算法简单复习

算法简单复习

1. 算法概论
    1.1 最大公约数的分析
    1.2 算法常识
2. 算法分析基础
    2.1 增长速度表
    2.2 算法分析框架概述
    2.3 渐进复杂度
    2.4 常见的求和公式
    2.5 实例
3. 暴力法
    3.1 暴力法为什么要讲?
4. 递归法
    4.1 常见的递归法求解的问题
    4.2 递归解法的类型
5. 分治法
    5.1 怎样的问题适合分治法
    5.2 一个常规的分治法
    5.3 大数相乘问题
    5.4 矩阵相乘
    5.5 二分搜索和平均查找时间
    5.6 归并排序
    5.7 快速排序
    5.8 棋盘覆盖问题
    5.9 分治法总结
6. 减治法
    6.1 啥是减治法
    6.2 减治法的类型
    6.3 插入排序
    6.4 拓扑排序
    6.5 二分查找的递归形式
    6.6 假硬币问题
    6.7 俄罗斯农民乘法问题
    6.8 减治法回顾
7. 变治法
    7.1 三种变治法
    7.2 预排序

7.3 高斯消去法
7.5 堆和堆排序
7.6 霍纳法则（Horner's Rule）、秦九韶算法
7.7 问题化简
7.8 变治法的总结
8. 回溯法和分支限界法
8.1 穷举搜索算法
8.2 DFS和BFS的图搜索算法
8.3 回溯法
8.4 分支限界法
8.5 回溯法和其他方法之比较
9. 贪心法
9.1 贪心法简介
9.2 活动安排问题
9.3 贪心算法的基本要素
9.4 连续背包问题
9.5 单源最短路径 - Dijkstra
9.6 最小生成树 - Kruskall & Prim
10. 动态规划
10.1 动态规划的基本概念
10.2 计算二项式系数（杨辉三角）
10.3 最长公共子序列
10.4 矩阵连乘问题
10.5 0-1背包问题（动态规划）
10.6 多阶段决策问题
10.7 矩阵的传递闭包 - Warshall算法
10.8 多源最短路径：弗洛伊德算法
10.9 与贪心算法之区别
10.10 检查最优性原理的实例
10.11 小总结
谢谢阅读$Th_n^a k$ § $F\emptyset r\ R_e a^{di_n g}$

1. 算法概论

1.1 最大公约数的分析

欧几里得算法：

```
gcd(m,n) = gcd(n, m mod n) iteratively while n != 0

gcd(m,0) = m
```

连续整数检测法：

比较麻烦，先看看较小的能不能当gcd，不行的话，每次减1再判断。

1.2 算法常识

伪代码的介绍：

自然语言和编程语言相结合的产物

精确又简明

本课中：忽略变量的声明；用缩进来表示`if`和`while`等语句的范围；使用←表示赋值。

算法的特点：

输入、输出、确定、有穷、可行

算法的定义：

在有限的时间内，对问题求解的一个清晰的指令序列。

本章思考题：

证明欧几里得算法对每一对正整数都成立。

2. 算法分析基础

2.1 增长速度表

$1 < log_2 n < n < nlog_2 n < n^2 < n^3 < 2^n < n!$

2.2 算法分析框架概述

时间复杂度是通过计算基本操作（basic operations）得到的

时空复杂度是关于输入规模（n）的函数

2.3 渐进复杂度

设要分析的对象是$T(n)$，当

$$lim_{n \to \infty} \frac{T(n) - t(n)}{T(n)} \to 0$$

$t(n)$就是$T(n)$的渐进形式 *(asymptotic state)*

$t(n)$也叫作算法A的 $n \to \infty$ 渐进复杂度

- $t(n)$只考虑$T(n)$的 *leading term*
- 忽略常系数

符号	意义	例子
$O(g(n))$	增长得不比 g 快的一系列函数的集合	$10n^2 \in O(n^2), 10n^2+2n \in O(n^2)$, $100n+5 \in O(n^2)$, $5n+20 \in O(n)$
$\Omega(g(n))$	增长得至少和 g 一样快的一系列函数的集合	$10n^2 \in \Omega(n^2)$, $10n^2+2n \in \Omega(n^2)$, $10n^3 \in \Omega(n^2)$
$\Theta(g(n))$	增长得和 g 一样快的一系列函数的集合	$10n^2 \in \Theta(n^2)$, $an^2+bn+c \in \Theta(n^2)$ $a>0$, $\frac{(n)(n-1)}{2} \in \Theta(n^2)$, $n^2+\lg n \in \Theta(n^2)$

符号	意义
$o(g(n))$	增长得比g慢的一系列函数的集合（大O可以取等）
$w(g(n))$	增长得比 g 快的一系列函数的集合（大Ω可以取等）

渐进表达式的运算性质：

反身性、传递性、互对称性、对称性、大O加法乘法结合律

用极限来分析：

求

$$lim_{n\to\infty}\frac{T(n)}{g(n)}$$

结果有三种：

$$\begin{cases} 0, T(n)<g(n) \\ c>0, T(n)=g(n) \\ \infty, T(n)>g(n) \end{cases}$$

case1&2: $T(n) \in O(g(n))$

case2: $T(n) \in \Theta(g(n))$

case2&3: $T(n) \in \Omega(g(n))$

算不出来可以用洛必达法则

一些注意事项：

所有对数函数都属于$\Theta(\log n)$

所有多项式都属于$\Theta(n^k)$，k是多项式的次数

指数函数根据底数的不同，增长的速率不同。

2.4 常见的求和公式

$$\sum_{i=l}^{u}(a^i \pm b^i) = \sum_{i=l}^{u} a^i \pm \sum_{i=l}^{u} b^i$$

$$\sum_{i=l}^{u} 1 = u - l + 1$$

$$\sum_{i=0}^{n} i = \frac{n^2}{2} \in \Theta(n^2)$$

2.5 实例

汉诺塔，递归解法

```
void hanoi(int n, int a, int b, int c){
    if(n>0)
        {
            hanoi(n-1,a,c,b);
            move(a,b);
            hanoi(n-1,b,a,c);
    }
}
```

其基本操作数为：

$C(n) = 2C(n-1) + 1 = 2^n - 1$

数十进制数的二进制形式有几位数字

```
ALGORITHM BinRec(n)
    //Input: A positive decimal integer n
    //Output: The number of bin. digits in n's bin representation
    if n==1 return 1
    else return BinRec(floor(n/2)+1)
```

基本操作：$+1$的动作

解决这种递归的方法：令$n = 2^k$，但是这种做法似乎忽略了许多个n的值。*Smoothness Rule* 告诉我们这是ok的。

易知$A(n) = \log_2 n = \Theta(\log n)$

3. 暴力法

3.1 暴力法为什么要讲?

它给解决问题提供了一个下界。

对于这些问题，都可以用暴力法：

排序 - 选择排序、冒泡排序

穷举 - （哈密顿图）TSP问题、背包问题、分派工作：NP-Hard问题

4. 递归法

4.1 常见的递归法求解的问题

- 阶乘
- 斐波那契
- 汉诺塔问题
- 排列问题

4.2 递归解法的类型

Decrease-by-one

$T(n) = T(n-1) + f(n)$

Decrease-by-a-constant-factor

$T(n) = aT(\frac{n}{b}) + f(n)$

The Master Theorem

$T(n) = aT(\frac{n}{b}) + f(n)$, where $f(n) \in \Theta(n^k)$

a. $a < b^k$, then $T(n) \in \Theta(n^k)$

b. $a = b^k$, then $T(n) \in \Theta(n^k \log n)$

c. $a > b^k$, then $T(n) \in \Theta(n^{\log_b a})$

5. 分治法

5.1 怎样的问题适合分治法

原始问题的解可以规模均衡地分成相同类型的互相独立的最小问题地解，将后者做合并得到原始问题的解

5.2 一个常规的分治法

假设：*把一个规模为n的问题，分成a个规模为$\frac{n}{b}$的子问题。a是每次分叉要解决的问题个数（分叉数）*

则可以得到

$$T(n) = \begin{cases} O(1), n = 1 \\ aT(\frac{n}{b}) + f(n), n > 1 \end{cases}$$

$f(n)$是我们在分叉和合并的时候需要花的时间。

5.3 大数相乘问题

给两个n位数字，做乘法。

- 蛮力法带来n^2的时间复杂度

一次分治算法的尝试

分治法解决，将两数这样分解：

$$X = A * 10^{\frac{n}{2}} + B$$

$$Y = C * 10^{\frac{n}{2}} + D$$

其中A, B, C, D都是$\frac{n}{2}$位数字

那么有：

$$X * Y = A * C * 10^n + (A * D + B * C) * 10^{\frac{n}{2}} + B * D$$

因为$*10^n$或者$*10^{\frac{n}{2}}$都不能算作和n的规模有关，只能算作多加了几个零，或者最后列竖式的时候左移了几个位置。所以考虑这个子问题规模的时候，只用看到把**1**对n位数相乘，变成了**4**对$\frac{n}{2}$位数相乘的问题。每对需要的时间复杂度是$O(n)$。

得出了

$$T(n) = \begin{cases} O(1), n = 1 \\ 4T(\frac{n}{2}) + O(n), n > 1 \end{cases}$$

之后，利用4.2中的公式，可以很快得到，

$T(n) \in \Theta(n^{\log_2 4})$

即$\Theta(n^2)$。

并没有啥改进啊。

Karatsuba算法（卡拉楚巴算法）

分成

$$X = A * 10^{\frac{n}{2}} + B$$

$$Y = C * 10^{\frac{n}{2}} + D$$

$$X * Y = A * C * 10^n + (A * D + B * C) * 10^{\frac{n}{2}} + B * D$$

之后，

观察$(A * D + B * C)$这一项，发现可以由

$$(A + B) * (C + D) - A * C - B * D$$

得到。那么上式可以这样变换一下：

$$X * Y = A * C * 10^n + [(A + B) * (C + D) - A * C - B * D] * 10^{\frac{n}{2}} + B * D$$

发现只用计算

$A * C, (A + B) * (C + D), B * D$这三项就可以得到。直接把四次乘法变成了三次乘法。

抓大放小，可以得到：

$$T(n) = \begin{cases} O(1), n = 1 \\ 3T(\frac{n}{2}) + O(n), n > 1 \end{cases}$$

这样的话，时间复杂度是

$T(n) \in \Theta(n^{\log_2 3})$（这里也可以作$n = 2^k$的变换）

即$\Theta(n^{1.585})$。

快速傅里叶变换算法

太难了，略。

5.4 矩阵相乘

蛮力法，要用$O(n^3)$才能解决（n是矩阵的阶数）。

一次分治算法的尝试：

将最终结果分成四部分，

得到：

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

即

$$\begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

易知这样的分治，得到的还是

$$T(n) = \begin{cases} O(1), n = 1 \\ 8T(\frac{n}{2}) + O(n^2), n > 1 \end{cases}$$

用公式得出$T(n) \in \Theta(n^{\log_2 8})$，*没有甚么区别啊。*

史特拉森（Strassen）演算法：

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

设每一个M都是$\frac{n}{2}$阶的矩阵线性变换得到的，那么原始可以写成：（M具体可以在*PPT-Chap5*的第13页找到）

$$\begin{bmatrix} M_5 + M_4 - M_2 + M_6 & M_1 + M_2 \\ M_3 + M_4 & M_5 + M_1 - M_3 - M_7 \end{bmatrix}$$

可以得到：

$$M(n) = \begin{cases} O(1), n = 1 \\ 7M(\frac{n}{2}), n > 1 \end{cases}$$

用公式得出$T(n) \in \Theta(n^{\log_2 7}) = \Theta(n^{2.807})$

注意事项：

如果阶数并不是2的指数，矩阵可以通过填0来得到。

一般来说，当这个算法跨越了某个交点之后，就可以转换成暴力法了。

5.5 二分搜索和平均查找时间

问题描述：在一个非降序的数组中查找元素，存在返回位置，不存在返回-1.

迭代求法

```
ALGO BinSearchIt(A[0...n-1], K)
l = 0; r = n-1
while l<=r do
    m = floor((l+r)/2)
    if K == A[m] return m
    else if K < A[m] r = m - 1
    else r = m + 1
return -1
```

递归求法

```
ALGO BinSearchRec(A[0...n-1], K, l, r)
if l > r return -1
else
    m = floor((l+r)/2)
    if K == A[m] return m
    else if K < A[m] return BinSearchRec(A[0...n-1], K, l, m - 1)
    else return BinSearchRec(A[0...n-1], K, m + 1, r)
return -1
```

分析：

最坏情况下，$\Theta(\log n)$；最好情况下，$O(1)$，一次找到。

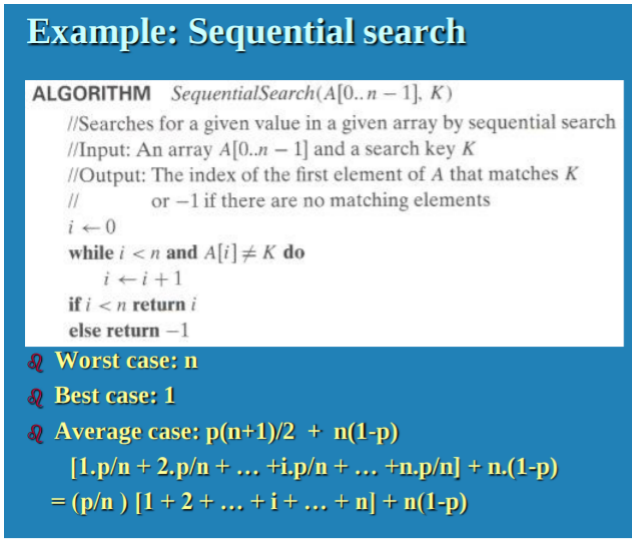
一般情况下，要分析平均时间。（图片在*PPT-Chap5*第23页）

$$A(n) = \frac{1}{n}\sum_{i=1}^{k} i * 2^{i-1} \approx \log(n+1) - 1$$

这个式子就是计算一棵搜索树里将每个结点寻找的次数加一起，再除以总数，得到平均值。

- 二元比较树的表示方法，分析方法
- 内结点、外结点代表啥

逐个查找法，平均查找时间这样算：



这里的P是查找某个元素的概率。一般来说，如果找每个元素的概率都相同，应该是$\frac{1}{n}$。

5.6 归并排序

```
ALGO MergeSort(A[0...n-1])
//Input:
//Output: ~non-decreasing order~
if n>1
    copy A[0...floor(n/2)-1] to B[0...floor(n/2)-1]
    copy A[floor(n/2)...n-1] to C[0...ceiling(n/2)-1]
    MergeSort(B[0...floor(n/2)-1])
    MergeSort(C[0...ceiling(n/2)-1])
    Merge(B,C,A)
```

```
ALGO Merge(B[0...p-1],C[0...q-1],A[0...p+q-1])
//将两个已排好序的数组合并为一个数组
//Input:
//Output:A[0...p+q-1]
i = 0; j = 0; k = 0;
while i < p and j < q do
    if B[i] <= C[j]
        A[k] = B[i]; i = i + 1
    else A[k] = C[j]; j = j + 1
    k = k + 1
if i == p
    copy C[j...q-1] to A[k...p+q-1]
else copy B[i...p-1] to A[k...p+q-1]
```

可以分析一下最深处的一次递归，是把两个数字排好序，放到一个大小为2的数组中。

时间复杂度比较好分析：

$$\begin{cases} C(n) = 2C(n/2) + C_{merge}(n)\ ,\ n>1 \\ C(n) = 0,\ n=1 \end{cases}$$

$$where\ C_{merge}(n) = n - 1$$

这里的C_{merge}就是上面的`Merge(,,)`做的事情。代入之前的公式（$k = 1, a = 2, b = 2$），易得：

$$C(n) \in \Theta(n * \log n)$$

5.7 快速排序

概括一下快速排序的中心思想：所谓分类（Partition）就是把一个数组，分成大于某个数和小于某个数的两个子数组。这时，“某个数”的位置已经被排好。再递归对某个数前后的数组进行Divide操作，最后的大数组自动排好，不用合并。

从左到右扫描时，用`i`索引。遇到小于等于基准元素的，自增跳过。遇到第一个大于基准元素的就停下来。

从右到左扫描时，用`j`索引。和上面的刚好相反。

此时，有两种情况导致扫描停止：

`case 1`

`i,j`没有相遇，也即`i<j`

交换元素

`case 2`

`i,j`相遇也即`i=>j`

本轮划分扫描结束，将基准点放在它应该处的位置上（这里应该是将基准点元素和`a[j]`作了交换）：`a[j] = pivot`，返回基准点的位置：`return j`

在大递归调用中，再把基准点坐标作为中点，进行递归的划分。

```
ALGO QuickSort(A[l...r])
//Input:
//Output: 非降序排列的子数组。
if l<r
    s = Partition(A[l...r])
    QuickSort(A[l...s-1])
    QuickSort(A[s+1...r])
```

```
ALGO Partition(A[l...r])
//Input:
//Output: pivot's location
i = l + 1; j = r ;
pivot = A[l]
while true do
    while A[i] < pivot
```

```
        i = i + 1
    while A[j] > pivot
        j = j -1
    if i >= j break
    Swap(A[i], A[j])
a[l] = a[j]
a[j] = pivot
return j
```

对快排算法的分析：

一次划分（Partition）需要的比较次数：$n+1$或者n。前者是两指针已经相遇并穿过了对方，后者是刚好相遇。

在完美情况下，我们选择的基准每次都可以被安插在这一段的中间：

$$\begin{cases} C(n) = 2C(n/2) + \Theta(n)\ ,\ n>1 \\ C(n) = 0,\ n=1 \end{cases}$$

代公式（或者也可以用$n = 2^k$替换）得到：

$$C(n) \in \Theta(n * \log n)$$

在最坏情况下，这是一个倒序数组。

$$C(n) = C(n-1) + \Theta(n)$$

易知结果是

$$C(n) \in \Theta(n^2)$$

这里是这样算的：

$$C = (n+1) + n + \ \ldots\ + 3 = (n+1) * (n+2)/2 - 3 = \Theta(n^2)$$

以左值为基准，`i`停在左值旁边的第一个元素`A[1]`，`j`一直滑到`A[0]`，一共`n+1`次比较。比到`n=2`。

随机划分的情况下，据说*《算法导论》第七章*有证明，可以得到是：

$$C(n) \in \Theta(n * \log n)$$

5.8 棋盘覆盖问题

问题描述：有点长，可以看*PPT-Chap5-Page48*。

分治策略：

把$2^k * 2^k$的棋盘分成四个$2^{k-1} * 2^{k-1}$的小小棋盘。在**没有特殊方格**的三个小棋盘的汇合处，用一个L型骨牌覆盖。再递归进行分割。这时被L型骨牌覆盖的部分，当作特殊方格处理。

分析：

$$T(n)=\begin{cases}O(1),k=0\\4T(k-1)+O(1),k>0\end{cases}$$

$$T(n)=O(4^k)$$

5.9 分治法总结

分治法的时间效率满足：$T(n)=aT(\frac{n}{b})+f(n)$

归并排序在任何情况下时间效率都是：$\Theta(n\log n)$。

快排的最快时间效率和最差时间效率。

折半查找的时间效率。

n位大整数乘法的最优时间效率。

Strassen算法也是分治算法，它的时间效率...

算法简单复习

1. 算法概论
    1.1 最大公约数的分析
    1.2 算法常识
2. 算法分析基础
    2.1 增长速度表
    2.2 算法分析框架概述
    2.3 渐进复杂度
    2.4 常见的求和公式
    2.5 实例
3. 暴力法
    3.1 暴力法为什么要讲?
4. 递归法
    4.1 常见的递归法求解的问题
    4.2 递归解法的类型
5. 分治法
    5.1 怎样的问题适合分治法
    5.2 一个常规的分治法
    5.3 大数相乘问题
    5.4 矩阵相乘
    5.5 二分搜索和平均查找时间
    5.6 归并排序
    5.7 快速排序
    5.8 棋盘覆盖问题
    5.9 分治法总结
6. 减治法
    6.1 啥是减治法

6.2 减治法的类型
6.3 插入排序
6.4 拓扑排序
6.5 二分查找的递归形式
6.6 假硬币问题
6.7 俄罗斯农民乘法问题
6.8 减治法回顾
7. 变治法
7.1 三种变治法
7.2 预排序
7.3 高斯消去法
7.5 堆和堆排序
7.6 霍纳法则（Horner's Rule）、秦九韶算法
7.7 问题化简
7.8 变治法的总结
8. 回溯法和分支限界法
8.1 穷举搜索算法
8.2 DFS和BFS的图搜索算法
8.3 回溯法
8.4 分支限界法
8.5 回溯法和其他方法之比较
9. 贪心法
9.1 贪心法简介
9.2 活动安排问题
9.3 贪心算法的基本要素
9.4 连续背包问题
9.5 单源最短路径 - Dijkstra
9.6 最小生成树 - Kruskall & Prim
10. 动态规划
10.1 动态规划的基本概念
10.2 计算二项式系数（杨辉三角）
10.3 最长公共子序列
10.4 矩阵连乘问题
10.5 0-1背包问题（动态规划）
10.6 多阶段决策问题
10.7 矩阵的传递闭包 - Warshall算法
10.8 多源最短路径：弗洛伊德算法
10.9 与贪心算法之区别
10.10 检查最优性原理的实例
10.11 小总结
谢谢阅读$Th_n^a k$§ $F\emptyset r\ R_e a^{di_n g}$

6. 减治法

6.1 啥是减治法

如果一个问题“一个实例”的解和它的“较小实例”的解存在某种关系，就可以用减治技术。*感觉还是不太懂啊。*

6.2 减治法的类型

减常量

插入排序（较小实例：`A[0...n-2]`，问题实例:`A[0...n-1]`）、拓扑排序（大有向图总是通过小有向图在孤立结点里找到一个结点完成的）。

减常数因子

二分搜索、假硬币问题、俄罗斯农民乘法问题

减可变因子

欧几里得算法求`gcd`

6.3 插入排序

```
ALGO InsertionSort(A[0...n-1])
for i = 1 to n - 1 do
    v = A[i]
    j = i - 1
    while j >= 0 and A[j] > v do
        A[j+1] = A[j]
        j = j - 1
    A[j+1] = v
```

分析：

$$C_{worst}(n) = \sum_{i=1}^{n-1}\sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2)$$

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n-1 \in \Theta(n)$$

$$C_{ave}(n) \approx \frac{n^2}{4} \in \Theta(n^2)$$

6.4 拓扑排序

DFS方法

基本思想就是，从第一个结点开始，一直搜索直到一个没有任何出度或所有出边都被找过的结点，入栈。

这一段来自力扣的代码写得相当清楚：

```
class Solution {
private:
    // 存储有向图
    vector<vector<int>> edges;
    // 标记每个结点的状态：0=未搜索，1=搜索中，2=已完成
    vector<int> visited;
    // 用数组来模拟栈，下标 0 为栈底，n-1 为栈顶
    vector<int> result;
    // 判断有向图中是否有环
    bool valid = true;

public:
    void dfs(int u) {
        // 将结点标记为「搜索中」
        visited[u] = 1;
        // 搜索其相邻结点
        // 只要发现有环，立刻停止搜索
        for (int v: edges[u]) {
            // 如果「未搜索」那么搜索相邻结点
            if (visited[v] == 0) {
                dfs(v);
                if (!valid) {
                    return;
                }
            }
            // 如果「搜索中」说明找到了环
            else if (visited[v] == 1) {
                valid = false;
                return;
            }
        }
        // 将结点标记为「已完成」
        visited[u] = 2;
        // 将结点入栈
        result.push_back(u);
    }

    vector<int> findOrder(int numCourses, vector<vector<int>>&
prerequisites) {
        edges.resize(numCourses);
        visited.resize(numCourses);
```

```
        for (const auto& info: prerequisites) {
            edges[info[1]].push_back(info[0]);
        }
        // 每次挑选一个「未搜索」的结点，开始进行深度优先搜索
        for (int i = 0; i < numCourses && valid; ++i) {
            if (!visited[i]) {
                dfs(i);
            }
        }
        if (!valid) {
            return {};
        }
        // 如果没有环，那么就有拓扑排序
        // 注意下标 0 为栈底，因此需要将数组反序输出
        reverse(result.begin(), result.end());
        return result;
    }
};

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/course-schedule-ii/solution/ke-cheng-biao-ii-by-leetcode-solution/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

分析（图的深度优先搜索时间复杂度）：

$O(E+V)$

V来自每个点进行遍历的外循环，E来自每个点的所有出边。

删源结点法（Source Removal Method）

每次寻找入度为0的结点，加入结果，从图中删除。重复（减治法）。

6.5 二分查找的递归形式

见 5.5.

6.6 假硬币问题

问题描述：n个硬币，一个是假的，用天平鉴别出假的。

减常数因子：每次对半砍

开始之前，如果n是奇数，每次留下一个，如果数量为$\frac{n-1}{2}$的两堆硬币一样重，说明拿出的那个是假币，问题就解决了。否则将剩下的硬币分成两堆去比较重量。

分析：

$$T(n)=\begin{cases}0,n=1\\T(\frac{n}{2})+1,n>1\end{cases}$$

可知：

$$T(n)\in O(\log_2 n)$$

减常数因子：每次分成三堆

比较两个n/3的，剩下的当第三堆。分析：

$$T(n)=\begin{cases}0,n=1\\T(\frac{n}{3})+1,n>1\end{cases}$$

可知：

$$T(n)\in O(\log_3 n)$$

6.7 俄罗斯农民乘法问题

问题描述：想要获得n*m，将式子变为：

$$\begin{cases}n*m=\frac{n}{2}*2m,\ n\ is\ even\\n*m=\frac{n-1}{2}*2m+m,\ n\ is\ odd\end{cases}$$

```
ALGO RPM(m,n)
res = 0;
while n > 0 do
    if n is odd
        res = res + m;
    m = m * 2;
    n = n / 2;
return res
```

分析：（在StackOverflow上有详细到比特操作的复杂度分析，此处按照减治法的分析方法简化一下：设n为小值）

$$T(n)=\begin{cases}O(1),n=1\\T(\frac{n}{2})+O(1),n>1\end{cases}$$

可知：

$$T(n)\in O(\log_2 n)$$

6.8 减治法回顾

其实就是上面几条，都不多。注意插入排序和拓扑排序。

7. 变治法

7.1 三种变治法

- 实例化简
- 改变表现形式
- 问题化简

7.2 预排序

实例化简

许多问题，如果把没有序的输入变成有序的输入，就会简单得多。

- 查找、搜索
- 计算中值
- 元素独特性

也要选一个合适的排序算法。

元素独特性检查

$$C(n) = C_{sort}(n) + C_{scan}n = \Theta(n \log n) + \Theta(n) = \Theta(n \log n)$$

计算众数

和上面的情况一样

查找元素

见5.5。

7.3 高斯消去法

问题描述：求解线性方程组，给一个系数矩阵，用高斯消元法。在我们的学习中，第一个阶段是将一个矩阵完成**初等变换**成为一个上三角矩阵，第二个阶段是用**向后替换算法**求解。

向后替换算法：在最后一个方程中直接得到x_n，再在倒数第二个式子中（批量）替换。

第一阶段：先让第二到最后一个方程的a_{11}系数为零，重复操作形成一个上三角矩阵。

第二阶段：直接代入、批量替换。

思考两个问题：

如果A[i][i]为0：将这一行换成下方某个不为零的。

如果A[i][i]过小，下面的用它当分母可能会过大：将这一行替换成绝对值最大的。

```
ALGO GaussElimination(A[1...n,1...n],b[1...n])
for i= 1 to n - 1 do
    for j = i + 1 to n do
        temp = A[j,i]/A[i,i]
        for k = i to n + 1 do
            A[j,k] = A[j,k] - A[i,k] * temp
        end
    end
end
//下面是倒退求解具体的x
for i = n downto 1 do
    x[i] = b[i]
    for j = i + 1 to n do
        x[i] = x[i] - A[i,j]*x[j]
    end
    x[i] = x[i]/A[i,i]
end
```

分析：

$$\Theta(n^3)+\Theta(n^2)=\Theta(n^3)$$

这个三次方虽然可以猜到，但是算还是不会算。三个求和嵌套，主注意换元和数学计算。

记得这个公式：

$$\sum_{j=1}^{n} j^2=\frac{n(n+1)(2n+1)}{6}$$

这个公式记不住好像也没关系，可以通过这个推（移项，累加）：

$$(n+1)^3=n^3+3n^2+3n+1$$

高斯消元的应用：

LU分解、求逆、计算行列式

7.5 堆和堆排序

改变表现形式体现在哪？用堆来代替一个优先级队列，可以获得更快的增删查的性能。

堆的特点：

是一个完全二叉树、（最大堆）结点的值大于其所有孩子的值

堆的性质：

高度：$\lfloor \log_2 n \rfloor$

用数组代表堆：

- 非叶结点都在数组的前$\lfloor n/2 \rfloor$个元素中，叶结点都在后$\lceil n/2 \rceil$个结点中。
- 关系：`parent(i) : return floor(i/2); left(i) : return 2*i; right(i) : return 2*i+1`。

堆的构造：

自底向上：

```
ALGO HeapBottomUp(H[1...n])
//input:
//Output: A heap.
for i = floor(n/2) downto 1 do
    MaxHeapify(H,i)
```

```
ALGO MaxHeapify(H,i)
l = left(i);
r = right(i);
if l <= r and H[l] > H[i]
    largest = l;
else largest = i
if r <= n and H[r] > H[largest]
    largest = r
if largest != i
    swap(H[i], H[largest])
    MaxHeapity(H,largest)
```

自顶向下：$O(\log n)$

删除堆顶：

与最后一个结点K交换，长度减一，`siftdown(K)`。$O(\log n)$

堆排序：

删n次堆顶即可。

由于要做n次山堆顶操作，时间复杂度是$O(n \log n)$

7.6 霍纳法则（Horner's Rule）、秦九韶算法

计算多项式取某个x_0的值。改变形式的变治法之典型。

将一个多项式变成这样的形式：

$$p(x) = (\dots(a_n x + a_{n-1})x\dots)x + a_0$$

```
ALGO QinJiuShao(P[0...n],x)
//Input: n次升幂排序多项式的系数数组，以及定点x
//Output: 多项式在x处的取值
res = p[n]
for i = n - 1 downto 0 do
    res = x * res + p[i]
return res
```

7.7 问题化简

> *解析几何的根本思想就是把几何问题化简为代数问题。*

求最小公倍数

变治法求解：`lcm(m,n) = (m*n)/(gcd(m,n))`

线性规划（高中的那种用不等式数形结合的问题）

投资问题、背包问题、缩减为图的抽象问题（狼羊菜人过河）

这种可以缩减为图的问题，一般都可以用状态图表示每一个状态。

7.8 变治法的总结

其他还行，可以再看看堆。

8. 回溯法和分支限界法

8.1 穷举搜索算法

用来求解TSP问题的时间下限，列所有的可能，选最合适的。

8.2 DFS和BFS的图搜索算法

```
ALGO DFS(G)
//I
//O
stack.push(v[0])
while not stack.empty
    stack.pop(s)
    if not visited(s)
        res.push_back(s)
        s.visited = true
    for each vertex w adjacent to s do
        if not visited(w)
            stack.push(w)
return res
```

```
ALGO BFS(G)
//I
//O
queue.en(v[0])
s.visited = true
while not queue.empty
    queue.pop(s)
    res.push_back(s)
    for each w adjacent to s
        if not visited(w)
            queue.en(w)
return res
```

分析：

BFS和DFS都有：

邻接矩阵：$\Theta(|V|^2)$，邻接表：$\Theta(|V|+|E|)$

8.3 回溯法

回溯法的算法框架：递归回溯、迭代回溯、子集树、排列树

系统性与跳跃性：在包含问题所有解的解空间中，按照DFS从根结点出发搜索解空间树（系统性）。至解空间树任一结点时，判断该结点为根的子树是否包含问题的解。不包含直接跳过该结点为根的子树，向其祖先结点回溯；否则继续DFS（跳跃性）。

问题的解空间：一般一个问题的解能通过一个n元式表达最好；这个式子里每个分量的取值确定；满足某种施加在各个分量的约束；满足显式约束条件的所有多元组，构成该实例的一个解空间。

两种常用的解空间树：

子集树：当问题是从n个元素的集合S中找出满足性质的子集时相应的解空间。一般有2^n个叶子结点，总结点数为$2^{n+1}-1$，遍历子集树的时间为$\Omega(2^n)$

排列树：问题是n个元素的某种排列。一般有$n!$个叶子结点，遍历子集树的时间为$\Omega(n!)$

TSP问题、N皇后问题

0-1背包问题

```
ALGO KSbyBacktrack(int i)
//Input: ith item
//Output: x[] as result.
if i > n
    bestv = min(bestv, cv)
    output(x);
else
    if cw + w[i] <= capacity //左放右不放
        x[i] = 1
        cw += w[i]
        cv += v[i]
        KSbyBacktrack(i+1)
        cw -= w[i]
        cv -= v[i]
    x[i] = 0
    KSbyBacktrack(i+1)
```

有限界函数的算法，基本思想是如果cv+r<=bestv，加了都没有最好的好：直接不加了，裁剪掉。怎样确定左右子树上界？简单粗暴的办法是啥都不管，左子树把当前和剩下的所有东西价值加一起，右子树跳过当前把剩下的所有东西加一起。

回溯法效率分析

各种约束函数在这里，很难分析这个问题，与很多因素都有关。有一点值得注意：如果其他条件都相同，可取值最少的x[i]优先，像背包问题这样一次只有两个取值，做一个决定就能把一大半砍掉。如果是很多取值，一次决定能减掉的枝数较少。

8.4 分支限界法

言简意赅：在每个拓展结点，生成一个下一步的活结点表，从中选择一个最有利的前进。

有一点值得注意：每个活结点仅有一次机会变成拓展结点，并不是只有一个拓展结点。不可能导出最优解的，将其删除，剩下的加入活动表（队列）中，再从活动表里选择结点再拓展。直到活动表为空。

*队列式分支限界法：*从活结点表中取结点的顺序和加入结点的顺序相同。活结点表就是个队列。

*优先队列：*可以用堆建立活结点表，每次查找一个最优的。

0-1背包问题

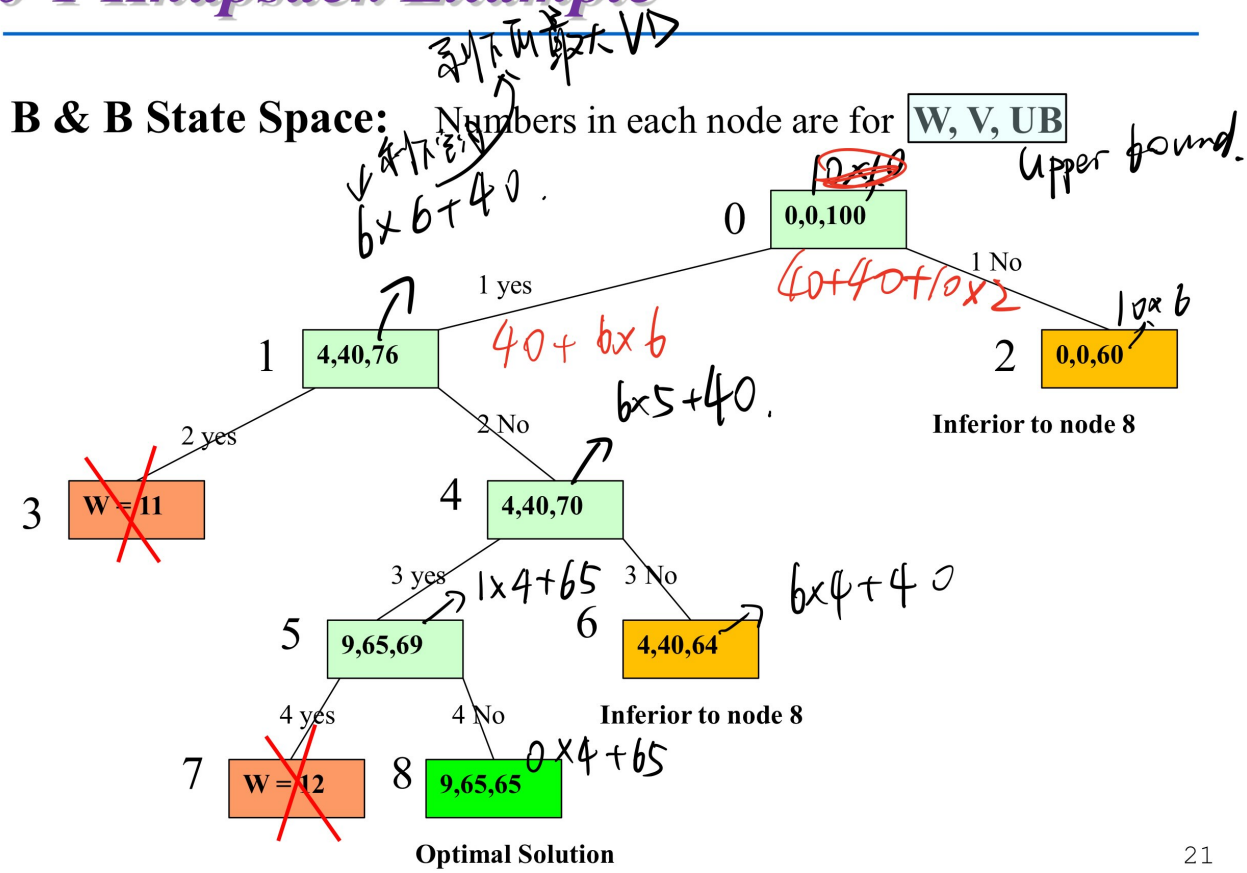
队列式这里就忽略了。

一种优先队列的分支限界算法：

```
ALGO  KSbyBB(n, capacity, w[], v[])
---
//key: 排序前，物品已经按照单位价值率的降序排列。
cw = 0
cv = 0
bestv = 0
i = 1
up = Bound(1)
while i != n+1
    //左结点：放
    if cw + w[i] <= capacity
        if cv + v[i] > bestv
            bestv = cv + v[i]
        AddLiveNode(up, cv+v[i], cw+w[i], true, i+1)
    up = Bound(i+1)
    if up >= bestv
        AddLiveNode(up, cv, cw, false, i+1)
    H -> DelMax(heap_peak)
    i = heap_peak.level
```

```
ALGO Bound(int i)
---
cap_left = c - cw
b = capacity
while i<=n AND w[i] <= cap_left
    cap_left = cap_left - w[i]
    b = b + v[i]
    i++
if i <= n // 执行完上一步，有可能剩下的空间不够放所有东西，在这一步用当前物品的价值率塞满剩余空间。
    b += v[i]/w[i] * cap_left
return b
```

这里要结合这个图一起看：



装载问题

这里不详细展开，和背包算法的区别在于，装上两艘货船。容易证明，首先将第一艘货船尽可能装满，再将剩余的集装箱装上第二艘货船，这样的策略得到的就是最优装载方案。突然发现，这不就是两个背包问题吗?

旅行售货员问题

可以用队列和优先级队列做。第i层结点有n-i个分支。如果用队列法，按照顺序求解就行，如果用优先级队列法，在每次将一个拓展结点的活结点都加入队列后，从堆顶取元素。这意味着不管是第几层的活结点，在这一步都有平等的机会。

对于优先级队列的分支限界法，一旦有一个叶结点成为当前扩展结点，就可以断言这是最优解，此时可以终止算法。

8.5 回溯法和其他方法之比较

与分支限界法相比：

求解目标：回溯法目标是找到解空间树中满足约束的所有解，分支限界是尽快地找出满足约束条件的一个解。

搜索方法：回溯法使用深度优先，而另一个使用广度优先或者最佳优先比较多。

拓展方式：分支限界法中，活结点只有一次机会成为拓展结点，一旦成为拓展结点就一次性产生所有子女结点。

存储空间的要求：分支限界法的存储空间比回溯法大，内存容量有限时，回溯法成功的可能性更大。

与穷举法相比

穷举法是先将一个接的各个部分完全生成再检查；回溯法是各个部分逐步生成，发现不满足约束条件就放弃，回到上一步。

9. 贪心法

9.1 贪心法简介

- 可行性
- 局部最优
- 不可撤回

贪心法对于找钱问题、最小生成树、单源最短路径、哈夫曼编码等有最优解，但是对于大多数问题，没有最优解，可以用于快速做出评估。比如TSP问题和背包问题。

当问题具有**最优子结构**的性质时，可以使用贪心算法。即原问题的最优解包含子问题的最优解。

贪心算法的基本步骤:

1. 选取度量标准，对于n个输入进行排序。
2. 如果输入和当前的部分最优解 $\notin$ 可行解，不把这个输入加到解中。
3. 否则将输入合并到解中
4. 持续到n个输入都被考虑完毕。

9.2 活动安排问题

问题描述:

有n个活动的集合，同一时间只能进行一个活动。每个活动都有一个时间区间，它的举行时间是一个半开的时间区间$[s_i, f_i)$。也即要求任何活动的时间区间不能相交。

我们的任务是，挑选出最大的相容活动子集合。目标是**让最多的活动能够相容地进行**。

```
ALGO GreedyActivitySelector(s[], f[], Select[])
//Input: s和f是成对的，是按照结束时间的升序排好的
//Output:
Select[1] = true
int j = 1
for i = 2 to n do
    if s[i] >= f[j]
        Select[i] = true
        j = i
    else
        Select[i] = false
end
return Select[]
```

分析：活动i与当前已经选择所有活动相容的充分必要条件是：开始时间s[i]不早于最近加入集合Select[]中的活动j的结束时间f[j]，即s[i]>=f[j]（这是一个半开区间，所以有等号）。

时间复杂度，取决于对于无序活动序列的排序时间。故为$O(n)+O(n\log n)=O(n\log n)$。

数学归纳法证明贪心可以求得整体最优解：

设E为已经按照非降序顺序排列好的活动集合。易知活动1有最早完成时间。

证明总存在一个以贪心选择开始的最优活动安排方案：

设$A\in E$是该问题的最优解，且A中的活动也按结束时间非降序排列。A中的第一个活动是K

- 若K=1，则A就是一个一贪心选择开始的最优解
- 若K>1，设$B=A-\{k\}\cup\{1\}$. 由于$f_1\leq f_k$，且A中的活动互为相容，故B中的活动也互为相容。（没有重复区间），且A是最优的，故B也是最优的。

如此证明了总是存在一个一贪心选择开始的最优活动安排方案

证明贪心算法能产生原问题的一个最优解：

即证明：若A是原问题的一个最优解，则$A'=A-\{1\}$是活动安排问题
$E'=\{i\in E,s_i\geq f_1\}$的一个最优解。

- 如果我们能找到E'的一个解B'，包含比A'更多的活动，那么将活动1加入到B'中将产生一个解B，包含比A更多的活动，与A的最优性相矛盾。
- 所以，每一步所做的贪心选择都将问题简化为一个更小的与原问题具有相同形式的子问题

当$n=1$时，已证明贪心算法成立。

假设：当$n=k$时，贪心算法已经选出了包含k个活动的最优安排方案。

证明：当$n=k+1$时，再经过上面两个证明可以得到贪心法产生的是该问题的最优解。

9.3 贪心算法的基本要素

必须证明每一步所作的贪心选择最终导致问题的整体最优解。（考察整体最优解，修改最优解，使其以贪心选择开始）

最优子结构性质：A是原问题的一个最优解，则$A' = A - \{1\}$是活动安排问题$E' = \{i \in E, s_i \geq f_1\}$的一个最优解。

和动态规划算法的比较：见？？？

9.4 连续背包问题

贪心法能解决的是连续背包问题，不是放或者不放某个物品的0-1背包问题。

你可以选择以**价值**作为度量标准，选择局部最优解；也可以选择以**空间**作为度量标准，选择局部最优解。但是他们都不能带来整体最优解。很容易想到，应该用价值率作为度量标准来装填背包，使我们获得单位最大收益。

分析:

主要的时间还是花在将物品以价值率作为度量标准排序上，也即$O(n \log n)$。

证明:

设$X = (x_1, x_2, \ldots, x_n)$是贪心解，$Y = (y_1, y_2, \ldots, y_n)$是最优解。$\sum_{i=1}^{n} w_i y_i = C$。$j$是使$x_j \neq 1$的最小下标。

若$X = Y$，则X是最优解。

否则二者至少在一个分量上存在不同。

设k是使得$y_k \neq x_k$的最小下标，则有$y_k < x_k$

- 若$k < j$，则$x_k = 1$。因为$y_k \neq x_k$，所以$y_k < x_k$
- 若$k = j$，由于$\sum_{i=1}^{n} w_i y_i = C$，有$1 \leq i < j = k$或$k = j < i \leq n$则有
  - $k = j < i \leq n \Rightarrow x_j \in [0,1], x_j = x_k \in [0,1] \& x_i = 0$
  - 如果$x_k < y_k$ & $\sum_{i=1}^{n} w_i y_i = C$则
  - $y_k \in [0,1]$ & $x_k < y_k \Rightarrow y_{k+1} = 0$ & $\sum_{i=1}^{n} w_i y_i > C$
  - 如果$y_k = 1$，比上式只会大得更多，以此类推...
- 若$k > j$，则$\sum_{i=1}^{n} w_i y_i > C$，不能成立。

调整Y中的解：将y_k增加到x_k，同时从k后面的式子里减去同样多的量，设调整后的解为$Z = (z_1, z_2, \ldots, z_n)$，有：

$$\sum_{k<i\leq n} w_i (y_i - z_i) = \sum_{k<i\leq n} w_k (z_k - y_k)$$

好难，不想写了。分析Z的利益值，发现$\sum_{1\leq i\leq n} v_i z_i = \sum_{1\leq i\leq n} v_i y_i$

易知：

当$\sum_{1\le i\le n} v_i z_i > \sum_{1\le i\le n} v_i y_i$，$Y$不是最优解，矛盾

当$\sum_{1\le i\le n} v_i z_i = \sum_{1\le i\le n} v_i y_i$，$X$就是最优解。

9.5 单源最短路径 - Dijkstra

问题描述：给定一个有权图，找到从一个给定的源点出发的到达所有其他点的路径。

个人理解：要用树来想这个问题。我们有一个子树T_i，里面有已经选好的$i-1$个顶点，在下次选择中，要考虑所有和这棵树相邻的顶点（边缘顶点），在这些顶点中，选择一个距离这棵树的顶点最近的。怎么确定距离这棵树的顶点的距离呢？对于每个边缘顶点u，求出它到最近的树中顶点v的距离w[u,v]与从起点到树中顶点v的距离之和。选和最小的加入这棵树。之后，需要判断一下余下的每个边缘顶点，如果和刚刚加进来的前边缘顶点相连，而且从前边缘顶点走要比在比较谁是边缘顶点的过程算出的权值更小，则将这个边缘顶点的父母记录改为前边缘顶点，权值更新为更小的权值。

```
ALGO DIJKSTRA(G,s)
---
//Input: 非负权重加权连通图
//Output: 对每个顶点v来说，从s到v的最短路径长度。
Init(PriorityQ)
for each vertex v in V do
    d[v] = inf
    p[v] = null
    Insert(PriorityQ,v,d[v])
d[s] = 0
Decrease(Q,s,d[s])//更新s的优先级
V_tree = empty
for i = 0 to |v|-1 do
    u* = DeleteMin(PriorityQ)
    V_tree = V_tree + u*
    for each vertex u in V-V_tree do
        if d[u*] + w(u*,u) < d[u]
            d[u] = d[u*] + w(u*,u)
            p[u] = u*
            Decrease(Q,u,d[u])
    end
end
```

分析：

只使用邻接表和无序数组，时间复杂度为$O(|V|^2)$。使用二叉堆当作优先队列，可以得到$O(|E|\log|V|)$

9.6 最小生成树 - Kruskall & Prim

这两个算法不陌生了，简要地概括一下：

*Prim算法*和Dijkstra算法有很相似的地方。

开始的时候，任意选一个顶点加到我们的树中，再选不在树里，但离树最近的顶点加到树中。

```
ALGO Prim(G)
//Input: 加权连通图G<V,E>
//Output: 组成G的最小生成树的边的集合。
V_tree += v[0]
E_tree = empty
for i = 1 to |V|-1 do
    在所有边(v,u)中，选出使得v在V_tree，u在V-V_tree的最小权值的边e* = (v*,u*)
    V_tree = V_tree + u*
    E_tree = E_tree + e*
return E_tree

```

分析：主要的时间都浪费在了选最小权值的这一步。如果是无序数组当优先级队列，算法的时间复杂度为时间复杂度为$O(|V|^2)$。最小堆当作优先队列，且使用邻接矩阵：可以得到$O(|E|\log|V|)$。（执行了$|V|-1$次删除最小元素的操作，进行了E次验证。因为在连通图里，$|V|-1\leq E$，这里可以直接写成$(E+|V|-1)O(\log|V|)=O(|E|\log|V|)$）

*Kruskall*算法

选边法，每次从边集中选择一个不与现有树连通且权值最小的边。

```
ALGO Kruskall(G)
//Input: 加权连通图G<V,E>
//Output: 组成G的最小生成树的边的集合。
F := 空集合
for each vertex in V
    do 将v加入森林F
所有的边按权重递增排序
for each edge(u,v) in E do
    if u 和 v 不在一棵子树
        F := F + edge(u,v)
        将u和v所在子树合并
end
return E_tree
```

分析：使用并查集，库鲁斯卡尔可以得到$O(|E|\log|E|)$，每个边在最外层循环一次，每次循环要检查规模小于$|E|$的连通集。

10. 动态规划

10.1 动态规划的基本概念

多阶段决策过程的优化问题、最优化原理。把多阶段过程转化为一系列单阶段问题逐个求解，创立了解决这了过程优化问题的新方法——动态规划。

几个概念：

阶段（Stage）：把所给的问题的求解过程恰当地划分为若干个相互联系的阶段。

状态（State）：每个阶段开始时，问题或系统所处的客观状况。既可以是该阶段的某个起点，又是前一个阶段的某个终点。适用于动态规划求解的问题具有状态的无后效性。

策略：各个阶段决策确定后，就组成了一个决策序列，该序列称之为一个策略。有某个阶段开始到终止阶段的过程称为子过程，对应的某个策略成为子策略。

~~八股文？看不太懂。~~

啥是Bellman最优性原理？求解问题的一个最优策略序列的子策略序列总是最优的，则称该问题满足最优性原理。

动态规划的思想实质：分治思想+解决冗余。

与分治法相比，都将原问题分解为若干个子问题，先求解子问题，从子问题中的解得到原问题的解。但动态规划的子问题不是互相独立的。如果用分治法去解，一些步骤会被重复很多次。

动态规划法用一个表来记录所有已解的子问题的的答案。

动态规划的基本解题思路：

1. 找出最优解的性质，刻画结构特征
2. 递归的定义最优值，写出动态规划方程
3. 以自底向上或自顶向下的方式计算出最优值。
4. 根据计算时最优值得记录信息构造最优解。

两种求解方式：

1. 自顶向下的备忘录法
2. 自底向上

动态规划的有效性依赖于问题本身具有得两个重要的适用性质：：最优子结构和重叠子问题。

1. 最优子结构：如果问题的最优解是由其子问题的最优解来构造，说明该问题具有最优子结构的性质。
2. 重叠子问题：使用递归算法时反复求解相同的子问题。

10.2 计算二项式系数（杨辉三角）

一个组合数为C_n^k，则有$0 \leq k \leq n$。

由杨辉三角联想得到，这个问题有个递推关系式。

$$C_n^k = C_{n-1}^{k-1} + C_{n-1}^k \text{ , } for\, n > k > 0$$

$$C_n^0 = C_n^n = 1$$

```
ALGO Binominal(int n, int k)
for i= 0 to n do
    for j = 0 to min(i,k) do
        if j = 0 OR j = i
            BiCoef[i,j]=1
        else
            BiCoef[i,j] = BiCoef[i-1,j-1]+ BiCoef[i-1,j]
return BiCoef[n,k]
```

分析：（分成两部分：前k行的三角形，和剩下的长方形）

$$A(n,k) = \sum_{i=1}^{k}\sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^{n}\sum_{j=1}^{k} 1 = \sum_{i=1}^{k}(i-1) + \sum_{i=k+1}^{n} k = \frac{k(k-1)}{2} + k(n-k)$$

即为$\Theta(nk)$.

10.3 最长公共子序列

~~PPT这里是疯子么？本来就是难点，你上来就整个4页的证明过程，根本看不下去。~~

问题描述：假设X = BACDB，Y = BDCB，那么LCS = BCB。

我想要一个用动态规划解决这个问题，首先分析一下有没有最优子结构，我发现每一个子序列都是由它长度更小的子序列+1得到的。

参考了labuladong的解释，我这里是这样的理解：对于x和Y来的每个字符来说，要么在LCS中，要么不在LCS中。

从后往前遍历这两个串，如果X[i]==Y[j]那么说明这个字符一定在LCS中，如果不等于那么说明这两个字符至少有一个不在LCS中，需要丢弃一个。

递归解法这样解：

```
def longestCommonSubsequence(str1, str2) -> int:
    def dp(i, j):
        # 空串的 base case
        if i == -1 or j == -1:
            return 0
        if str1[i] == str2[j]:
            # 这边找到一个 lcs 的元素，继续往前找

```

```
            return dp(i - 1, j - 1) + 1
        else:
            # 谁能让 lcs 最长，就听谁的
            return max(dp(i-1, j), dp(i, j-1))

    # i 和 j 初始化为最后一个索引
    return dp(len(str1)-1, len(str2)-1)

```

这样的解法没有生成一个类似于备忘录的dp_table一样的东西，直接返回了LCS的长度，没法进行回溯获取LCS。可以看看这个dp_table是怎样生成的。

用备忘录来做，就是这样：

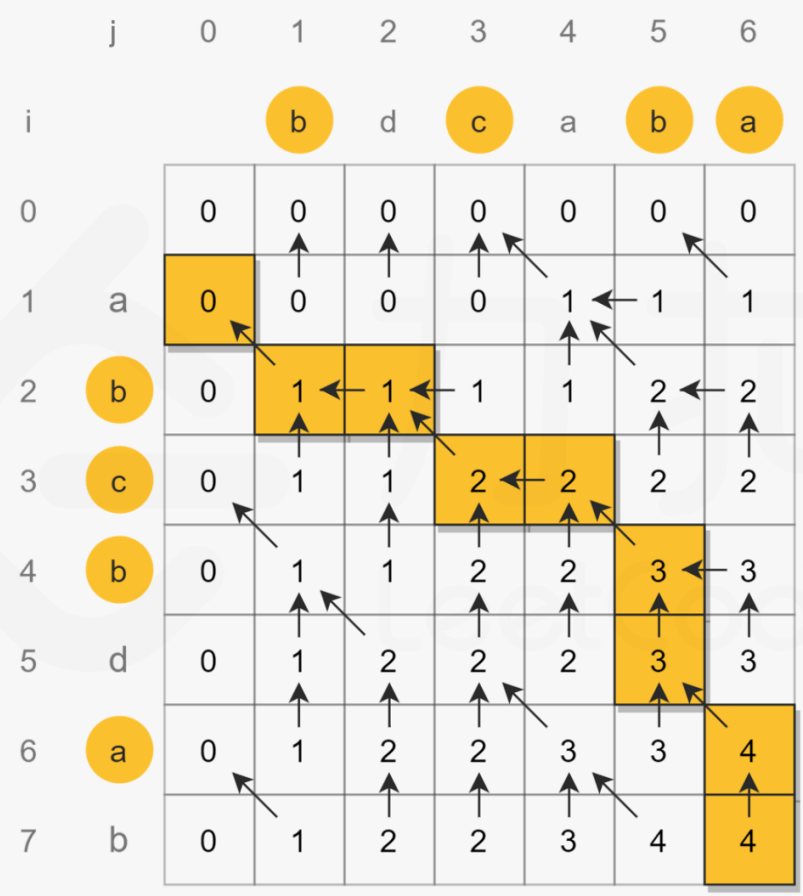
```
def longestCommonSubsequence(str1, str2) -> int:
    m, n = len(str1), len(str2)
    # 构建 DP table 和 base case
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    # 进行状态转移
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if str1[i - 1] == str2[j - 1]:
                # 找到一个 lcs 中的字符
                dp[i][j] = 1 + dp[i-1][j-1]
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    return dp[-1][-1]

```

		0	1	2	3	4	5	6
	str2 / str1	”	b	a	b	c	d	e
0	”	0	0	0	0	0	0	0
1	a	0	0	1	1	1	1	1
2	c	0	0	1	1	2	2	2
3	e	0	0	1	1	2	2	3

这个表的含义是什么？随便选一个元素，就算这个元素对应的两个字符不相等，但是如何知道截止到这个元素，目前的公共子序列状况是怎样？取这个元素上方和左方二者的最大值。因为在右下角，说明进行到这个元素的时候已经包含了*过了上方和左方的状态对应的两个位置*中的一个。这时就不用重复地判断前面那么多问题了。同理，因为向上和向左的情况，已经包含了*沿对角线向左上*这样的情况，在`else`中，不必专门针对沿对角线向左上的情况进行讨论，得到的LCS长度必定是包含在前两种情况之中的。



得到这个表之后，怎样得到LCS？从最右下角的一个元素出发，找对角线方向上减1的元素对应的位置处的字符即可。

时间复杂度：$O(mn)$，空间复杂度也是$O(mn)$。

10.4 矩阵连乘问题

问题描述：给定一系列不同规模的矩阵，找出使得总乘法运算最少的矩阵连乘次序。比如四个矩阵连乘，`(A((BC)D))`和`(A(B(CD)))`、`((AB)(CD))`等等，所做的乘法次数是不同的。每一次`m*n n*p`的两个矩阵进行乘法，需要进行$m*n*p$次乘法运算。

最优解的结构：`(()())`，在第`k`个矩阵后分割，所做的乘法运算总次数为第一个括号最优解加上第二个括号的最优解再加上分割处的矩阵做的乘法次数$s_i s_{k+1} s_{j+1}$，$i \leq j < j$.

目前已经找到了这个问题和子问题的联系以及最优性定理的支持（每一步的两个矩阵组都是最优解）。接下来就是去看看能不能递归地把最优解求出来。

设一个二维矩阵，存储最小值的信息：

m[i,j]存储A[i,j]的最小乘法数目，问题的最终解是A[1,n]。如果i==j则A[i,j]就是A[i]，m[i,j]=0（对角线为零）。如果如果i<j那么可以得到一个递推关系式：m[i,j] = m[i,k]+m[k+1,j]+p[i-1]p[k]p[j]

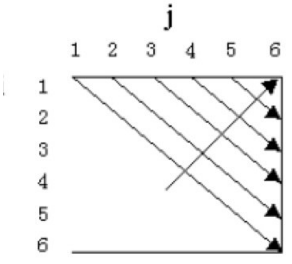
维护两个dp二维数组，m[][]用来记录上述的当前计算乘法总数的最优解，s[][]用来记录得到对应最优值的分割点在哪里。

```
ALGO MaxChain(int p[], int n, int m[][], int s[][])
for i = 1 to n do
    m[i][i] = 0
for r = 2 to n
    for i = 1 to n - r+ 1
        j = i + r - 1
        m[i][j] = m[i+1][j]+p[i-1]*p[i]*p[j]
        s[i][j] = i
            for k = i + 1 to j
                t = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j]
                if t < m[i][j]
                    m[i][j] = t
                    s[i][j] = k
            end
    end
end

```

不难看出时间复杂度是$O(n^3)$，可能看图才能理解：

A1	A2	A3	A4	A5	A6
30×35	35×15	15×5	5×10	10×20	20×25



(a) 计算次序

j

i	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

(b) m[i][j]

j

i	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

(c) s[i][j]

$$m[2][5]=\min\begin{cases} m[2][2]+m[3][5]+p_1p_2p_5=0+2500+35\times15\times20=13000 \\ m[2][3]+m[4][5]+p_1p_3p_5=2625+1000+35\times5\times20=7125 \\ m[2][4]+m[5][5]+p_1p_4p_5=4375+0+35\times10\times20=11375 \end{cases}$$

$$=7125$$

$$\therefore s[2][5]=3$$

10.5 0-1背包问题（动态规划）

在这个语境下，这就是一个整数线性规划的问题，找到一个n维0-1向量就是我们想要的结果。

维护一个二维数组V[i,j]，代表前i个物品满足当前剩余容量容量j的最优价值量（价值最大值）。也就说我们的终极目标就是计算V[n,W]

放还是不放第i个物品?

如果第**i**个项目不能放进背包，那么前**i**个物品的最优子集就是前**i-1**个物品的最优子集。

如果可以放进背包，就有放或者不放两种选择。不放，结果是**V[i-1][j]**；放，结果是**vi + V[i-1][j-w[i]]**。每一次的状态转移就是取这两种做法的最大值。

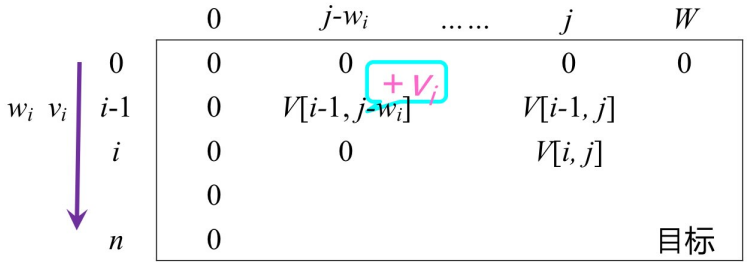
也就有了这样的状态转移方程:

$$V(i,j)=\begin{cases} max\{V(i-1,j),V(i-1,j-w_i)+v_i, & j\geq w_i \\ V(i-1,j), & 0\leq j<w_i \end{cases}$$

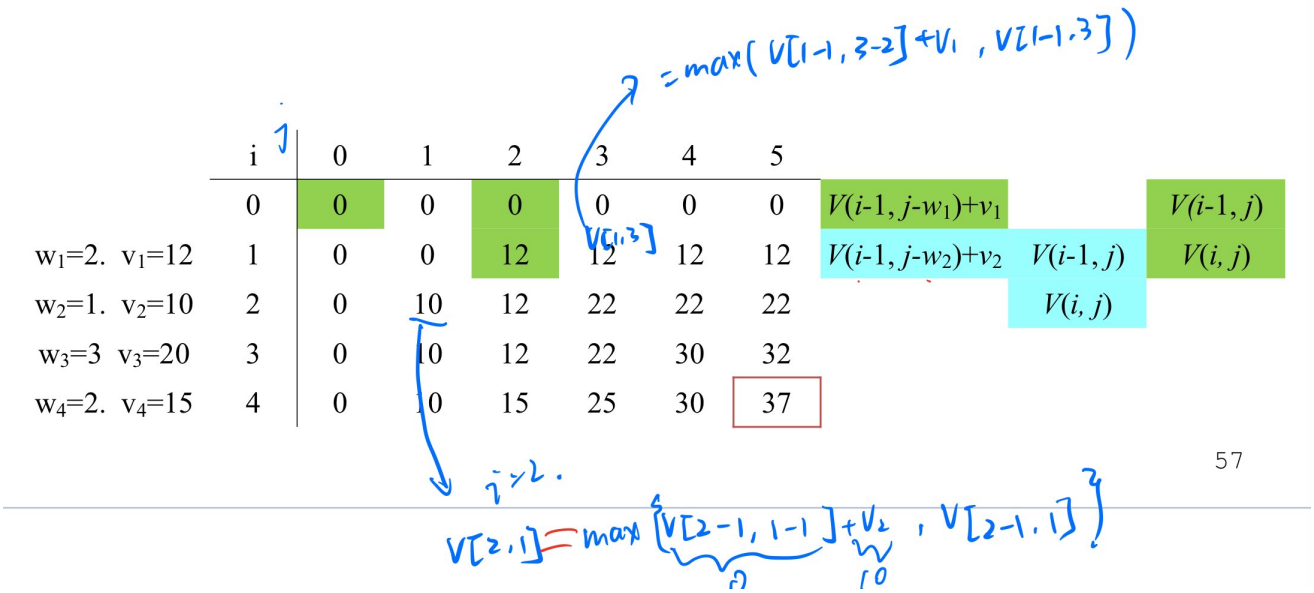
```
ALGO DPKnapsack(v[],w[],n,W)
\\Input:
\\Output:
for w = 0 to W do
    V[0,w] = 0
for i = 1 to n
    for j = 0 to W
        if w[i] <= j
            V[i,j] = max(V[i-1,j],v[i]+V[i-1,j-w[i]])
        else
            V[i,j] = V[i-1,j]
    end
end
return V[n,W]
```

时间复杂度是$O(nW)$。这个动规的数组看起来就是这样的:

Ex. recurrence from the first item



item	weight	value
1	2	12¥
2	1	10¥
3	3	20¥
4	2	15¥



从这个动规矩阵去推导解向量的时候，需要从右下角开始回溯。

如果m[i][j]=m[i-1][j]，说明有没有第i件物品都一样，因此第i件物品没有装进背包，所以x[i]=0，回到m[i-1][j]；
如果m[i][j]=m[i-1][j-w[i]]+v[i]，说明装了第i个商品，该商品是最优解组成的一部分，x[i]=1,随后我们得回到装该商品之前，即回到**m[i-1][j-w(i)]**；
也可以从第n个物品找到第1个物品，先填最后一行：m[n,j] = (j>=w[n])?v[n]:0。再让i从n-1开始遍历。

```
if w[i] <= j
            V[i,j] = max(V[i+1,j],v[i]+V[i+1,j-w[i]])
        else
            V[i,j] = V[i+1,j]
```

10.6 多阶段决策问题

多段图问题，描述：求由s到t的最小成本路径。

什么是w[i][j][k]？第i层，第j个顶点的第k条边的权值。

什么是dp[m][n]？起点到第m层第n个结点的最优值。

从前往后推导时，

递归公式：

$$d[i][j] = min_{j \in V_i \wedge (k,j) \in E}\{d[i-1][k] + w[i-1][k][j]\}$$

$$path[i][j]=arg\{min_{j\in V_i\wedge(k,j)\in E}\{d[i-1][k]+w[i-1][k][j]\}\}$$

初始化的时候：

$$d[0][j]=\begin{cases}w[0][0][j], & <0,j>\in E\\ \infty, & <0,j>\notin E\end{cases}$$

从后往前推导时，

递归公式：

$$d[i][j]=min_{j\in V_i\wedge(k,j)\in E}\{d[i+1][k]+w[i][k][j]\}$$

$$path[i][j]=arg\{min_{j\in V_i\wedge(k,j)\in E}\{d[i+1][k]+w[i][k][j]\}\}$$

初始化的时候：

$$d[n-1][j]=\begin{cases}w[n-1][0][j], & <0,j>\in E\\ \infty, & <0,j>\notin E\end{cases}$$

再想一下，从前往后就是每个顶点的`dp[][]`就是上一层里每个有这个顶点的边的顶点的`dp[][]`加权值里的最大值。从后往前就是把起点变成终点，没有大的不同。

10.7 矩阵的传递闭包 - Warshall算法

问题描述：略。符号描述：$R^{(k)}$是一个矩阵，其中的每个元素$r_{ij}^k=1$表示存在从`i`到`j`的有向路径，其中间结点不超过`k`个。

问题在于如何从$R^{(k-1)}$到$R^{(k)}$

对于$R^{(k)}$来说，$r_{ij}^k=1$说明在上一层有两种情况：第一种是：从第`i`个到第`j`个顶点存在一条长度至少为`1`的路径，且中间结点中不存在大于`k`的顶点；第二种是：中间结点中存在第`k`个结点，此时是第`k`个结点第一次出现在路径中。对于第一种情况，说明上一层早就存在了$r_{ij}^{k-1}=1$，第二种情况，则需要用第`k`个结点进行判断。当且仅当$r_{ik}^{k-1}=1$和$r_{kj}^{k-1}=1$时，才存在一条绕`k`结点的通路。

所以：

$$r_{ij}^k=1\Rightarrow\begin{cases}r_{ij}^{k-1}=1\\ \vee\\ r_{ik}^{k-1}=1\wedge r_{kj}^{k-1}=1\end{cases}$$

易知时间复杂度为$O(n^3)$

```
ALGO Warshall(M)
\\Input:
\\Output:
for k = 0 to n - 1 do
    for i = 0 to n - 1 do
        for j = 0 to n -1 do
            M[i,j] = M[i,j]OR(M[i,k] AND M[k,j])
        end
    end
end
```

10.8 多源最短路径：弗洛伊德算法

问题描述：给你一个图，返回一个距离矩阵一样的东西，可以得知某个点到某个点之间的距离。其实，Floyd算法和Warshall算法的思想都是一样的，就是Warshall算法处理布尔值，仅此而已。这种问题都叫做Floyd-Warshall算法。

$$d_{ij}^{k} = min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}), for\, k \geq 1$$

$$d_{ij}^{k} = w_{ij}, for\, k = 0$$

第0层如果两个顶点之间没有通路，相应的w[i][j]是∞。

10.9 与贪心算法之区别

	动态规划	贪心算法
子问题	每步所做的选择依赖于子问题，解出相关子问题后才能做出选择	仅在当前状态下做出最好选择，不依赖于子问题的解。
求解方式	通常以自底向上的方式解各子问题	通常以自顶向下（随着决定，给定问题的实例越变越小）的方式进行。每一次贪心选择就将所求问题简化为规模更小的子问题

10.10 检查最优性原理的实例

例1：

设G是一个有向加权图，则G从顶点i到顶点j的最短路径问题满足最优性原理。证明?

证明： 设$i - i_p - i_q - j$是一条最短路径，但其中子路径$i_p - i_q - j$不是最优的（反证）。假设$i_p - i'_q - j$为最优路径。重新构造一条路径$i - i_p - i'_q - j$，显然该路径长度小于$i - i_p - i_q - j$，与$i - i_p - i_q - j$是最短路径相矛盾。

例2：

$$\begin{cases} \sum_{i=l}^{n} w_i x_i \le c \\ x_i \in \{0,1\}, l \le i \le n \end{cases} \Rightarrow Knap(l,n,c)$$

证明0-1背包问题`Knap(1,n,c)`满足最优性原理。

证明： 设$(y_2,\dots,y_n)$是`Knap(1,n,c)`的一个最优解，下证$(y_2,\dots,y_n)$是`Knap(2,n,c-w[1]y[1])`的一个最优解。

假设$(y_2,\dots,y_n)$不是`Knap(2,n,c)`的一个最优解，则设$(z_2,\dots,z_n)$是`Knap(2,n,c-w[1]y[1])`的一个最优解。因此有：

$$\sum_{i=2}^{n} v_i z_i > \sum_{i=2}^{n} v_i y_i \ \wedge \sum_{i=2}^{n} w_i z_i \le c - w_i y_i$$

推出

$$v_1 y_1 + \sum_{i=2}^{n} v_i z_i > \sum_{i=1}^{n} v_i z_i \ \wedge \ \sum_{i=2}^{n} w_i z_i \le c$$

说明$(y_1,z_2,\dots,z_n)$是`Knap(1,n,c)`的一个更优解，矛盾。

例3：

最长路径问题不满足最优性定理，证明。

在脑子里画一个正方形，对角顶点的最长路径经过两条边，其中要经过一个相邻顶点路径。相邻顶点路径是对角顶点路径的一个子问题。但是相邻顶点路径的最长路径可以是经过三条边的。所以原问题不满足最优性原理。

10.11 小总结

动规的设计技巧： 阶段的划分、状态的表示、存储表的设计。根据labuladong的总结，也可以这样说：明确 **base case** -> 明确「状态」-> 明确「选择」 -> 定义 **dp** 数组/函数的含义。

```
# 初始化 base case
dp[0][0][...] = base
# 进行状态转移
for 状态1 in 状态1的所有取值：
    for 状态2 in 状态2的所有取值：
        for ...
            dp[状态1][状态2][...] = 求最值(选择1，选择2...)

```

*动规的时间复杂度：*用子问题个数乘以解决一个子问题需要的时间。

动规的两种方向（以 Fibo 为例）：

「自顶向下」注意我们刚才画的递归树（或者说图），是从上向下延伸，都是从一个规模较大的原问题比如说 `f(20)`，向下逐渐分解规模，直到 `f(1)` 和 `f(2)` 这两个 base case，然后逐层返回答案，这就叫「自顶向下」。

「自底向上」反过来，我们直接从最底下，最简单，问题规模最小的 `f(1)` 和 `f(2)` 开始往上推，直到推到我们想要的答案 `f(20)`，这就是动态规划的思路，这也是为什么动态规划一般都脱离了递归，而是由循环迭代完成计算。

*动规的状态转移方程：*这个感觉没啥奇技淫巧，状态转移方程就是一种暴力解法，

谢谢阅读$Th_n^a k\,\S\quad F\emptyset r\ R_e a^{di_n g}$