1][x_2][y_2][k]$ 表示子矩阵 $(x_1, y_1), (x_2, y_2)$ 切分成 k 不分的所有方案。其中 x 是行, y 是列。维护的属性是 $\sum_{i=1}^n (x_i - \bar{x})^2$ 的最小值。

接下来来看状态计算。我们认为有沿着 x 轴切; 沿着 y 轴切。一共各自有 7 种情况, 分别选上面和下面的情况。沿着 x 轴切有类似的情况。我们的目标是求每一类的最小值, 然后取 \min 。对于每一类, 我们有上面继续切的分值, 加上下面剩余的分值。由于右边的和是固定的, 于是可以用二维的前缀和求出来。最后求解就可以了。

如果要用循环来实现, 那么会很复杂。并且循环的顺序也可能一不留神写错。这时候我们采用记忆化搜索的方式完成本问题。 `◇C++5752.cpp`

SECTION 16

树形 DP

下面我们把刚刚的这种思想扩展到树上, 看一看在树上会如何确定状态, 并且使用 `dfs` 来自然地决定他们之间的转移。

🔗P1352 没有上司的舞会。 我们仍然使用前面的问题开始想起。定义状态 $f[u][0]$ 为从以 u 为根的这个子树中选择的方案, 但是不选 u 的方案; $f[u][1]$ 与之类似, 但是要选择 u 的方案。维护的属性是选择的点最大值。接下来来看状态的计算。如果要得到 $f[u][0]$, 先计算出来他们所有儿子的值 $f[s_0][0], f[s_0][1], f[s_1][0], f[s_1][1], \dots$ 。要想让整个最大, 那么我需要对所有的子树最大。也就是 $f[u][0] = \max(f[s_0][0], f[s_0][1]) + \max(f[s_1][0], f[s_1][1]) + \dots$ 。那么 $f[u][1] = h[u] + f[s_0][0] + f[s_1][0] + \dots$ 。这个一共有 $2n$ 个状态, 需要枚举的是两个儿子, 所以时间复杂度为 $\mathcal{O}(n)$ 的。 `◇C++P1352.cpp`

🔗树的直径。 给定一棵树, 树中包含 n 个节点, 编号为 $1 \sim n$ 和 $n-1$ 条无向边, 每条边有一个权值。现在想找到树中的最长的直径。

这次我们的边是无向边, 我们可以通过建立两条有向边的情况下来模拟无向边的情形。下面考虑如何设计状态。我们假设把所有的路径都枚举一遍, 在这些里面找到边的权重最大的。我们先想着把它分为若干类, 并在所有的进行取得 \max 。

首先随便找到一个点, 把它当做根节点. 我们在每条路径上面选择一个高度最高的点, 然后把这条路径归到最高的点这个上面去. 我们按照这条路径上面最高的这个点去枚举, 似乎就会好很多.

那么如何进行状态的计算呢? 首先我们把所有的子节点的往下走的最大长度. 这个挂的点可以分为两种情况: 第一种是一直往下走, 另一种是两个拼在一起, 也就是穿过了这条路径. 第一种的计算方式就是往下走的最大距离加上这条边的权值就行了. 第二种的计算方式, 就相当于给我们了很多条边, 让我们在里面任取两条, 使得拼出来的最大. 这样子我们肯定选最大的一个和次大的一个拼起来. 这就指示我们用最大值和次大值相加得到这个问题的解答.

如何求最大值和次大值呢? 我们可以使用一个循环的方式, 每次先更新最大值, 最大值更新了之后把最大值给次大值就行了. 下面来看代码:

注意搜索的时候不能向上搜索, 只能向下搜索, 否则会出现死循环. 于是 dfs 的时候可以加上一个参数 father. `◇C++diameter.cpp`

如果是无权重的边, 有如下的算法亦可以做:

- 任取一点作为起点, 距离该点找一个最远的点 u ;
- 再找距离 u 最远的一点 v , 此时, u, v 之间的路径就是一条直径.


这个算法并不直观. 我们给出证明:

PROOF 我们只要证明第一步找出的 u 一定是某个直径的起点. 假设任选一点 a , 距离 a 最远的其中一点为 u . 考虑反证法, 假设它不是一个直径的起点, 假设某一条真正的直径是 $b \rightarrow c$. 我们现在按照此算法得到的直径与真正的直径有如下的几种情形:

(1) 两个直径是不相交的: 由于连通性, 从 $a \rightarrow b$ 这条路径上面一定存在一个点可以 (不一定是一步) 走到 $b \rightarrow c$ 这条路径上. 这条路径交于刚刚的两条路径为 x, y . 由于 u 距离 a 最远, 因此 $\text{dis}(x, u) \geq \text{dis}(x, y) + \text{dis}(y, c)$. 并且做移项并放大, 有 $\text{dis}(x, u) + \text{dis}(x, y) \geq \text{dis}(y, c)$. 那就说明, 沿着 $y \rightarrow x \rightarrow u$ 的距离一定是大于 $y \rightarrow c$ 的距离, 所以更长的直径是 $b \rightarrow y \rightarrow x \rightarrow u$. 因此, u 一定是某一条直径的端点.

(2) 两个直径之间有公共的端点, 记作 x : 由于 u 是距离 a 最远的一点, 那么 $\text{dis}(x, u) \geq \text{dis}(x, c)$. 由于 $b \rightarrow c$ 是一条直径, $\text{dis}(b, u) \geq \text{dis}(b, c)$ 因此 $b \rightarrow u$ 是一个更长的直径, 我们的证明了 u 一定在某一个顶点上.


□

 **树的中心.** 题目大意: 给定一棵树, 树中有 n 个结点 (结点编号为 $1 \sim n$), 请求出该树的中心结点的编号. 树的中心指的是, 该结点离树中的其他结点, 最远距离最近. ($n \leq 10^5$)

一个节点距离最远的有哪些类呢? 首先可以是往下走, 其次可以是往上走的. 往下走的倒是很好说, 直接是刚刚存的 `dist`; 但是往上走的就不一样了. 往上走的话, 又可以分为两类: 一类是接着往上走; 另一类是往下走. 往下走又分为两类: 第一类是走过了当前点, 另一类是没有走过当前点.

往下走的情况, 对于走过了当前点而言, 那就加上次大值 (因为不能再走回去绕圈圈); 对于没有走过这个点而言, 那就加上最大值构成这个点的最优情形. 请看 `◇C++center-with-weight.cpp`.


对于本问题, 可以对上述代码稍加改动得到: `◇C++center-with-weight-outnum.cpp`.


 **LOJ10155 数字转换**. 我们发现可以用图的观点来建模这道题. 如果 x 可以转换为 y , 那么就把 x, y 之间连一条边. 由于每个数的约数之和是给定的, 那么每个点至多有一个父节点. 因此会得到一些树的集合 (注意不一定是一棵). 所以我们在这些树里面找到最长的路径就可以了.

关于枚举因数之和, 我们可以先枚举一个数, 再枚举哪些数是它的倍数, 这样就会快一些. 第一次要枚举 n 次, 第二次 $n/2$ 次, 第三次 $n/3$ 次. 于是总的枚举次数是

$$n \left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \right).$$

里面的 $\sum_{i=1}^n 1/i$, 当 $n \rightarrow \infty$ 的时候, 这个渐进复杂度是 $\ln n + \gamma$ 的¹⁴. 因此, 这个时间复杂度是 \ln 级别的. 剩下的细节请参看文件 `◇C++LOJ10155.cpp`


 **LOJ10153 二叉苹果树**. 我们定义 $f[i][j]$ 为以 i 为根的子树里面选择 j 条边的最大价值. 那么, 有哪些节点可以到达 $f[i][j]$ 呢? 肯定是要从它的子节点中找线索. 也就是我们需要考虑子节点 (例如叫 s), 选 0 个 ($f[s][0] + w$), 选 1 个 ($f[s][1] + w$), \cdots , 选 $j-1$ 个 ($f[s][j-1] + w$, 留一个给 j).

 **P2016 战略游戏**. 我们定义 $f[u][0]$ 表示为所有以 u 为子树中, 且不设置的最小值; 定义 $f[u][1]$ 是选择, 其余与 $f[u][0]$ 一致. 下面我们来看状态的计算:

有哪些来可以到这里呢? 看 $f[u][0]$: 注意到这一个集合的每一个方案都可以分成若干部分, 每一个部分代表每个子树. 根据每个子树的选法, 因为他们是独立的, 我们可以拼凑出整个稍微大一点的树的选法. 想让整个最小, 要让每一部分别最小. 所以就是 $\min\{f[s_1][1], f[s_2][1], \cdots, f[s_n][1]\}$.

如果是 $f[u][1]$, 那么就选了这个点, 就可选可不选. 那么就是 $\min\{f[s_1][1], f[s_1][0], f[s_2][1], f[s_2][0], \cdots, f[s_n][1], f[s_n][0]\}$.

下面我们来看代码: `◇C++P2016.cpp`

 **LOJ10157 皇宫看守**. 我们可以按照我们的需求来设计状态. 一般来说, 就是看一看当前节点有哪些情形. 我们发现, (0) 它可以被父节点看到; (1) 它可能在子节点上面设置; (2) 它可以自己上面就有一个士兵三种状态. 所以我们顺势设置三个状态: $f[i][0]$ 表示它可以被父节点看到所有摆放方案的最小花费; $f[i][1]$ 表示被子节点看到的所有摆放方案的最小花费, $f[i][2]$ 同理, 不过是在当前节点摆放的左右方案的最小花费.

考虑状态的计算: 第一种情况是 $f[i][0] = \sum_{j \in i's \text{ son}} \min\{f[j][1], f[j][2]\}$; 第二种情况是 $f[i][2] = \sum_{j \in i's \text{ son}} \min\{f[j][0], f[j][1], f[j][2]\}$; 最后一种情况是 $f[i][1]$. 这种情形比较麻烦, 因为我们不知道是哪一个节点看到的这个节点. 因此我们需要枚举出来放在哪一个上面去了, 取最小的出来. 也就是 $f[i][1] = \min_k \{f[k][2] + \sum_{j \neq k} \min\{f[j][1], f[j][2]\}\}$. (首先 k 放置警卫, 然后再在其他点不放置)

具体实现细节请看代码 `◇C++LP2016.cpp`.

闲聊与练习

著名数学教育家应先生: 听不懂比听懂好.

著名竞赛教练、国家队教练冷老师: “反应慢是上天给你的礼物”.

—韩涛¹⁵, 在朋友圈的一条推文.

¹⁴ 这是因为 $1 + 1/2 + \cdots + 1/n - \ln n$ 在 $n \rightarrow \infty$ 的时候是有极限的. 因为有极限的其中之一原则是随着 n 增大, 数列单调递减, 并且有下界 (不会走下某一个值). γ 大约是 0.57122 左右, 称为 Euler 常数.

¹⁵ 北大毕业, IMO 金牌教练. 学而思集训队 (星队) 创始人之一. 数十位学生进入数学国家集训队, 2 位学生获得 IMO 金牌.

终点固然令人向往, 这一路的风景更是美不胜收.

— 魏恒峰, 在南京大学编译原理课上

状态合并么? 你 GET dp 的本质了。实际上, 如果你去看计数问题, 相当多的 dp 都是找到一个好的特征, 然后把集合划分成同构/存在某种数量映射关系的子集, 然后只算少量几次。特征就是关键

— 蒋炎岩, 在一次聊天中

Takeaway Messages:

- (!) 知道了动态规划其实就是对于有**最优子结构 (也就是我们一直在用最优的小问题试图拼最优的大问题)** 问题设计的更聪明地枚举 (设计好重叠的小情形);
- (!) 注意到了记忆数组和遍历顺序的重要性;
- 认识了不同种类的动态规划类型, 感受到了划分集合的重要性;
- 发现了其实和上一节一些计数问题的解决遵循相同的规则.

A 组问题

一. 回答下列小问题

- 简单使用动态规划解决问题的一些特征.
- 使用迭代的方式和记忆化搜索的方式给你的感觉有什么相同和不同?

数论简介

SECTION 17

整数, Euclid 算法

PART

VII

任何一本书上面的开头好像都喜欢用 Euclid 算法求解最大公约数开场. 这是一个十分古老的算法, 但是要是仔仔细细证明这个算法, 还是不那么显然的.

下面我们来借助这个算法, 来简单回顾一些数论的基本概念.

要求两个数 m, n 的最大公因数 (**greatest common divisor, gcd**), Euclid 发现了这样一个算法并声称: $\text{gcd}(m, n) = \text{gcd}(n \% m, n)$. 这里, 沿用 C++ 里面的取模, $a \% b$ 的值就等于 a/b 的余数.

如果我们用更聪明的记号来表示的话, 或许 $a \% b = a - [a/b] \times b$. 就相当于模拟了 b 次减法. $[x]$ 表示 x 下取整. 比如 $[\pi] = 3$. 这是一个很有趣的小符号, 它实际上表示的是一个不等式的关系.

因数的概念可能是数学中产生的最自然的概念之一. 当我们要对一个东西平均的分配的时候, 一个数”能被另一个数整除”这个性质就显得尤其重要. 这里, 我们发现一些数只能被 1 和这个数本身整除, 这样的数我们一般叫做“**质数 (prime)**”. 比如 $3 = 1 \times 3$. (其实 $3 = -1 \times -3$, 但是这里我们只考虑这些乘数都是正的). 还有一些数分解的可能就多了不少. 例如 $14 = 1 \times 14 = 2 \times 7$. 这样的数我们称作“**合数 (composite)**”.

质数与合数
定义

Definition 16 (质数与合数) 若一个正整数只含有 1 和它本身这两个不同的正因数, 则称其为素数. 若一个正整数出了 1 和它本身外, 还有其他的因数, 则称为合¹⁶数. 1 既不是素数, 也不是合数.

¹⁶ 合指的是“复合”的意思.

有时候在处理一些证明的时候, 如果题目中告诉了我们 n 是一个合数, 我们可以知道 $n = n_1 n_2$, 其中 $1 \leq n_1, n_2 \leq n$. 这样子有时候可以帮助我们分析问题.

更一般的, 有时候我们仅仅关注一个数能不能被另一个数整除, 也即是 a/b 的余数是不是为 0, 如果是, 我们就说 b 是 a 的一个**因子 (factor)**. 有时候可以写作 $b|a$. 像是从 a 中”抓出来”了一个它的更小的部分放在前面.

所以我们有一个形式化的定义:

Definition 17 (整除) $m|n \iff m > 0$ 并且对于某个整数 $k, n = mk$.

如果我们抓一个数出来写出它的所有因子, 就会发现因数具有共轭的属性. 意味着 $d|n \iff \frac{n}{d}|n$. 这个问题我们在后续学习 Mobius 反演系列的内容的时候会频繁地使用这个内容变换和的下标.

在知道为什么这个算法是对的之前, 我们需要发掘一点整除的性质.

整除的性质
简介

Theorem 5 设 $a, b, c \in \mathbb{Z}, c \neq 0$, 我们有如下的性质:

- $c|b, b|a \implies c|a$. (整除的传递性)
- $b|a \implies cb|ca$.
- $c|a, c|b, \implies \forall m, n \in \mathbb{Z}$, 有 $c|(ma + nb)$. (线性组合, 非常重要)
- $b|a$, 并且 $a \neq 0 \implies |b| \leq |a|$.
- $b|a$, 并且 $|b| > |a| \implies a = 0$.
- $a|b, b|a \implies |a| = |b|$.

PROOF 对于 (4) 的证明: 因为 $b|a$, 所以存在 $k \in \mathbb{Z}$, 使得 $a = kb$. 由于 $a \neq 0$, 意味着 $k \neq 0$. 所以 $|a| = |kb| = |k||b| \geq |b|$ \square

带余除法 介绍

我们看到了整除的理论, 接下来我们来看带余数的除法有什么值得关注的.

Theorem 6 (带余除法) 对于给定整数 a, b , 且 $b \neq 0$, 必定存在一对整数 q, r , 使得 $a = bq + r$, $0 \leq r < |b|$.¹⁷

PROOF 存在性: 分类讨论. (1) 考虑 $b|a$, $\exists k \in \mathbb{Z}$, 使得 $a = kb$, 取 $q = k, r = 0$ 即为答案. (2) 若 $b \nmid a$, 取这样的集合 $S = \{a - bq : q \in \mathbb{Z}\}$. 其中 a, b 是给定的整数, q 是可以变化的整数. 显然 S 中存在正整数. 必有最小者. 记它为 r . 下证 $0 < r < |b|$. 考虑反证法, 如果这个不成立, 那么 $r \geq |b|$. (i) 若 $r = |b|$, $a - bq = |b| \implies b|a$, 矛盾! (ii) $r > |b| : 0 \leq r - |b| = a - bq - |b| = a - b(q \pm 1) \in S$, 但是 $r - |b| < r$, 矛盾. 唯一性: 假设存在两对整数 q_1, r_1, q_2, r_2 , 使得 $a = bq_1 + r_1 = bq_2 + r_2$, 移项, $b(q_1 - q_2) = r_2 - r_1 \implies b|(r_2 - r_1)$, 并且 $|r_2 - r_1| < |b|$, 得到 $r_2 = r_1, q_2 = q_1$. \square

上面存在性 (2)(ii) 的内容被称为递降法. 我们想要证明正整数集 S 里面是空集, 可以从它的反面, 即正整数集 S 非空, 其中必有 S 中的最小元素 x_0 . 但是我们发现某一个 $f(x_0) < x_0$, 并且 $f(x_0) \in S$. 这就表明出现了矛盾. 其实 S 就是空集.

这就可以让我们用 (a, b) 的问题通过做带余除法的方式, 转化为 (b, r) 的化大为小的操作.

在介绍我们今天的主角 Euclid 算法之前, 我们先来看一组关于公因数的相关的概念.

Definition 18 (公因数 (common divisor) 和公倍数 (common multiple)) 设 D, d, M, m 以及 $a_1, a_2, \dots, a_n \in \mathbb{N}_+$:

- (1) 若 $d|a_i, i = 1, 2, \dots, n$, 则称 d 是 a_1, a_2, \dots, a_n 的一个公因数. 若 D 是 a_1, \dots, a_n 的一个公因数, 且对于 a_1, a_2, \dots, a_n 的任一公因数 $d|D$, 则称 D 是 a_1, a_2, \dots, a_n 的最大公因数¹⁸. 记作 (a_1, a_2, \dots, a_n) 或 $\gcd(a_1, a_2, a_3, \dots, a_n)$.
- (2) 若 $a_i|M, i = 1, 2, 3, \dots, n$, 则称 M 是 a_1, a_2, \dots, a_n 的一个公倍数. 若 M 是 a_1, \dots, a_n 的一个公倍数, 且对于 a_1, a_2, \dots, a_n 的任何一个公倍数 M , 都有 $m|M$, 则称 m 是 a_1, a_2, \dots, a_n 的最小公倍数. 记作 $[a_1, a_2, \dots, a_n]$, 或这 $\text{lcm}(a_1, a_2, \dots, a_n)$.
- (3) 若 $(a_1, a_2, \dots, a_n) = 1$, 那么称 a_1, a_2, \dots, a_n 互素.

¹⁷ 为什么还要证明这样显然的东西? 事实上, 小学时候我们做的事情只是一种给定的情况. 我们很多时候需要做一些并不显然的东西. 这里的任意性导致我们必须证明这样的定理. 数学不是“实验科学”.

¹⁸ 这个与所有的公因数中的最大者是等价的. 因为我们有整除的性质中的不等式可以知道.

在解题中我们经常碰见的是两个数互素. 通俗来讲, 当两个数没有公共的素因子的时候, 就称为他们是互素的. 另外, 一个更强的条件是两两互素的. 整体互素只要保证他们每一个是没有素因子的, 两两互素要求任意的两个都没有素因子.

Euclid 算法

简介

欧几里得算法声称, 我们如果要求 (a, b) 的最大公约数, 我们可以对它做带余除法. 也就是 $a = bq + r (0 \leq r < b)$, 然后我们就只要求 (b, r) 的最大公因数就行了.

Theorem 7

(Euclid 算法原理) 若 $a, b \in \mathbb{N}_+$, $a = bq + r (0 \leq r < b)$, 那么 $\gcd(a, b) = \gcd(b, r)$.

PROOF

容易说明 $\gcd(a, b) = \gcd(a - b, b)$. 因为如果 d 是 a 的因数, 也是 b 的因数, 那它一定是 a 和 b 的一个公因数. 根据整除的性质, d 一定是 $a - b$ 的因数. 因此 d 就是 $a - b$ 和 b 的公因数. 反之, 如果一个数是 $(a - b)$ 的公因数, 同时也是 b 的公因数, 我们很容易推出是 a 的公因数. 它们是一一对应的. 自然, 他们的最大公因数是相等的. 由此, $(a, b) = (a - b, b) = (a - 2b, b) = \cdots = (a - qb, b) = (b, r)$. \square

有了上述的原理, 我们就可以用这样的手段不断化大为小, 直到第二个位置为 0. 这时候第一个位置就是我们的最大公因数. 下面只写余数不为 0 的情况: 这时候,

$$\begin{aligned} b &= r_1q_1 + r_1 (0 \leq r_1 < r) & 0 \leq r_1 < r, (b, r) &= (r, r_1) \\ r &= r_1q_2 + r_2 (0 \leq r_2 < r_1) & 0 \leq r_2 < r_1, (r, r_1) &= (r_1, r_2) \\ r_1 &= r_2q_3 + r_3 (0 \leq r_3 < r_2) & 0 \leq r_3 < r_2, (r_1, r_2) &= (r_2, r_3) \\ &\dots \end{aligned}$$

这个方法可以无限地进行下去吗? 其实是不能的. $b > r > r_1 > r_2 > \cdots \geq 0$, 这一一定是有限的. 必有一步余数是 0. 我们记作这一步为 r_{n+1} . 也就是 $r_{n-1} = r_nq_{n+1}$, 意味着 $(r_{n-1}, r_n) = r_n$. 综上所述, $(a, b) = (b, r) = (r, r_1) = \cdots = (r_{n-1}, r_n) = r_n$.

用这个方法可以求一系列有趣的例子. 如 Fibonacci 数列中的一个有趣的性质:

Example

Fibonacci 数列是 $F_1 = 1, F_2 = 1, \forall n \in \mathbb{N}_+$ 都有 $F_{n+2} = F_{n+1} + F_n$. 可以证明 $(F_m, F_n) = F_{(m,n)}$.

仿照刚刚 Euclid 算法证明的过程, 我们可以尝试证明 $(F_m, F_n) = (F_{n-m}, F_m)$. 当 $n = m$ 的时候显然成立. 只研究 $n > m$ 的情形. $n = mq + r, 0 \leq r < m$.

而使用 “1” 的代换, 因为 $F_1 = 1, F_2 = 1$, 可以乘上去并且对 F_{n-1} 再用一次. 有 $F_n = F_{n-1} + F_{n-2} = F_2F_{n-1} + F_1F_{n-2} = F_2(F_{n-2} + F_{n-3}) + F_1F_{n-2}$. 也就是 $F_2F_{n-2} + F_2F_{n-3} + F_1F_{n-2}$, 提取公因式, 就有 $F_3F_{n-2} + F_2F_{n-3}$. 继续这样做下去, 写作 $F_3(F_{n-3} + F_{n-4}) + F_2F_{n-3}$. 我们同样可以用同样的方法写作 $F_4F_{n-3} + F_3F_{n-4}$, 一直做下去, 最后可以得到 $F_mF_{n-m+1} + F_{m-1}F_{n-m}$. 这里就出现了我们期待已久的 F_m, F_{n-m} 的结构了, 如果 $d|F_m, d|F_n \implies d|F_{m-1}F_{n-m}$. 我们只要证明 F_m, F_{m-1} 是互素的. 根据 Fibonacci 的性质 $(F_m, F_{m-1}) = (F_{m-1}, F_{m-2}) = \cdots = (F_2, F_1) = 1$ 不难看出.

反之, $d|F_{n-m}, d|F_m \implies d|F_m, d|F_n$. 我们证得 $(F_m, F_n) = (F_{n-m}, F_m)$. 持续地辗转相除, 即可得到.

Bezout 定

理

简介

TBD...Bezout 定理可以帮助我们解决这样一类整数不定方程的问题:

Theorem 8 设 $d = (a, b)$, 则存在 $x, y \in \mathbb{Z}$, 使得 $xa + yb = d$.

PROOF 不妨设 $a > b > 0$, 则 $a = bq + r, 0 \leq r < b$. 只考虑除了最后一步的各个步骤, 余数大于 0 的情形. 由于 $(a, b) = (b, r)$. 根据带余除法的过程, 有:

$$\begin{aligned} b &= r_1q_1 + r_1 & 0 \leq r_1 < r, \\ r &= r_1q_2 + r_2 & 0 \leq r_2 < r_1, \\ r_1 &= r_2q_3 + r_3 & 0 \leq r_3 < r_2, \\ &\dots \\ r_{n-3} &= r_{n-2}q_{n-1} + r_{n-1} & 0 \leq r_{n-1} < r_{n-2} \\ r_{n-2} &= r_{n-1}q_n + r_n & 0 \leq r_n < r_{n-1} \\ r_{n-1} &= r_nq_{n+1} \end{aligned}$$

根据 $d = (a, b) = (b, r) = (r, r_1) = (r_1, r_2) = \dots = (r_{n-1}, r_n) = r_n$, 我们倒推回去

$$\begin{aligned} d &= r_n = r_{n-2} - r_{n-1}q_n \\ &= r_{n-2} - (r_{n-3} - r_{n-2}q_{n-1}) \\ &= -q_n r_{n-3} + (1 + q_{n-1}q_n)r_{n-2} \\ &\dots \\ &= xa + yb, (x, y \in \mathbb{Z}). \end{aligned}$$

□

从上面的内容可以看到, d 可以写作辗转相除过程中, 任意相邻两步余数的线性组合. 那么这个问题的逆命题成立吗? 这个显然是没有逆命题的. 上面这个定理的逆命题是若 $d = (a, b)$, 则存在 $x, y \in \mathbb{Z}$, 使得 $xa + yb = d$. 把这个式子乘上 k , 得到 $(kx)a + (ky)b = kd$, 这就说明有无数个可以表示为 a, b 的整系数线性组合的数. 它们不可能都是 a, b 的最大公因数. 但是我们可以附加一个条件让我们的这个逆命题成立.

使得 $xa + yb = (a, b)$ 成立的 x, y 有多少组呢? 其实是有无数组. 只要我找到了其中的一组, 比如叫做 x_0, y_0 , 那么我们就可以在 x_0 上面加上若干倍的 kb , 在 y_0 上面减掉若干倍的 ka , 形成 $(x_0 + kb)a + (y_0 - ka)b$ 的形式. 这个同样是满足原来的式子的.

将上述形式稍加修改, 我们就得到了这个定理具有逆定理的形式.

Theorem 9 (充分必要的 Bezout 定理) $(a, b) = d \iff d|a, d|b$, 并且 $\exists x, y \in \mathbb{Z}$, 使得 $xa + yb = d$.

PROOF 必要性上述已经证明, 下面证明充分性.

我们先证明一个引理: (a, b) 是形如 $xa + yb (x, y \in \mathbb{Z})$ 的正整数中的最小者. 记 $S = \{xa + yb : x, y \in \mathbb{Z}\}$, $l_0 = x_0a + y_0b$ 是 S 中的最小者. 下面证明 $l_0 = (a, b)$. 从 S 中取出

任意的一个 $l = xa + yb$, 若记 $l_0 | l$, 由于 l_0 是正数, 根据带余除法, $l = l_0 q + r, 0 \leq r < l_0$.

$$r = l - l_0 q = (xa + yb) - (x_0 a + y_0 b)q = (x - x_0 q)a + (y - y_0 q)b \in S.$$

但是 $r < l_0$, r 只能等于 0. 意味着 l_0 是 l 的因数. 即 $l_0 | l$

下面证明 l_0 是 (a, b) . 一方面, $(a, b) | a, (a, b) | b \implies (a, b) | (x_0 a + y_0 b)$, 也就是 a, b 是 l_0 的因数.

另一方面, 由于 $a, b \in S \implies l_0 | a, l_0 | b \implies l_0 | (a, b)$.

由于上述两方面, $l_0 = (a, b)$. 引理证明完毕.

下面证明 l_0 就是 d . 根据上述的引理, $(a, b) | (xa + yb) = d$, 又因为条件中的 $d | a, d | b \implies d | (a, b)$

所以 $d = (a, b)$. \square

事实上, Bezout 定理可以推广到 n 个正整数的最大公因数的情形. $\forall a_1, a_2, \dots, a_n \in \mathbb{N}_+, \exists k_1, k_2, k_3, \dots, k_n \in \mathbb{Z}$, 使得 $k_1 a_1 + k_2 a_2 + \dots + k_n a_n = (a_1, a_2, \dots, a_n)$. 另外, 由于系数不唯一可能会导致一些研究的困难. 这时候我们对于系数做一些限定:

Theorem 10 设 $a, b \in \mathbb{Z}, (a, b) = 1$, 且 $|a| \geq 2, |b| \geq 2$, 则 $\exists u_0, v_0 \in \mathbb{Z}$, 使得 $u_0 a + v_0 b = 1$, 且 $|u_0| < |b|, |v_0| < |a|$.

PROOF 根据 Bezout 定理, 存在 $u, v \in \mathbb{Z}$, 使得 $ua + vb = 1$. 一定会有一个 $u = qb + u_0$, 其中 $0 \leq u_0 < |b|$, $1 = ua + vb = (qb + u_0)a + rb = u_0 a + (aq + v)b$. 令 $v_0 = aq + v$, 记作 $u_0 a + v_0 b$. 我们只要证明 $|v_0 b| < |a|$ 即可.

我们发现 $|v_0 b| = |1 - u_0 a| \leq 1 + |u_0 a| = 1 + u_0 |a|$. 由于 $1 < |a|$, 我们就知道 $1 + u_0 |a| < |a| + u_0 |a|$. 也就是 $(1 + u_0)|a|$. 由于 $u_0 < |b|, 1 + u_0 \leq |b|$, 那么继续放大为 $|b| \cdot |a|$, 得到 $|v_0| < |a|$. \square

这个定理可以帮助我们求解一部分不定方程的求解问题. 我们下面来看这个例子:

Example 关于 x, y 的不定方程 $ax + by = c (a, b, c \in \mathbb{Z}, a, b$ 不全为 0), 它有整数解的充要条件是 $(a, b) | c$.

对于此的证明, 必要性是显然的, 设 $d = (a, b)$, 则 $d | ax, d | by, d | ax + by \implies d | c$.

充分性: $(a, b) | c \implies \exists k \in \mathbb{Z}$, 使得 $c = k(a, b)$. 由 Bezout 定理, 存在 $x_0, y_0 \in \mathbb{Z}$, 使得

$$x_0 a + y_0 b = (a, b).$$

将上述的内容两边同时乘上 k , 有 $kx_0 a + ky_0 b = k(a, b) = c$, 所以 kx_0, ky_0 是方程的一组整数解.

算术基本定理 算术基本定理, 也被称为质因数分解定理, 是数论中的一个重要结果. 它告诉我们, 每个大于 1 的整数都可以唯一地表示为一系列质数的乘积, 而且这个表示方式是唯一的.

介绍

TBD: 一个无穷维的空间

Theorem 11 (算术基本定理) 设 n 是一个大于 1 的正整数, 则它可以写成 $n = p_1 p_2 \dots p_k$, 其中 $p_i (1 \leq i \leq k)$ 都是素数. 且在不计次序的情况下, 该表达式是唯一的.

这个定理表明了每一个自然数的分解都是唯一的. 下面我们来证明. 分为存在性和唯一性两方面.

PROOF 存在性: 设 $n > 1$, 且 $n \in \mathbb{N}_+$, 则 n 的最小的大于 1 的素数 p 必为素数. 若 $p_1 = n$, 则 n 为素数. 否则 $p_1 < n$, 则 n/p_1 仍然为一个大于 1 的整数. 则 n/p_1 的最小的大于 1 的因数记作 p_2 必为素数. 若 $p_2 = n/p_1$, 则 n/p_1 为素数, 也就是 $n = p_1 p_2$. 若 $p_2 < n/p_1$, 则可对 $n/(p_1 p_2)$ 重复前面的推理, 直到素数 $p_k = \frac{n}{p_1 p_2 \cdots p_k}$, 唯一性: 我们假设还有另外一个表达式 $n = q_1 q_2 \cdots q_m$. 其中 $q_j (1 \leq j \leq m)$ 都是素数. 不妨设 $p_1 \leq p_2 \leq \cdots \leq p_k, q_1 \leq q_2 \leq \cdots \leq q_m$. 意味着 $q_1 | p_1 p_2 \cdots p_k$. 回顾这样的定理: 如果素数 $p | ab$, 那么 $p | a$ 或 $p | b$ (可由 Bezout 定理推出), 那么 q_1 一定是某一个 $p_i (1 \leq i \leq k)$, 使得 $q_1 | p_i$. 根据素数的定义, $q_1 = p_i$. 同理, $\exists q_j \in q_1, q_2, \cdots, q_m$, 使得 $q_j = p_1$. 因为 $p_1 = q_j \geq q_1 = p_i \geq p_i$. 自然而然, $p_1 = q_1$. 则在等式两边同时约去 p_1 , 重复上述推理. 得 $p_2 = q_2, p_3 = q_3, \cdots$ 直到 $q_{k+1} q_{k+2} \cdots q_m = 1$, 这在 $k < m$ 的条件下不成立. 必然 $k = m$. 唯一性得到证明. \square

我们可以把相同的数合并起来写在指数的位置, 记作 $n = p_1^{a_1} p_2^{a_2} \cdots p_m^{a_m}$, 每一个 $a_i \in \mathbb{N}_+$. 如果 $d | n$, 那么就说明 d 可以写作 $p_1^{b_1} p_2^{b_2} \cdots p_m^{b_m}$ 的形式, 每一个 $b_i, i \in \{1, 2, \cdots, m\}$ 是 a_1, a_2, \cdots, a_m 的其中一个.

那么有了这个, 我们简单看一看正因数的个数有几个. 假设有一个数 $n = p_1^{a_1} p_2^{a_2} \cdots p_m^{a_m}$, 它的正因数有几个呢? 根据选法, 第一个 p_1 我们可以选择 0 个, 1 个一直到 a_1 个, 共 $(a_1 + 1)$ 个选法. 第二个 p_2 我们可以选择 0 个, 1 个一直到 a_2 个, 共 $(a_2 + 1)$ 个选法. 如此继续, 做到 p_k 才算是做完了. 我们发现一个数的正因数有

$$d(n) = (a_1 + 1)(a_2 + 1) \cdots (a_m + 1) = \prod_{i=1}^m (1 + a_i).$$

那么, 一个数 $n = p_1^{a_1} p_2^{a_2} \cdots p_m^{a_m}$, 的正因数之和为多少呢? 我们可以通过素因子把它划分到如下的几个集合里面:

$$\{1, p_1, p_1^2, \cdots, p_1^{a_1}\}, \{1, p_2, p_2^2, \cdots, p_2^{a_2}\}, \cdots, \{1, p_m, p_m^2, \cdots, p_m^{a_m}\},$$

n 的每一个正因数相当于在这 m 个集合中各自选择一个因数, 把它乘到一起. 那我们把所有的选法求和, 就是求出了所有正因数的和. 我们根据多项式的乘法法则, 就可以说明正因数的和 $\sigma(n)$ 是

$$\sigma(n) = \prod_{i=1}^m (1 + p_i + p_i^2 + \cdots + p_i^{a_i}).$$

有了这样的理论, 我们就可以对于公因数和公倍数有一个更加直观的视角了. 因为这个分解是唯一的,

(最大公因数和最小公倍数与唯一分解定理) 设 $a_i = p_1^{a_{i1}} p_2^{a_{i2}} \cdots p_k^{a_{ik}}, i = 1, 2, \cdots, n$. 并且 p_1, p_2, \cdots, p_k 是不同的素数, $a_i \in \mathbb{N}$. 令 $t_i = \min_{1 \leq j \leq n} a_{ij}, s_i = \max_{1 \leq j \leq n} a_{ij}$, 那

公因数与公
倍数

算术基本定

理的视角
Theorem 12

么

$$(a_1, a_2, \dots, a_n) = p_1^{t_1} p_2^{t_2} \cdots p_k^{t_k};$$

$$[a_1, a_2, \dots, a_n] = p_1^{s_1} p_2^{s_2} \cdots p_k^{s_k}.$$

Theorem 13

(1) 若 $d|a_i, i = 1, 2, 3, \dots, n$, 则 $d|(a_1, a_2, \dots, a_n)$.

(2) 若 $a_i|A, i = 1, 2, 3, \dots, n$, 则 $[a_1, a_2, \dots, a_n]|A$.

(3) 若 $m \in \mathbb{N}_+$, 那么

$$(ma_1, ma_2, \dots, ma_n) = m(a_1, a_2, \dots, a_n),$$

同样有

$$[ma_1, ma_2, \dots, ma_n] = m[a_1, a_2, \dots, a_n].$$

(4) 可以通过化归转化的方法求解一组数的最大公因数:

$$(a_1, a_2, a_3 \cdots, a_n) = ((a_1, a_2), a_3 \cdots, a_n),$$

同样有

$$[a_1, a_2, a_3 \cdots, a_n] = [[a_1, a_2], a_3 \cdots, a_n].$$

Theorem 14

$$ab = (a, b)[a, b].$$

PROOF

我们可以使用两边夹的方法说明这件事. 回忆定理若 $a, b \in \mathbb{N}, a|b, b|a \implies a = b$.

我们设 $c = \frac{ab}{(a, b)}$, 由于 $(a, b)|a, (a, b)|b$, 一定存在 k_1, k_2 , 使得 $a = k_1(a, b), b = k_2(a, b), k_1, k_2 \in \mathbb{Z}$.

根据前面的设定, $c(a, b) = ab = k_1 k_2 (a, b)^2$. 约去 (a, b) . 于是 $c = k_1 k_2 (a, b)$. 运用 a, b 的定义, 有 $c = k_1 k_2 (a, b) = a k_2 = b k_1$. 使用整除的定义, $a|c, b|c$. 根据上一个定理 (2), $[a, b]|c$. (1)

根据 Bezout 定理, 一定存在整数 x, y , 使得 $xa + yb = (a, b)$. 把等式两边同时除以最大公因数得到了

$$\frac{a}{(a, b)}x + \frac{b}{(a, b)}y = 1$$

也就是 $k_1 x + k_2 y = 1$. 同时乘以 $[a, b]$ 有:

$$k_1 [a, b]x + k_2 [a, b]y = [a, b].$$

做替换, 把 k_1 替换为 $c/b, k_2$ 替换为 c/a , 得到

$$\frac{c}{b}[a, b]x + \frac{c}{a}[a, b]y = [a, b].$$

提取公因式 c , 因此有

$$c \left(\frac{[a, b]}{b} x + \frac{[a, b]}{a} y \right) = [a, b].$$

因此 $c|[a, b]$. (2)

根据 (1), (2) 可得 $c = [a, b]$. 得证. \square

这个定理可不可以向多个整数推广呢? 这个是不可以的. 因为

$$[a, b, c] = [[a, b], c] = \left[\frac{ab}{(a, b)}, c \right] = \frac{\frac{abc}{(ab)}}{\left(\frac{ab}{(a, b)}, c \right)} = \frac{abc}{ab, c(a, b)} = \frac{abc}{(ab, bc, ca)}.$$

我们取两个正整数, 他们互素的概率多大? 设 p_1, p_2, \dots 是从小到大的素数. 则 a 是 p_i 的概率是 $1/p_i$, 同理 b . 因此 a, b 有公共素因子 p_i 的概率为 $1/p_i^2$. 因此, a, b 没有任何公共素因子的概率为¹⁹

$$\prod_{i=1}^{\infty} \left(1 - \frac{1}{p_i^2} \right).$$

¹⁹ 这个并不直观. 实际上, 我们还需要定义可数无穷的的概率的定义. 不过这份资料就不做定义了. 感兴趣可以参考概率相关的参考书.

要计算它可能要一点数学分析的知识, 我们直接给出概率的结果: $6/\pi^2$.

SECTION 18

同余

同余简介

同余启发我们按照余数分类. 我们以前看到过奇偶分析, 这个就是相当于按照模 2 的余数分类. 另外, 同余是一种有趣的语言. 下面我们来看一看什么是同余:

Definition 19 若 $a, b \in \mathbb{Z}$ 除以 M 所得的余数相同, 则称 a, b 对模 M 同余. 记作 $a \equiv b \pmod{m}$. 否则, 称为二者不同余. 记作 $a \not\equiv b \pmod{m}$.

我们并不关心商是多少, 我们只关心余数是多少.

下面我们给出 $a \equiv b \pmod{m}$ 的充要条件:

Theorem 15 $a \equiv b \pmod{m} \iff m|(a - b)$.

PROOF

$$a \equiv b \pmod{m} \iff \exists k_1, k_2 \in \mathbb{Z},$$

使得 $a = k_1 m + r, b = k_2 m + r, 0 \leq r \leq m$. 根据同余的定义得到, $a - b = (k_1 - k_2)m$, 根据整除的定义, $m|(a - b)$. \square

那么, 同余有哪些性质? 和整除理论一样, 我们发现:

(1) $a \equiv m \pmod{b}, c \equiv d \pmod{m} \implies a \pm c \equiv b \pm d \pmod{m}$. 等式的基本性质可以就可以放在这里了. 我们可以用同余的定义和同余的充要条件证明之.

(2) $a \equiv b \pmod{m}, c \equiv d \pmod{m} \implies ac \equiv bd \pmod{m}$. 和上一个问题如出一辙.

(3) 这是一个有趣的性质: $a \equiv b \pmod{m} \implies a^n \equiv b^n \pmod{m}, n \in \mathbb{N}_+$. 这个可以通过性质 (2) 不断作用 n 次得到.

$$(4) ac \equiv bc \pmod{m} \implies a \equiv b \pmod{\frac{m}{(m,c)}}.$$

PROOF 设 $(m, c) = d$, 则 $m = dm_1, c = dc_1$, 满足 $(m_1, c_1) = 1$.

$$\begin{aligned} ac \equiv bc \pmod{m} &\iff m|c(a-b) \iff dm_1|dc_1(a-b) \iff m_1|c_1(a-b) \\ &\implies m_1|(a-b) \iff a \equiv b \pmod{m} \end{aligned}$$

也即是 $a \equiv b \pmod{\frac{m}{(m,c)}}$. □

这个暗示了“约分”这一行为在模算术的做法.

(5) 如果 $a \equiv b \pmod{m}, n|m, n \in \mathbb{N}_+ \implies a \equiv b \pmod{n}$. 这个表示了如果有一个数同余, 它一定和这个数的因子同余.

PROOF 根据定义, 我们有:

$$\begin{aligned} n|m &\iff m = kn, k \in \mathbb{Z} \\ a \equiv b \pmod{m} &\iff m|(a-b) \iff kn|(a-b) \\ &\implies n|(a-b) \iff a \equiv b \pmod{n} \end{aligned}$$

□

(6) 与之类似, 我们有 $a \equiv b \pmod{m}, a \equiv b \pmod{n} \implies a \equiv b \pmod{[m, n]}$.

(7) 如果我们用 $\mathbb{Z}[x]$ 代表整系数多项式, 那么对所有整数 $a, b (a \neq b)$, 有 $(a-b)|f(a)-f(b)$. 也就是说若 $a \equiv b \pmod{n}$, 对于某个 $n \in \mathbb{Z}$ 成立, 则 $f(a) \equiv f(b) \pmod{n}$.

PROOF 由于因式分解的相关知识, 我们知道 $a-b|(a^m-b^m)$, 设 $f(x) = c_k x^k + c_{k-1} x^{k-1} + \dots + c_1 x + c_0$, $c_i \in \mathbb{Z}, i = 0, 1, 2, \dots, k$, 那么 $f(a) - f(b) = c_k(a^k - b^k) + \dots + c_1(a - b)$, 根据之前的结论, 因为 $a \equiv b \pmod{n}$, 我们知道 $n|(a-b) \implies n|(a^m - b^m)$, 于是得证. □

同余 我们如果把余数相同的集合放在一起看会怎么样? 如果我们把整数按照这个方面划分成若干类会如何? 下面我们来介绍同余类的概念.

Definition 20 (同余类) 设 m 是大于 1 的给定的正整数, 则可将整数集划分为 n 个子集, 记作 k_0, k_1, \dots, k_{m-1} , 其中, $k_r = \{x | x \equiv r \pmod{m}, x \in \mathbb{Z}\}, r = 0, 1, \dots, m-1$. 称 k_0, k_1, \dots, k_{m-1} 为同余类, 简单记作 $[0], [1], \dots, [m-1]$.

我们可以证明这是一个划分, 自然满足划分的两个性质. 根据定义, 两个数同属于同一个同余类, 当且仅当他们对 m 同余. 也就是 $a, b \in K_r \iff a \equiv b \pmod{m} \iff m|(a-b)$.

Definition 21 (完全剩余系) 若整数 a_0, a_1, \dots, a_{m-1} 中没有任何两个数同属于模 m 的一个同余类, 则称 a_0, a_1, \dots, a_{m-1} 构成模 m 的一个完全剩余系.

这就相当于 k_0, k_1, \dots, k_{m-1} 这 m 个同余系中, 每个同余类中取一个代表, 构成的

数组.²⁰ 这是一个整体思想. 虽然我们并不知道他们模 m 的结果具体是多少, 但是我们可以“筛选”某些特征, 使得我们可以让这个集合是定下来的.

这里有一些明显的性质, 我们说一下:

(1) 如果 m 个整数构成的模 m 的完全剩余系 \iff 这 m 个数模 m 两两不同余.

(2) 通过乘常数和加常数构造新的完全剩余系: 如果 a_0, a_1, \dots, a_m 是模 m 的一个完全剩余系, $a, b \in \mathbb{Z}$, 且 $(a, m) = 1$, 那么 $aa_0 + b, aa_1 + b, \dots, aa_{m-1} + b$ 也是模 m 的完全剩余系.

(3) 通过两个完全剩余系构造: 若 $(m, n) = 1$, a_1, a_2, \dots, a_m 和 b_1, b_2, \dots, b_n 是模 m 和模 n 的两个完全剩余系, 那么 $\{na_i + mb_j : 1 \leq i \leq m, 1 \leq j \leq n\}$ 是模 mn 的完全剩余系. 要证明这个, 我们需要证明这个数组里面有 mn 个数, 并且模 mn 两两不同余.

PROOF 我们可以把数组列一个 m 行 n 列的表格, 因此一共有 mn 个数.

$$\begin{array}{cccc} na_1 + mb_1, & na_1 + mb_2 & \cdots & na_1 + mb_n \\ na_2 + mb_1, & na_2 + mb_2 & \cdots & na_2 + mb_n \\ \cdots & \cdots & \cdots & \cdots \\ na_m + mb_1, & na_m + mb_2 & \cdots & na_m + mb_n \end{array}$$

假设存在 $a, a' \in \{a_1, a_2, \dots, a_m\}, b, b' \in \{b_1, b_2, \dots, b_n\}$, 且 $(a, b) \neq (a', b')$, 使得 $na + mb \equiv na' + mb' \pmod{mn}$.

$$\begin{aligned} &\implies mn | n(a - a') + m(b - b') \\ &\implies m | n(a - a') + m(b - b') \implies m | n(a - a') \\ &\implies m(a - a') \implies a \equiv a' \pmod{m}, \end{aligned}$$

与 m 来自完全剩余系矛盾. 因此, $a = a'$, 同理, $b = b'$. □

简化剩余系 (缩系) 简介

既然我们可以按照这样的方式取, 我们还可以用什么样的方式取? 一个想法是可以使用取和这个数互素的这些数.

Definition 22

设 K_r 是模 m 的一个同余类, 且 $(r, m) = 1$, 则称 K_r 为与 m 互素的同余类. 从每一个与 m 互素的同余类中各取一个数, 组成的数组叫做模 m 的简化剩余系.

我们发现, 一个模 m 互素的剩余类中, 每个数都与 m 互素. 因为 $a \in K_r, (r, m) = 1$, 对 a 做带余除法, $a = mq + r, 0 \leq r \leq m - 1$, 根据 Euclid 算法, $(a, m) = (m, r) = 1$. 更进一步地, 每个数都与模 m 互素.

那么, 这样一个简化剩余系 (缩系) 里面有多少个数? 这就是大名鼎鼎的欧拉函数的来源了. 模 m 的缩系可以看做一个模 m 中的一个完全剩余系中所有与 m 互素的数所构成的数组. 这个数有 $\varphi(m)$ 个数. 这就是 m 的欧拉函数.

Definition 23

如果记 $\#S$ 表示 S 集合中元素的数量, 那么欧拉函数可以定义为

$$\varphi(m) = \#\{k : 1 \leq k \leq m, (k, m) = 1\}.$$

²⁰ 后来如果你学习抽象代数, 你会发现这就可以引申为商集.

那么 φ 函数具有怎样的性质呢? 我们应该如何计算呢? 一个直接公式的方法如下:

Theorem 16

$$\varphi(n) = n(1 - \frac{1}{p_1})(1 - \frac{1}{p_2}) \cdots (1 - \frac{1}{p_k}).$$

PROOF 我们可以使用容斥原理来说明这个问题. 设 $1 \sim n$ 这 n 个正整数中 p_i 的倍数的集合为 $A_i, i = 1, 2, \dots, k$, 则 $|A_i| = n/p_i$; 同时是 p_i 和 p_j 的倍数的集合记作 $|A_i A_j| = n/(p_i p_j)$; 同时是 p_i 和 p_j 和 p_k 的倍数的集合记作 $|A_i A_j A_k| = n/(p_i p_j p_k)$. 以此类推. 那么我们现在考虑与他们不互素的数有多少个. 我们记作 $|\bigcup A_i|$, 根据容斥原理 (我们会在下一讲说明这个原理的正确性.), 有

$$\left| \bigcup_{i=1}^k A_i \right| = \sum_{i=1}^k |A_i| - \sum_{1 \leq i < j \leq k} |A_i A_j| + \sum_{1 \leq i < j < m < k} |A_i A_j A_m| - \cdots + (-1)^{k-1} |A_1 A_2 \cdots A_k|.$$

也就是

$$\left(\frac{n}{p_1} + \frac{n}{p_2} + \cdots + \frac{n}{p_k} \right) - \left(\frac{n}{p_1 p_2} + \frac{n}{p_1 p_3} + \cdots + \frac{n}{p_{k-1} p_k} \right) + \left(\frac{n}{p_1 p_2 p_3} + \cdots + \frac{n}{p_{k-2} p_{k-1} p_k} \right) - \cdots + (-1)^k \frac{n}{p_1 p_2 \cdots p_k}.$$

于是与之互素的数是 $\varphi(n) = n - |\bigcup A_i|$.

化简这个式子, 就有

$$\begin{aligned} \varphi(n) &= n - \left| \bigcup_{i=1}^k A_i \right| \\ &= n \left(1 - \left(\frac{1}{p_1} + \cdots + \frac{1}{p_k} \right) + \left(\frac{1}{p_1 p_2} + \cdots + \frac{1}{p_{k-1} p_k} \right) - \left(\frac{1}{p_1 p_2 p_3} + \cdots + \frac{1}{p_{k-2} p_{k-1} p_k} \right) \right. \\ &\quad \left. + \cdots + (-1)^k \frac{1}{p_1 \cdots p_k} \right) \\ &= n \left(1 - \frac{1}{p_1} \right) \left(1 - \frac{1}{p_2} \right) \cdots \left(1 - \frac{1}{p_k} \right) \end{aligned}$$

□

数论倒数

介绍

我们已经将同余方程里面写成了方程的形式. 我们自然地要问一问: 有没有办法解一个方程呢? 我们从最简单的一次同余式子开始看起.

Definition 24

设 $m, a, b \in \mathbb{Z}, a \not\equiv 0 \pmod{m}$, 则 $ax + b \equiv 0 \pmod{m}$ 称为一次同余或者一次同余方程. 我们说如果两个解不同, 是指互不同余的解.

Theorem 17

设 $m \in \mathbb{N}_+, a, b \in \mathbb{Z}, a \not\equiv 0 \pmod{m}$, 则 $ax + b \equiv 0 \pmod{m}$ 有解 $\iff (a, m) | b$.

PROOF

\implies : 设 $c \in \mathbb{Z}$ 满足 $ac + b \equiv 0 \pmod{m}$, 则 $ac + b = kx + m(k \in \mathbb{Z})$. 设 $d = (a, m)$, 则 $d|a, d|m \implies d|b$.

\impliedby : 设 $d = (a, m)$ 且 d 是 b 的因数, 则 $b = kd, k \in \mathbb{Z}$. 由 Bezout 定理, 存在整数 u, v , 使得 $d = ua + vm$, 同时乘上 d , 得到 $kd = kua + kvm$, 即 $kua + kvm = b$.

令 $c = -ku$, 则 $-ca + kvm = b$, 移项就有 $ac + b = kvm \equiv 0 \pmod{m}$. 也就是 c 是

| $ax + b = 0$ 的一个整数解.

□

但是解由多少个呢?

组合数学与概率简介

SECTION 19

二项式系数

PART
VIII

二项式系数由很多有趣的性质. 在计数原理一章中, 我们介绍了它的来历. 今天我们看一下它由什么好玩的性质.

组合恒等式看上去复杂, 但是其实挺有趣的. 在学习组合数的时候会遇到一些组合恒等式, 可能会觉得很难记忆和理解. 我们可以用故事的方法记忆组合恒等式.

Theorem 18 对于 $n \geq 0$ 的整数, 有

$$\binom{n}{k} = \binom{n}{n-k}$$

这个的含义是从 n 个元素中选出 k 个元素的组合数等于从 n 个元素中选出 $n-k$ 个元素的组合数。

Theorem 19 对于整数 k , 有

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Theorem 20 对于 $k \neq 0$, 有

$$\binom{n}{k} = \frac{n}{k} \binom{n-1}{k-1}$$

Theorem 21 对于 $k \neq 0$, 有

$$\binom{k}{k} + \binom{k+1}{k} + \cdots + \binom{k+r}{k} = \binom{k+r+1}{k+1}$$

Theorem 22 (Vandermonde 卷积)

$$\binom{n}{0} \binom{m}{r} + \binom{n}{1} \binom{m}{r-1} + \cdots + \binom{n}{r} \binom{m}{0} = \binom{m+n}{r}$$

多项式乘法
与二项式定
理
简介

我们在中学的时候学习多项式乘法, 比如 $(x_1 + x_2 + x_3)(y_1 + y_2 + y_3)$. 我们可以通

过表格的方法了解这一点. 例如下面的这个:

$$\begin{array}{ccc}
 x_1 & x_2 & x_3 \\
 \downarrow & \downarrow & \downarrow \\
 y_1 \rightarrow & x_1 y_1 & x_1 y_2 & x_1 y_3 \\
 y_2 \rightarrow & x_2 y_1 & x_2 y_2 & x_2 y_3 \\
 y_3 \rightarrow & x_3 y_1 & x_3 y_2 & x_3 y_3
 \end{array}$$

这就不禁让我们想到乘法原理: 我们在每个括号中只能选择一个数, 然后把所有的括号都选满. 找到所有这样的内容再相加就好了. 这就天然地启发我们使用排列组合的内容.

我们首先考虑比较简单的问题: 当我们遇到形如 $(a+b)^n$ 的表达式时, 如何高效地展开它们, 而不是逐项相乘? 二项式定理为你揭示了谜底! 这个定理告诉我们, 任何实数 a 和 b , 以及正整数 n , 它们的幂 $(a+b)^n$ 都可以用一种聪明的方式展开。

我们先来看几个实例:

$$\begin{aligned}
 (a+b)^1 &= a+b \\
 (a+b)^2 &= a^2 + 2ab + b^2 \\
 (a+b)^3 &= a^3 + 3a^2b + 3ab^2 + b^3 \\
 (a+b)^4 &= a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + b^4
 \end{aligned}$$

注意到, 每一个之前都是我们之前算过的组合数. 于是我们猜测, n 次的情形, 就有这样的形式:

$$(a+b)^n = \binom{n}{0} \cdot a^n \cdot b^0 + \binom{n}{1} \cdot a^{n-1} \cdot b^1 + \binom{n}{2} \cdot a^{n-2} \cdot b^2 + \dots + \binom{n}{n} \cdot a^0 \cdot b^n$$

这就是我们的二项式定理了.

Theorem 23 (二项式定理) 如果 n 是一个正整数, 那么对于所有的 x, y , 有

$$(a+b)^n = \binom{n}{0} \cdot a^n \cdot b^0 + \binom{n}{1} \cdot a^{n-1} \cdot b^1 + \binom{n}{2} \cdot a^{n-2} \cdot b^2 + \dots + \binom{n}{n} \cdot a^0 \cdot b^n,$$

用求和记号缩写, 就有

$$(x+y)^n = \sum_{k=0}^n \binom{n}{k} x^{n-k} y^k.$$

PROOF 将 $(x+y)^n$ 写作 $(x+y)(x+y)\cdots(x+y)$, 利用分配律将这个乘积完全展开, 然后再合并同类项. 因为在将 $(x+y)^n$ 乘开时, 对于每一个因子 $(x+y)$, 我们要么选择 x , 要么选择 y , 所以结果有 2^n 项. 并且每一项都可以写成 $x^{n-k}y^k$ 的形式 ($k = 0, 1, \dots, n$). 在 n 个因子中, y 选择 k 且在剩下的因子自然就选择了 x , 我们就得到了 $x^{n-k}y^k$. 由

于这样的选法一共有 $\binom{n}{k}$ 种, 自然

$$(x+y)^n = \sum_{k=0}^n \binom{n}{k} x^{n-k} y^k$$

成立. \square

有了二项式定理, 那么对于多项式的乘法也有类似的结论. 其核心就是现在每个括号里面选择一个数, 然后把所有的选择的数乘起来, 最后把我们所有可能的选法加起来.

SECTION 20

容斥原理

我们在上一次介绍计数问题的时候介绍了“减法原则”. 它允许我们使用整体减去部分的方式来得到我们要得到的问题. 这个有一个形象的名字, “正难则反”. 下面, 我们来看一下另一个类似的问题: 我们的加法原理只能胜任不相交集合的种类问题. 那我们相交的情况会如何呢?

从两个集合相交的情形开始看: 对于两个集合而言, 有

$$|S - A \cup B| = |S| - |A| - |B| + |A \cap B|.$$

这个感觉就像是逐渐地在调整. 减多了就加回来一点, 加多了就减掉一点. 对于三个集合而言, 我们就有

$$\begin{aligned} |S - A \cup B \cup C| &= |S| - |A| - |B| - |C| \\ &\quad + |A \cap B| + |A \cap C| + |B \cap C| \\ &\quad - |A \cap B \cap C| \end{aligned}$$

这就启发我们能不能让这个问题推广到一般的情形. 因为许多问题都有“交比并简单”的性质. 如果我们能够用交求并, 这样很多麻烦的计数问题就可以求解了. 所以, 现在我们的任务就是决定一个系数, 使得我们可以用集合的交表示集合的并. 我们一共有 8 个集合, 要算出他们的系数至少要 8 个未知数.

$$(x_0 \ x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \ x_7) \cdot \begin{pmatrix} |\emptyset| = 0 \\ |A| \\ |B| \\ |C| \\ |A \cap B| \\ |A \cap C| \\ |B \cap C| \\ |A \cap B \cap C| \end{pmatrix} = |A \cup B \cup C|$$

我们考虑从最简单的开始, 假设有一个元素的集合, 我们可以推出如下的表达式:

$$e \in A, e \notin B, e \notin C \Rightarrow x_1 = 1$$


$$e \in A, e \notin B, e \in C \Rightarrow x_1 + x_3 + x_5 = 1$$

$$e \in A, e \in B, e \in C' \Rightarrow x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 = 1$$

如果推广到 n 个集合, 上述的思路仍然适用. 不过, 我们可以使用计算机的代数求解方程的工具: [比如使用 Python 的 z3 工具库](#). 我们就一定程度上更加确定这个内容是对的:

Theorem 24 (容斥原理) 设 S 是一个有限集, A_1, A_2, \dots, A_n 是 S 的 n 个子集, 则

$$\left| S - \bigcup_{i=1}^n A_i \right| = \sum_{i=0}^n (-1)^i \sum_{1 \leq j_1 < j_2 < \dots < j_i \leq n} \left| \bigcap_{k=1}^i A_{j_k} \right|$$

那么, 什么问题求交容易、求并难? 我们来看  **错排问题**.: 求所有 $1, 2, \dots, n$ 排列中, 每个数字都不在原位的排列数量.

比如, $n = 4$ 的情形, 记 A, B, C, D 表示 $1, 2, 3, 4$ 恰好在第 $1, 2, 3, 4$ 个位置的排列. $n! - |A \cup B \cup C \cup D|$ 就是答案, 但是 $|A| = (n-1)!, |A \cap C \cap D| = (n-3)!$, 求交很容易, 求并就不那么容易了.

容斥原理, 加法原理, 乘法原理之所以叫做原理, 当然是因为他们是非常通用的. 与其说做知识点我认为更好的内容是叫做它们一种思维.

SECTION 21

概率问题简介

条件概率简介

我们在中学时候玩过很多和概率有关的游戏. 我们先对于一些基本的概念做一个简单的回顾:

Definition 25 事件是样本空间可能发生内容的子集, 概率是加到了事件上面.

Axiom 2 概率的可加性: 如果 $A \cup B \neq \emptyset$, 那么 $P(A \cup B) = P(A) + P(B)$.
对于可列无穷的情形: $P(A \cup B \cup C \cup \dots) = P(A) + P(B) + \dots$.

对于可数无穷的情形, 我们会发现很微妙的一件事情: 假设你有一个正方形, 上面随机选一个点, 那么这个点被选中的概率是多少? 一个点的面积是 0 , 所以我们的概率为 0 . 这时候你可能会发现, 概率为 0 的事件也有可能发生, 而不可能发生的时间概率为 0 . 这个到时候学习了测度论相关的内容就会知道数学家是如何构造的了.

我们接收到的信息总是部分的, 因此, 我们应该仔细探讨在给定一个情况下, 一个事情发生的概率.

假设我们在如下的概率空间里面:

TBD: A 3/6 AB 2/6 B1/6

我们定义...

从树状数组到线段树

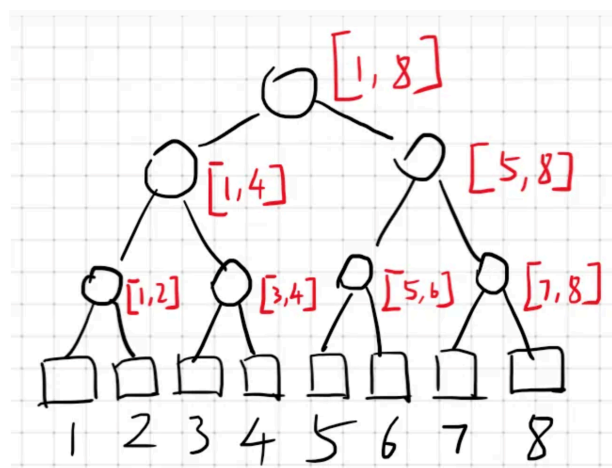
PART

IX

SECTION 22

树状数组简介

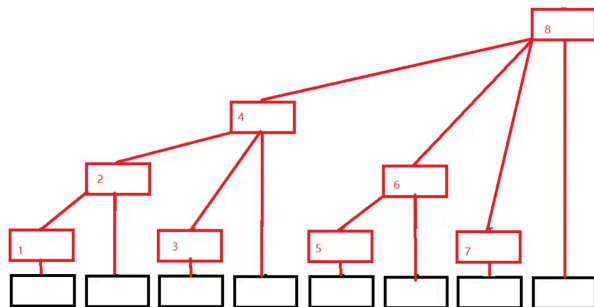
我们考虑一个前缀和的情形. 我们在上一回考虑的时候粗暴地为每一个节点都累加了一次. 这就导致了我们的节点查询快, 修改很慢. 下面, 我们能不能减小一部分节点, 使得我们可以更好的应对修改操作, 有不耽误查询操作? 这就要求我们尽可能减少冗余的节点. 一种想法可能是树:



我们可能会考虑一个二叉树. 这就是我们后面要了解的线段树的构型. 但是现在我们发现, 如果单单从 1 开始的情况下, 很多时候, 我们到底需要哪些节点.

$$\begin{aligned} S(8) &= [1, 8] \\ S(7) &= [1, 4] + [5, 6] + [7] \\ S(6) &= [1, 4] + [5, 6] \\ S(5) &= [1, 4] + [5] \\ S(4) &= [1, 4] \\ S(3) &= [1, 2] + [3] \\ S(2) &= [1, 2] \\ S(1) &= [1] \end{aligned}$$

于是把那些不必要的节点抽掉就有如下的一张图.



那么我们应该如何存储这张图呢？考察二进制的情况，就会发现，我们的 $C[i] = A[i - 2^k + 1] + A[i - 2^k + 2] + \dots + A[i]$ ，其中， k 是 i 的二进制中从最低位到高位看，连续零的长度。这样一个看似奇怪的规律其实是我们每一次固定我们的前缀和的起点导致的。接下来的主要问题就是如何求出二进制中从最低位到高位连续零的长度。我们可能会想到一些位运算的技巧。这个用到的技巧就比较多了。我们直接给出结论：这个值就是 $x \& (\sim x + 1)$ 。我们可以用一些实例来看一看。现在我们就有了我们的 `lowbit` 函数。

```
int lowbit(int x){
    return x & (-x); // 补码表示中  $\sim x = -x + 1$ 。
}
```

那么，这样一个前缀和数组，我们只要加上“有用的”位置的数据，不就成了吗？另外，我们在对区间 $[1..x]$ 加上一个数 y 的时候，我们就不用 $\mathcal{O}(n)$ 的时间了。

```
void add(int x, int y){
    for(int i=x; i<=n; i+=lowbit(i))
        t[i]+=y;
}
```

另外，如果查询 $[1..x]$ 的前缀和，我们就需要这样写：

```
int query(int x){
    int tot=0;
    for(int i=x; i>=1; i-=lowbit(i))
        tot+=t[i];
    return tot;
}
```

这样，我们就可以写出单调修改加上区间查询的代码。详细可以看例题 [P3374](#) 【模板】树状数组 1。

既然前缀和能做，那么差分当然也想试一试。我们对差分数组求前缀和可以求出当前的数 x ，就像以前说过的那样。我们这样做就可以应对区间修改，单点查询的问题了。区间修改来的时候，就像修改差分数组那样修改两个点的位置，单点查询来的时候，就求一次前缀和。具体的问题可以参考 [P3368](#) 【模板】树状数组 2。

那自然就会问了：区间修改 + 区间查询来了该怎么做？如果有原数组 a ，差分数组 c ，区间查询 a 的时候，来看一下区间查询的时候到底查询了什么：比如从 1 到 n 查询，

我们到底要得到什么？我们试着把它整理为以 $c[i]$ 为主元的内容，来观察一下：

$$\begin{aligned}
 \sum_{i=1}^n a[i] &= a[1] + a[2] + a[3] \cdots + a[n] \\
 &= c[1] + (c[1] + c[2]) + (c[1] + c[2] + c[3]) + \cdots + \\
 &\quad (c[1] + c[2] \cdots + c[n-1] + c[n]) \\
 &= \sum_{i=1}^n (n-i+1) \times c_i \\
 &= n \cdot \sum_{i=1}^n c[i] - \sum_{i=1}^n c[i] \times (i-1) \\
 &= \sum_{i=1}^n c[i] \times (n-i+1)
 \end{aligned}$$


好了，查询区间的工作我们可以划归为关于差分数组的问题。这个时候，我们就可以借助一些辅助数组来完成我们的内容了。由于运算的需要，我们让 ta 数组维护 a 的差分， tb 数组维护 $tb \times (i-1)$ 。这样子查询的时候就可以使用 $x*ta[i]-tb[i]$ 来做了。

具体地，求和工作可以这样写：

```
int query(int x){
    int tot=0;
    for(fint i=x;i;i-=lowbit(i))
        tot+=x*ta[i]-tb[i];
    return tot;
}
```

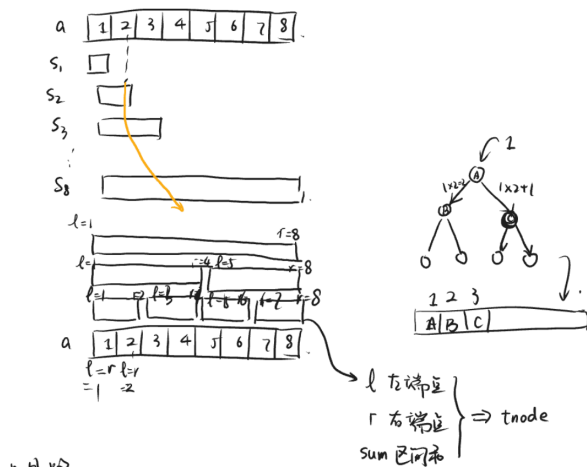
单点加：如果对 $[1..x]$ 单点加上 y ，会有什么影响呢？ $ta[i]$ 还是与原来的树状数组一样更新，让所有的波及到的 $ta[i]$ 的加上 x 。 $tb[i]$ 就是 $(ta[i] + x) \times (x-1)$ 。我们只需要更新 $x \times (i-1)$ 即可。

```
void adds(int x,int y){
    for(fint i=x;i<=n;i+=lowbit(i))
        ta[i]+=y,tb[i]+=y*(x-1);
}
```

至此，我们就在差分的基础上，了解了单点修改就是 $adds(i,x)$ ，在 $[l..r]$ 区间修改就是 $adds(l, x); adds(r+1, -x);$ ，单点查询为 $query(x)$ ，区间查询为 $query(r)-query(l-1)$ 。我们就可以用树状数组的方法通过  **P3372 【模板】线段树 1** 了。当然，也可以看到，用线段树实现比较简单的方法在树状数组这里就比较繁杂。而且我们发现如果操作太复杂，差分这个着力点消失了，这道问题就难以解决。我们接下来转而学习线段树的内容。如果大家有兴趣的话，可以[参考 kingxbz 写的一篇博客](#)。

线段树简介

我们沿用刚刚在使用树状数组的时候的内容. 我们同样用一个长度为 8 的数组说明.



每一个节点有它自己的左端点, 右端点, 以及值. 因为是二叉树, 我们采用左节点为当前节点乘 2, 右孩子为当前节点乘 2 加 1 的方法存储.

我们采用好理解的方式, 把所有的内容写到了一个结构体里面.

线段树建树

```
struct tnode{
    int l=0, r=0, sum=0;
};

struct SegTree{
    tnode t[MAXN];
    int mp[MAXN];
    //...
    void update(int o){
        int ch = o<<1;
        t[o].sum = t[ch].sum + t[ch+1].sum;
    }
    void build(int o, int l, int r){
        t[o].l = l, t[o].r = r;
        if(l != r){
            int mid = (l+r)/2;
            int ch = o*2;
            build(ch, l, mid);
            build(ch+1, mid+1, r);
            update(o);
        }else {
            t[o].sum = a[l];
            mp[l] = o;
        }
    }
};
```

```

    }
}

```

这段代码中, t 表示存储线段树节点的数组, 最大容量为 $MAXN$ 。每个元素 $t[i]$ 表示线段树的第 i 个节点。 mp 数组表示一个映射数组, 用于记录线段树中每个节点所对应的原始数组 a 的索引位置。

在结构体 `SegTree` 中定义了一个 `build` 方法, 用于构建线段树。该方法采用递归的方式建立线段树, 其中 o 表示当前节点操作的时哪一个节点, l 表示当前节点表示的区间的左边界。 r 表示当前节点表示的区间的右边界。在 `build` 方法中, 首先将当前节点 $t[o]$ 的左边界 $t[o].l$ 和右边界 $t[o].r$ 初始化为传入的参数 l 和 r 。然后判断当前节点 $t[o]$ 是否表示一个叶子节点 (即 l 和 r 相等)。如果是叶子节点, 将其 `sum` 初始化为原始数组 a 中索引为 l 的元素值, 并将 l 位置的映射值 $mp[l]$ 设置为当前节点的索引 o 。

若当前节点不是叶子节点, 则分别计算当前节点的左子节点和右子节点的索引。左子节点索引为当前节点索引 o 的两倍, 右子节点索引为当前节点索引 o 的两倍加一。然后递归调用 `build` 方法, 构建左子树和右子树。

最后, 在递归的过程中, 每次构建完一个子树后, 调用 `update` 方法来更新当前节点的 `sum` 值, 即将左子节点和右子节点的 `sum` 值相加。

我们可以来模拟一下这个内容。

线段树
查询

那么查询的时候应该如何查询呢? 这个也可以递归地进行。分为两种情形: 第一种情形是 l, r 落在同一区间。这时候我们只要在它的左孩子/右孩子里面继续查找就行了。如果跨过了中间的区域, 那就要分别去找左孩子和右孩子, 再相加。于是我们有如下的代码:

```

// In structure SegTree{...}
int qsum(int o, int l, int r){
    if(t[o].l == l && t[o].r == r){
        return t[o].sum;
    }
    int mid = (t[o].l+t[o].r)/2;
    int ch = o*2;
    if(r<=mid) qsum(ch, l, r);
    else if(mid < l) qsum(ch+1, mid, r);
    else return qsum(ch, l, mid)+qsum(ch+1, mid+1, r);
}

```

线段树
单点修改

我们顺着叶子结点, 一路加回去就好了, 就和树状数组一样的行为:

```

// In structure SegTree{...}
void change(int x, int y){
    x = mp[x];
    t[x].sum += y;
    while(x/=2){ // 正整数除法都是下取整的
        update(x);
    }
}

```



```

        return t[o].sum + t[o].lazy*(r-l+1);
    }
    int ch = 2*o, mid = (t[o].l+t[o].r)/2;
    if(r<=mid) return query(ch, l, r);
    else if(mid<l) return query(ch+1,l,r);
    else{
        return query(ch, l, mid)
            +query(ch+1, mid+1, r);
    }
}

```

我们之所以要 `change` 的时候就要 `pushdown`, 是因为防止出现这个问题: 假设我们有操作 (1): 将 $l..r$ 变为一个数; (2): 将 $l..r$ 加上一个数. 这时候对于 $5 \sim 8$ 变为 3, 理所当然, 他会打上一个变 3 的标记. 然后我们 $5 \sim 6$ 再加上 1, 它会在 $5 \sim 6$ 上打上加 1 的记号. 现在假设我们要查询 $5 \sim 6$, 我们的加 1 的标记就会被上面的覆盖掉了. 得到错误的结果.

懒标记
共性

我们来考察 `lazytag` 的共性. 具体的, 我们发现懒标记在使用的时候有三种操作:

- `pushdown`: 将上面的一个标记清空, 把下面的两个标记传达这个上面的标记的内容;
- `cal_lazy`: 运用当前的现有数据和懒标记得到当前树节点的真实值.
- `tag_union`: 如何将两个标记合并.

在刚刚的举的例子中的应用, 我们就可以把上述的 `pushdown` 写得更抽象一点:

```

void pushdown(int o){
    cal_lazy(o); // 计算当前节点的懒标记影响下的真实值
                // t[o].sum + t[o].lazy*(r-l+1)
    if(t[o].l == t[o].r){
        int ch = o*2;
        union_lazy(o, ch); // 将当前节点和左孩子的懒标记合并
                          // t[ch].lazy += t[o].lazy
        union_lazy(o, ch+1); // 将当前节点和右孩子的懒标记合并
                          // t[ch+1].lazy += t[o].lazy
    }
    init_lazy(o); // 将当前节点的标记清空
                // t[o].lazy = 0;
}

```

这样子, 剩下的代码还是正常书写. 我们就理解了一个最简单的懒标记.

我们来看一个例子, 这个例子是既有加, 又有乘的标记. 考虑到这两点, 我们假设有懒标记 a 和 b , 他们的任务是对于一个 `sum`, 它们的任务是 $\text{sum} \rightarrow a \times \text{sum} + b$. 我们按照上面的内容考虑:

区间标记的合并. 假设第一块上面有现在的和 x_1 , 加法懒标记 a_1 , 乘法懒标记 b_1 , 第二块有现在的和 x_2 加法懒标记 a_2 , 乘法懒标记 b_2 . 我们如何取呢? 因为我们的标记

是实时往下推的, 位于上面的标记一定是后来的, 在下面的标记一定是先来的. 这样子, 我们的结果就是 $a_1(a_2x + b_2) + b_1$. 化简有 $a_1a_2x + a_1b_2 + b_1$. 这就是线段树懒标记的合并.

在考虑这个的时候, 剩下的就顺便考虑到了. 这个想法写这个问题还是很好的. 其实这个问题可以同样运用区间的平方. 我们可以看 `◇C++lazy-tag.cpp` 文件的关注 `sum[0]` 的部分. `sum[1]` 维护了区间的平方和. 需要一点求和的技巧.

References

[Erickson] Erickson, J. *Algorithms*. Independently published.

[Jiang] Jiang, Y. 应用视角的操作系统. Retrieved from
<https://jyywiki.cn/OS/2023/build/lect2.ipynb>.