基础内容简介 (DRAFT)

AUGPath 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

PART

I

C 语言回顾

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

PART

II

递归问题

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

链表
单向/双向

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.

UVA806 Spatial Structures.

PART

III

一些有趣的思想

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.

Subsection 3.1

基本的例子

整数二分
原理

我们的目标是找一个性质的边界. 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)>>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>=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)>>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>=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.

Subsection 3.2

更多的例子

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 $S_n = a_1 + a_2 q + a_3 q^2 + \cdots + a_n q^{n-1}$.

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. 这里是原始文章.

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. 和上一个问题是类似的. 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 $s_i = a_1 + \cdots + a_i$.

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 $\cdots, a_l, a_{l+1}, \cdots, a_r, \cdots$, 那么经过这次对换之后, 这个区间的球的顺序就变成了 $\cdots, a_{l+1}, a_{l+2}, \cdots, a_r, a_l, \cdots$.
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: $1 \le n, m \le 10^5, 0 \le a, b \le 9, 1 \le l \le r \le n$.

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

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.

加一个平方数列?

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.

前缀和

TBD: 阐述数列 $1, 1, 1, 1, 1...$ 和杨辉三角, 组合数的联系.

多维前缀和

二维的前缀和

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.

Remark 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 Dymanic 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: $2^1=2, 2^2=4, 2^3=8, \cdots$. 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: $2\times 2=4, 4\times 4=16, 16\times 16=256, 256\times 256=65536\cdots$.

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:

[3] 没有无聊的时候? 相信我到时候一定会有的. 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.

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: $f[a..c]=f[a..b]\mathrm{OP}f[b..c]$. 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

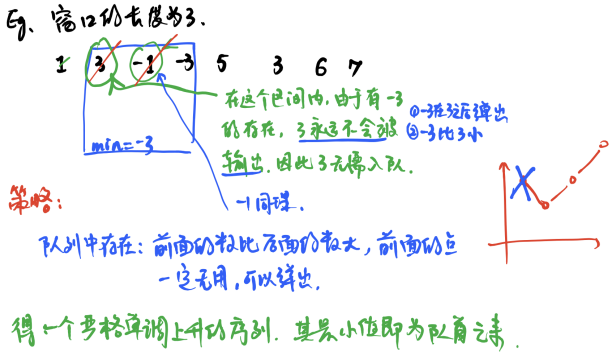
除此之外, 我们还会有很多很有趣的思考问题的方法. 我们下面举出几个例子:

滑动窗口

介绍

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 $\mathcal{O}(nk)$. 但是我们想一想这个真的需要这么复杂吗?

考虑优化, 有些元素似乎是没用了的. 具体如下图所示:



Section 8

并查集简介：指来指去

TBD: 介绍并查集的一些有趣的操作. 并且提示一个有名的话语: 计算机科学中的问题大部分可以加一层抽象层解决, 不能的那些就是加了太多的抽象层.

PART

IV

树与图

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

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

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 $\mathcal{O}(n^2)$ 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 $\mathcal{O}(n)$, 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.

Section 10

常见图算法 I

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: $A \to B \to C \to D$.

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数.

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, 大的胃 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)

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.

[4] 优先队列是可以让最大值出队列的一种数据结构.

简单计数问题

PART

V

TBD: 介绍基本的计数方法, 一方面可以用来分析自己程序的答案空间, 另一方面为后来的组合问题做铺垫.

PART

VI

动态规划简介

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 11

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, 我们发现这是我们要求最大决定的 – 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 (sufficent) 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.

看上去这会让我们的转移方程难以写. 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 1

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 $a[0], a[1], a[2], \cdots, a[n-1], a[n]$ 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
表达关系

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.

最大上升子序列之和

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, $a[1], \cdots, a[i-1]$. 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.

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. 但是我们发现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, $b[1], b[2], \cdots, b[j-1]$. 这样一来, 我们就从实际意义出发, 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 $N(1 \le N \le 10^5)$ 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 $N, k(1 \le k \le 100)$, 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条 (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时候也可以这样做.

Section 12

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.

0-1 背包问题

简介

简介 TBD. 我们考虑设计状态.

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.

多重背包问题

T

BD

Section 13

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: $f[i][j] = \min\{f[i][k]+f[k+1][j]+\sum_{s=i}^{j} a[i]\}$. 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. 这个和上一个是类似的: 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 $i+1, i+2, \cdots, r-2, r-1$. 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 $f[i][k]+f[k][j]+w[l]\times w[k]\times w[r]$. 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 $(l,l+1),(l+1,l+2),\cdots,(r-1,r),(r,l)$ 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 $l+1, l+2, \cdots, r-2, r-1$, 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 ($g[1][n] =: R$)[5] , 同时知道左子树的区间和柚子树区间为 $1..R-1, R+1..R$, 反复进行这个过程就行了.

[5] $:=$ 表示"定义做". 冒号在被定义的表达式那一侧

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: [6]

[6] 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 x 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 14

树形 DP

数论简介

PART

VII

TBD: 介绍一些和数论有关的内容.

组合数学与概率简介

PART

VIII

TBD: 可能需要更多的时间来理解一些问题... 介绍点容斥原理和二项式系数.

从树状数组到线段树

PART

IX

TBD: 这部分线段树的内容已经准备好, 重点在于准备标签的修改的阐释, 以及给出几个例子最后修改原先文稿的排版即可. 如果有空, 干脆把有些高级技术介绍了, 如动态开点等等...