

问题分析与线性优化策略

2021级第二次公开课 2021.10.31

GDUT-//undefined & GDUT20ZYL

前言

Why algorithms?

我想问大家一个问题：算法是什么？

在回答算法是什么之前，我们先想想算法是怎么来的。

认识算法的普通路线：

做很多很多道题 - 对着题目自闭很多很多次 - 有一天，忽然想到——

诶这几道题的做法都很相似啊！

然后呢？

我们概括归纳这一类题，把这一类题的“性质”总结出来，在“性质”之上总结一个通用的方法。

算法并不基于哪一道具体的题目，而是基于一部分问题所拥有的一种性质。

边注：题目和问题有什么区别？

在本节课的定义中，题目是最直接的、由文字所组成的；而问题是更宽泛的概念，多个题目可能实际上是一种问题，在解决一个问题时也可能有很多个子问题。

一个问题通常由给定的数据和期望的输出组成。

我们怎样认识算法？

没有性质就没有算法。

前面说的普通路线显然很笨拙，前期学习曲线也极其陡峭。但我们面对的情形是，留给我们学习的时间并不多，我们需要短时间内高效地掌握算法。

我们要怎么办？

认识算法的高效路线：

集中、快速地做适用同一个算法的题目。通过思考题目认识问题所拥有的性质。

考虑怎样修改题目会导致算法不适用，了解算法的**能力边界**。这也是认识问题所拥有的性质。

通过研究问题的性质认识算法，这就是本节课的主题。

目录

- 审题：思维的起点
 - 数据范围
 - 题目类型
- 优化：化腐朽为神奇
 - 差分与前缀和
 - 离散化
 - 双指针与二分
 - 桶

关于例题

课后建议完成课上例题。我龙哥认为有训练价值的题，都会写题号。题目号格式与洛谷相同。

Part 1: 审题

题目构成

Description

Zhazhahe很喜欢吃烧饼，但是要做好一块烧饼，要把两面都弄热，如果一次只能弄一个，zhazhahe一定会等得不耐烦，幸好现在有一个大的平底锅，一次可以同时放入 k 个烧饼，一分钟只能做好一面。而现在有 n 个烧饼，至少需要多少分钟才能全部做好呢？

- 背景（一般可以忽略）
- 题目描述
- 输入输出格式
- 数据范围

题目描述一定要和数据范围一起看，请看下面的例题来理解这句话。

斐波那契数列

斐波那契数列由两个1开始，之后的斐波那契数由它之前的两数相加而得出。

将斐波那契数列的第 n 项记为 a_n . 有 T 次询问，每次输出 $a_n \bmod 10^7$

数据范围：

- $1 \leq T \leq 10, 1 \leq n \leq 1000$
- $T = 1, 1 \leq n \leq 10^{18}$
- $1 \leq T \leq 5 \times 10^7, 1 \leq n \leq 10^9$

取模运算 (mod)

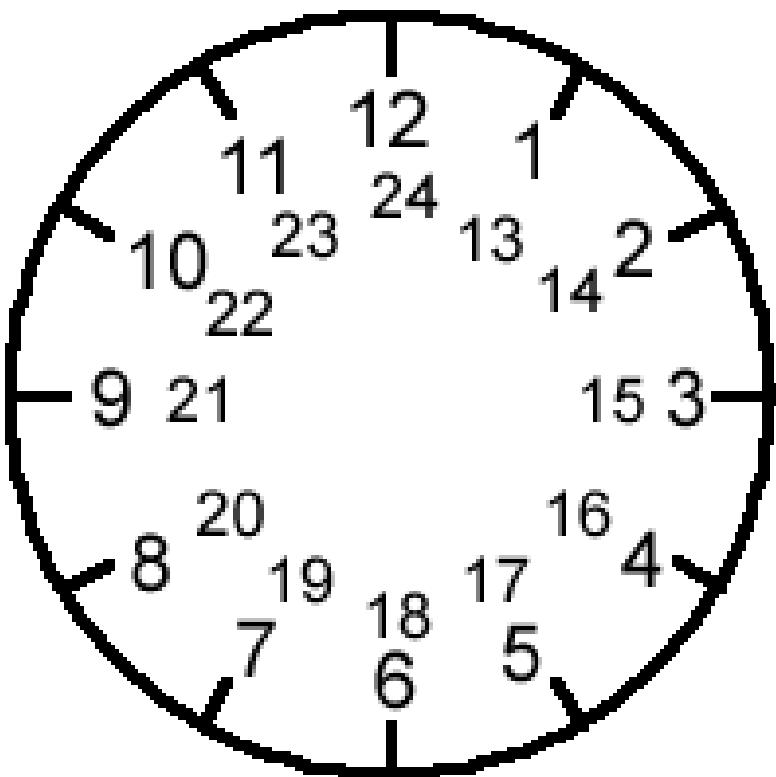
我们可以把取模运算想象成将数轴弯曲绕成一个圆圈。

参考右图中的时钟， $13 \bmod 12 = 1$. 取模运算就是求相除后得到的余数。

运算法则

与四则运算类似，但是除法例外。其规则如下：

- $(a + b) \bmod p = (a \bmod p + b \bmod p) \bmod p$
- $(a - b) \bmod p = (a \bmod p - b \bmod p) \bmod p$
- $(a \times b) \bmod p = (a \bmod p \times b \bmod p) \bmod p$
- $a^b \bmod p = ((a \bmod p)^b) \bmod p$



P1720

将斐波那契数列的第 n 项记为 a_n . 输出 $a_n \bmod 10^7$

数据范围 $1 \leq T \leq 10, 1 \leq n \leq 1000$

解法：模拟，总复杂度 $Tn \approx 10^4$

```
#include <iostream>
using namespace std;
const int MAXN = 1001;
const int MOD = 1e7;
int main() {
    int f[MAXN] = {0};
    f[1] = 1;
    f[2] = 1;
    for (int i = 3; i <= MAXN; i++) {
        f[i] = (f[i - 1] + f[i - 2]) % MOD;
    }
    int t, n; cin>>t;
    while(t--){
        cin>>n;
        cout<<f[n]<<endl;
    }
}
```

P1962

数据范围 $T = 1, 1 \leq n \leq 10^{18}$

解法：直接模拟会超时，需要使用**矩阵快速幂**将单次求值复杂度优化至 $\log_2 n$ ，总复杂度 $Tn \approx 60$ ，轻松AC！

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 \end{bmatrix}$$

“点积”是把**对称的元素相乘**，然后把结果加起来：

$$(1, 2, 3) \bullet (7, 9, 11) = 1 \times 7 + 2 \times 9 + 3 \times 11 = 58$$

$$\begin{bmatrix} fib(i) \\ fib(i-1) \end{bmatrix} = \begin{bmatrix} fib(i-1) \\ fib(i-2) \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix} = \begin{bmatrix} f(2) \\ f(1) \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-2}$$

P5110

数据范围 $1 \leq T \leq 5 \times 10^7, 1 \leq n \leq 10^9$

解法: $T \log_2 n \approx 1.5 \times 10^9 > 10^8$. 前一个优化也过不了。此题需要更进一步的技巧, 我们称之为**光速幂**。

至此已然达到**紫题/金牌题**难度, 故不展开讲了。—~~(其实是因为我也不会)~~—

时间复杂度与算法对照表

由时间复杂度推测可能的算法。在不同数据范围内，代码的时间复杂度和算法该如何选择？

- $n \leq 30$, 指数级别, dfs+剪枝, 状态压缩dp
- $n \leq 100 \Rightarrow O(n^3)$, floyd, dp, 高斯消元
- $n \leq 1000 \Rightarrow O(n^2), O(n^2 \log n)$, dp, 二分, 朴素版Dijkstra、朴素版Prim、Bellman-Ford
- $n \leq 10000 \Rightarrow O(n * \sqrt{n})$, 块状链表、分块、莫队
- $n \leq 100000 \Rightarrow O(n \log n) \Rightarrow$ 各种sort, 线段树、树状数组、set/map、heap、拓扑排序、dijkstra+heap、prim+heap、spfa、求凸包、求半平面交、二分、CDQ分治、整体二分
- $n \leq 1000000 \Rightarrow O(n)$, 以及常数较小的 $O(n \log n)$ 算法 \Rightarrow 单调队列、hash、双指针扫描、并查集, kmp、AC自动机, 常数比较小的 $O(n \log n)$ 的做法: sort、树状数组、heap、dijkstra、spfa
- $n \leq 10000000 \Rightarrow O(n)$, 双指针扫描、kmp、AC自动机、线性筛素数
- $n \leq 10^9 \Rightarrow O(\sqrt{n})$, 判断质数
- $n \leq 10^{18} \Rightarrow O(\log n)$, 最大公约数, 快速幂
- $n \leq 10^{1000} \Rightarrow O((\log n)^2)$, 高精度加减乘除
- $n \leq 10^{100000} \Rightarrow O(\log k \times \log \log k)$ (k 表示位数), 高精度加减、FFT/NTT

Tips:

$$\log_2 10^n \approx 3n$$

64MB至多开1600万个int

C++在一般测评机上1秒能计算 10^9 次加法。

题型

我们可以通过题目的特点猜测题目所对应的题型。这样我们可以快速缩窄思考方向，更高效地想出解法。

下面介绍几种常见题型：

- 动态规划
- 计数
- 数论
- 构造
- 数据结构
- 可行性
- 图论
- 树上
- 计算几何

动态规划简介

动态规划(Dynamic Programming, DP)用于求解具有最优子结构和重叠子问题的题目。

重叠子问题：记忆化

有一楼梯共 n 级，刚开始时你在第0级，若每次只能跨上一级或二级，要走上第 n 级，共有多少种走法？

请问平面上 n 条彼此相交而无三者共点的直线，能够把平面分割成多少部分？

最优子结构（也叫最优化原理）：最优策略的子策略也必定是最优的

比如我要从河一侧的一点 a 到达另一侧的一点 b ，而有且仅有一架桥*bridge*供过河（桥长度不计）。这两点与桥都有很多条道路相连接。那么我们可以肯定这两点最短距离 $D_{a,b}$ 满足

$$D_{a,b} = D_{a,\text{bridge}} + D_{\text{bridge},b}.$$

也就是子解法都为最优时，这个解法必定是最优的。

例题

肥猪的钢琴床 - 2020年广东工业大学第十届文远知行杯新生程序设计竞赛

怎么看出这道题是DP题呢？

其实准确地说，这是一道可以用DP做的题，也有不用DP的做法。

对于大多数人来说，这种题的DP做法易想易写，所以我们把它们称为DP题目。

观察此题：输入的是一个**序列**，输出的一个**最少的数量**。

从命题角度来看，序列提供了一个线性的DP载体，而最少的数量，说明了这是一个最优化问题，而DP常用来求解最优化问题

由于这种DP是在序列上面做，有限序列可以抽象成一条线段，所以这种DP被我们称之为：**线性DP**。

线性DP有一些统一的技巧和套路，可以进行专门的训练。

由于我们这里只是分析题型，就不展开来讲，如果后续有DP的课程，可以考虑做一个更深入的讲解。

DP题目的特征

其实DP有很多种：

线性DP、状压DP、DAG、树形DP、轮廓线DP、插头DP、概率DP.....

但不论是什么DP，从题面来看，都有类似的特征：

比方说：求最大/最小/最长/最短的某个数（**最值**），统计方案数、相似度等。

而且多数情况下，是提供一组**具体的数据**，然后算某个值。

如果没有提供具体数据（中间每一项的数据），比如像刚才的斐波那契数列，只给你一个n，求通项，这样往往就不属于DP问题

计数问题

把DP题的问法，换成不给具体数据的情况

比如：计算 n 以内的互质对数 ($n \leq 10^9$)，答案mod 998244353.

计数问题，就是数数题，一般是数一个东西有多少个

数论问题

题目描述

求

$$\sum_{i=1}^n \sum_{j=1}^m \sum_{k=1}^p \gcd(i \cdot j, i \cdot k, j \cdot k) \times \gcd(i, j, k) \times \left(\frac{\gcd(i, j)}{\gcd(i, k) \times \gcd(j, k)} + \frac{\gcd(i, k)}{\gcd(i, j) \times \gcd(j, k)} + \frac{\gcd(j, k)}{\gcd(i, j) \times \gcd(i, k)} \right)$$

由于答案可能过大，输出答案对 $10^9 + 7$ 取模的值。

数论是研究整除性相关的问题，所以遇到约数（因子）、倍数、素数（质数）、整除等概念时，一般就是数论问题

构造问题

让你给出一种方案，满足某些条件，答案不唯一。

1. 给定一个排列，允许元素交换操作，输出一组操作将此排列排序，要求操作个数为 $O(n)$
2. 输入一个数 n ，要求输出面积为 n 的非直角三角形的三条边长，要求边长为整数
3. 无输入，输出两个字符串，使得他们在给定的哈希方式下，哈希值相等

数据结构问题

维护某个东西，然后回答询问。

n 次操作，操作类型有：

删除 x 、加入 x 、翻转 $[L, R]$ 、求 $[L, R]$ 中的最大子段和

往往是某些可以直接用枚举、二分、递推等基本算法做的题，加上了修改操作，就变成了“动态问题”，我们就需要考虑用数据结构维护。

可行性问题

有一类问题，他需要你判断答案的可行性，只有是/否两种状态（或者加上非法状态共3种状态）

编一个程序，再给出初始条件之后，判断是先取必胜还是先取必败。

博弈问题属于组合数学研究的范畴，它在信息学竞赛中，往往属于可行性问题。

对于相关问题，有不少成熟的模型。

图论问题

图论是一个非常有意思的板块，非常依赖建模，只要能建立图论模型，都能套用经典图论算法解决。

有 n 个村庄，其中有 m 条道路，每条道路连接两个村庄。

可以抽象成 n 个点， m 条边，每条边连接两个点，并且点与点之间不一定互相连通（能通过道路互相走到则为连通），我们称它为图。

比如这道题：

CF463D Gargari and Permutations

给你 k 个长度为 n 的排列，求这些排列的最长公共子序列的长度。

看到序列求最长的长度，你想到了什么？

DP! 这道题DP显然可以做。

但在我们学习了图论之后，可以在建模后，通过求最长路的经典算法，解决这个问题。

树上问题

树是一种特殊的图。它是没有回路的连通图。

有 n 个村庄，其中有 $n - 1$ 条道路，每条道路连接两个村庄。每个村庄都可以通过这些道路，到达任意另一个村庄。

此题可以抽象成 n 个点， $n - 1$ 条边，并且互相连通（称为连通图）。此图一定是一棵树。（证明课后思考）

由于树有许多优美的性质，所以关于它的解法往往自成一派，于是我们称它为树上问题。

计算几何问题

和数学里的解析几何差不多，考到许多初高中几何知识。

母牛上柱 - 2020年广东工业大学第十届文远知行杯新生程序设计竞赛

ACM中的数学

在这一小节的最后，需要厘清几个题型相关的概念，帮大家避坑

首先，ACM数学并不能笼统的称为数论，根据实际经验，它涉及几个板块：

- 数论：主要研究整除性相关的问题，比如最大公约数、最小公倍数、约数、因子、倍数、素数等概念。
- 组合数学：主要研究组合计数类问题，比如排列组合问题、概率问题、博弈问题、解递推数列、生成函数等
- 具体数学：这里指的是一本书，里面包含大量数学推导技巧，后期专攻数学时可以考虑学习
- 近世代数：即抽象代数，在这里不展开讲，某些问题需要用到里面的一小部分知识。
- 计算几何：即用计算机解决几何问题，它的基本计算方法非常成熟，推荐后期深入学习。

Have a break

课间休息5分钟。可提问，也可以思考一下下面这道题。

CF1582A - Luntik and Concerts

题目大意：

Luntik有 a 个1分钟的曲目、 b 个2分钟的曲目、 c 个3分钟的曲目。现在，他需要把这些曲目安排到2场音乐会上，并使得这两场音乐会时长尽可能相近。一首曲目只能分配到一场音乐会。你需要回答这2场音乐会时长的最小差值。

输入：

第一行 t ($1 \leq t \leq 1000$)，代表测试用例数量。对于每一个测试点，输入三个数 a, b, c ($1 \leq a, b, c \leq 10^9$)

输出：

对于每个测试用例，输出2场音乐会时长之间的最小差值。

提示：请注意题目中的**数据范围** $1 \leq a, b, c \leq 10^9$.

Part 2: 优化

线性

线性一词由英文linear翻译而来。

1. 能够在图形上用直线表示；涉及或表现出两个相关量成正比变化。
2. 在一系列步骤中从一个阶段进展到另一个阶段；顺序的。

上节课讲了差分、前缀和、二分、尺取（双指针算法的一种）

其实这些操作，都属于线性优化。

线性，在这里指的就是一个序列，以及序列上的（一个接一个的）操作： $\{a_1, a_2, \dots, a_n\}$

还可以拓展到多维，乃至于更复杂的结构（树、图等）。比如二维，我们把每一行抽象成点，就是线性的叠加，二维序列也是一个线性序列。更高维度以此类推。

学会线性的情况以后，我们把更复杂的结构拆成线性结构就能分析了。

常见线性优化策略

常见的转化策略有以下几种：

1. 差分与前缀和：两者互为逆运算，这次会深入分析差分与前缀和的实质，以及它在各种难度题目中的运用
2. 离散化
3. 双指针：有多种类型，尺取只是其中一种，本质是利用了问题的单调性
4. 二分：二分和双指针，其实解决的是同一个问题，一般来说，能双指针就能二分，能二分就能双指针。
5. 单调队列/栈：实际上单调队列和单调栈是差不多的东西，后面细说

差分

我们已经知道，差分就是令`diff[1] = a[1] - 0`且`diff[i] = a[i] - a[i - 1]`

通过差分，我们就可以将区间操作，变成两个单点操作，比如我们要将 $a_i, i \in [L, R]$ 全部加1，只需要令`diff[l]`加1，`diff[r + 1]`减一。最后再通过前缀和`diff[i] = diff[i - 1] + a[i]`还原序列

现在请思考，差分操作的实质是什么？为什么我们可以这么做？它的局限性在哪？

差分的实质，就是对区间操作的降维。

我们把序列想象成一个线段，上面的每一个点都是序列中的一个元素。操作一个点，只影响这个点本身；读取这个点，也只是包含一个点的信息

当我们差分了之后，这个点就只存了当前位置元素，与前一个元素的差值。这意味着所有元素的真实值，都依赖于他前面的元素。

那么，我们改变 i 位置的元素的值，实际上是改变了从这个点开始，到整条线段右端点的子线段的值。

如果我们要改 $[L, R]$ 区间上元素的值，就只用从 L 开始把后面的值都改了，然后从 $R + 1$ 开始把后面的值复位就行了。

为什么可以差分

首先，一定要是做减法才算差分吗？一定要是区间加减才能用差分吗？

显然不是。

是因为减法操作具有逆操作——加法，且加法能够累计，所以可以满足条件。

比如异或，它自己是自己的逆运算，那么我们可以做前缀异或和以及异或差分；乘法可以有前缀积等。

异或简介

由真值表可以看出，取异或的两值不同时结果为真，取异或的两值相同时结果为假。

我们可以把异或看作“取反”操作，0代表维持原样，1代表取反。

异或这一名字就取自 $p \oplus q = (\neg p \wedge q) \vee (p \wedge \neg q)$

为什么可以差分

其次，题目**只在最后**要求输出序列。这意味着中途我们不需要知道原序列。

仔细想想，差分让我们付出了什么代价？我们用差分把点变成了线段，那点去哪了？

如果题目变成，长度为 n 的序列， T 种操作($n \leq 5000, T \leq 10^5$)，操作有两种类型：

1. 将 $[i, j]$ 加上 k ；
2. 立刻输出序列的值

差分做法，马上就举步维艰了。

我们在操作中途，获取原数列的点值的代价是 $O(n)$ 的，差分做法复杂度也将是 $O(Tn)$ 的，显然不行。

恰恰是因为这道题，将操作与询问分离，将修改与求值分开，才让我们有使用差分的机会。

差分的局限性

所以差分的主要局限性就是：只能解决**静态问题**。

静态问题指的就是将修改与求值分离的问题。与之相对的动态问题，指的是**一边修改，一边求值**的问题。

对于动态问题，一般使用数据结构，或者分治技巧进行求解，那是更加深入的内容，在本节课暂时不涉及。

线性优化思维点拨

前面的分析，是为了让大家了解前缀和与差分的实质，提高真正的算法思维水平。

在真正的高水平比赛、高质量题目面前，只是学会固定的套路是没任何作用的，我们要掌握的是分析问题的思维和方法。

以后我们如果遇到序列操作的问题，我们应该先分析题目的性质，然后见招拆招。

- 静态还是动态？
- 如果是静态，有没有区间操作？
- 有区间操作的话，区间运算是否有逆运算？
- 反复操作可不可以差分？反复询问可不可以前缀和？
- 如果是动态，能不能套数据结构/分治？

这种思维能帮我们快速掌握一类问题的切入点，分析清楚问题。

前缀和与差分例题

P1115 最大子段和

给你一个长度为 n 的序列 a ，选出其中连续且非空的一段使得这段和最大。

数据范围： $1 \leq n \leq 2 \times 10^5$

这个思维强的话，可以直接想出贪心做法：

假设当前数为一段的最后一个数，那么无论如何都不能跳过前一个数去选更前面的数。

设当前以第 i 个数结尾的子段最大和为 $\max[i]$ ，我们可以看出若是 $\max[i - 1] + a[i] < 0$ ，我们就还不如不选前面的数，直接 $\max[i] = a[i]$ 即可。其他情况下就必然是 $\max[i] = \max[i - 1] + a[i]$. 最后找出最大的 $\max[i]$ 就可以了。由于 $\max[i]$ 可以直接由一个变量存储，空间复杂度 $O(1)$. 时间复杂度 $O(n)$.

但这就没灵魂了，咱在学前缀和呢，能不能前缀和搞定？

P1115 前缀和做法

给你一个长度为 n 的序列 a ，选出其中连续且非空的一段使得这段和最大。

数据范围： $1 \leq n \leq 2 \times 10^5$

起点不确定的情况不好控制，我们可以把子段和转化成：第 1 到 x 个数之和减去第 1 到 y 个数之和，其中 $x > y$ 。对于每一个 x ，问题就转换成了求最小前缀和。时间复杂度 $O(n)$.

高维前缀和与差分

上节课讲了二维前缀和的题目

不知道大家有没有仔细分析，二维前缀和的代价是什么？

- 一维的前缀和: $s[i] = s[i - 1] + a[i]$ ——两项
- 一维差分: $diff[1] = a[1] - 0$, $diff[i] = a[i] - a[i - 1]$ ——两项
- 二维前缀和: $s[i][j] = s[i][j - 1] + s[i - 1][j] - s[i - 1][j - 1] + a[i][j]$ ——四项
- 二维差分: $diff[i][j] = a[i][j] - a[i - 1][j] - a[i][j - 1] + a[i - 1][j - 1]$ ——四项

其实 n 维差分的项数，就是 $2n$ 项，这本质是容斥原理，在此不需要深究。每高一维，我们单个操作项数就增加了一倍，实际上时间也多花了一倍

多维前缀和计算技巧

对于二维前缀和，我们是通过容斥求出来的：

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        f[i][j] = f[i][j - 1] + f[i - 1][j] - f[i - 1][j - 1] + val[i][j];
    }
}
```

还有另外一种求法，更好写也更通用（2维就算2次，3维就算3次，以此类推）：

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        f[i][j] += f[i][j - 1];
    }
}
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        f[i][j] += f[i - 1][j];
    }
}
```

前缀和总结

只要我们看到不带修改的静态问题

不管他是要求区间和，还是求树链和，还是求多维空间和，还是求子集和，还是求XX和，无非就是前缀和一下

不管他是要修改区间，还是修改树链，还是修改多维空间，还是修改子集，还是修改XX，无非就是差分一下

二维前缀和

P1719 最大加权矩形

一个 $N \times N$ 矩阵，求其中的总值最大的子矩阵。

数据范围： $1 \leq N \leq 120$

本题是最大子段和的二维版本。我们枚举所有子矩阵：

```
int ans = 0;
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        for (int x = 1; x <= i; x++) {
            for (int y = 1; y <= j; y++) {
                ans = max(ans, sum[i][j] - sum[x - 1][j] - sum[i][y - 1] + sum[x - 1][y - 1]);
            }
        }
    }
}
```

时间复杂度 $O(n^4)$ ，有没有办法优化？

矩阵压缩

和上面最大子段和的贪心思想一样，如果有某个矩形的和已经是负的，那么没有必要去考虑包含这个矩形。

矩阵压缩是什么？我以一维为例：我现在有 $\{a, b, c\}$ 这个数组，我求它的最大子段和，就是求 $\{[a], [ab], [abc], [b], [bc], [c]\}$ 中的最大值。

我们可以枚举自第 k 行起到第 i 行的子矩阵，这时我们把每一列求和：``f[j] = a[i][j] - a[i - k][j]``，然后像刚才那样贪心即可：

```
for (i = 1; i <= n; ++i) {
    for (k = 1; k <= i; ++k) {
        int f[150] = {0}, dp[150] = {0}; // f[j] 表示压缩的矩阵第 j 列的值
        for (j = 1; j <= n; ++j) { // 其实可以不开数组，一个 f 就可以
            f[j] = a[i][j] - a[i - k][j]; // 求压缩的矩阵第 j 列的值
            dp[j] = maxn(dp[j - 1] + f[j], f[j]); // 动态规划
            ans = maxn(ans, dp[j]); // 更新答案
        }
    }
}
```

离散化

离散化就是把很大的数映射到小的数

P1496 火烧赤壁

有一个白色的数轴， n 个操作，每次操作把区间 $[a_i, b_i]$ 染黑，求最后有多少区间被染黑了？

$n \leq 20000, -10^9 \leq a_i, b_i \leq 10^9$

提示：int占4个字节。 $1\text{MB} = 10^3\text{KB} = 10^6\text{B}$ 即4MB可以开 10^6 个int

我们可以把离散又很大的数映射到比较小的数上：

1	2	3	4	5
-10^3	1	10^5	10^6	10^9

我完全可以像上表一样映射，这样操作的值就很小了，哪怕每一次染黑，都要循环 n 个点， n^2 的复杂度也不算太大。最后再还原就好了。

将不方便处理的空间信息，映射到我们方便处理的空间之中，这是一种线性转化的思想。

离散化实现

首先，我们需要建立上一张幻灯里的那张表。我们可以用数组来表示和存储：

```
cin >> n;
int cnt = 0;
for (int i = 1; i ≤ n; i++) {
    cin >> l[i] >> r[i];
    num[+cnt] = l[i];
    num[+cnt] = r[i];
}
sort(num + 1, num + 1 + cnt);
int size = unique(num + 1, num + 1 + cnt) - num;
```

把所有数加到`sum`数组里面，然后排序、去掉重复的数，我们就得到了一张表。我们之后就把所有数映射为它们在这个数组对应的下标。

P1496 火烧赤壁

继续分析这道题目：

这道题是每次都对区间做操作，且中途没有询问，属于询问与修改分离，我们应该做什么？

```
for (int i = 1; i <= n; i++) {
    int nl = lower_bound(num + 1, num + size, l[i]);
    int nr = lower_bound(num + 1, num + size, r[i]);
    diff[nl]++;
    diff[nr]--;
}
int begin = -1;
for (int i = 1; i <= cnt; i++) {
    diff[i] += diff[i - 1];
    if (diff[i] && begin == -1) {
        begin = i;
    }
    if (!diff[i] && begin != -1) {
        ans += num[i] - num[begin];
        begin = -1;
    }
}
```

双指针

双指针中的“指针”，指的不是C语言中的“指针”

两个指针一般指代一个区间的左右端点。

我们往往是固定一个端点，然后从小到大枚举另一个端点，然后利用问题的单调性求解问题

输入一整数数列 $a_1, a_2, a_3, \dots, a_n$ ，以及另一个整数 k ，求区间 $[i, j]$ 使得 $a_i, a_{i+1}, \dots, a_k = k$.

这是上节课讲解的尺取算法例题

我们可以通过两个指针，从左到右的扫描判断，做出这道题。

那么我要问的是，为什么这道题可以使用双指针求解呢？

双指针的本质

其实，这道题之所以可以使用双指针求解，是因为：

假如我固定区间的左端点，由于 $a_i \geq 0$ ，随着右端点向右移，区间和只会单调上升（或者不变）

我们称这个性质为：**单调性**

由于单调性，当前的区间和大于答案 k 时，右端点再往右移动，答案只会更大，绝对不会等于 k 。所以我们可以把后面的答案全部舍弃。

我们称这个性质为：**局部舍弃性**

只要具有单调性和局部舍弃性，就可以使用双指针或者二分解决。

能双指针一定能二分吗？一般来说是的。

——这道题用二分怎么做？

先求前缀和，然后二分查找右端点。

桶

桶一般被用于求解存在性问题。

输入一整数数列 $a_1, a_2, a_3, \dots, a_n$, 以及另一个整数 k , 求区间 $[i, j]$ 使得 $a_i, a_{i+1}, \dots, a_k = k$.

思考：如果 a_i 可以为负数呢？

——失去了单调性和局部舍弃性，从左往右加有可能越加越小

新思路：前缀和+桶

参考前面的P1115 最大子段和，我们需要找 l, r 满足 $k = \text{sum}[r] - \text{sum}[l - 1]$

对于每个确定的 r ，我们要确认它前面的前缀和中有无 $\text{sum}[r] - k$

怎样优化存在性问题？开布尔数组 $\text{exist}[n]$ ，每次将出现的值置为1.

Acknowledgment

感谢师兄GDUT20ZYL(卓耀龙)，他完成了大部分的PPT。

感谢技术部里帮忙找虫的同学。

感谢大家前来上课。