

算法笔记

<http://www.csie.ntnu.edu.tw/>

关于算法笔记

本站介绍

本站数据是基于分享的精神，一字一句累积起来的。个人希望藉由此站，分享所学所得，支持计算机科学发展。

本站尚处创建阶段，内容不够详尽周全。各位读者若发现错误，欢迎利用留言板告知，俾能改正，十分感谢。

大家可以自行撷取本站的图文作各种用途。想想天地，摸摸良心，不要拿去为非作歹、谋取私利就可以了。

工作项目

过去工作项目

1. 提供算法教学数据。将常见的算法分门别类、编纂目录，归纳一套完整的理论系统。
2. 推广算法解题。收集大量的算法练习题目，穿插于本站的教学数据当中，例如 UVa Online Judge，藉此提升学生的程序设计能力。
3. 宣传算法竞赛。例如 ACM-ICPC、Google Code Jam、TopCoder Open 等等，增加学生的国际交流经验。
4. 出版算法设计的书籍：培养与锻炼……基础入门，推广算法设计的观念。书中内容现在已经汇整至网站上了。
5. 引进国外学校有教，但是国内学校没有教的算法。例如高等数据结构、高等算法、图论算法、组合优化、计算几何、字符串学等等，将重要的算法知识介绍给国内学子。

当前工作项目

1. 介绍计算机科学的应用。收集相关影片，让大众了解计算机科学有什么功用、计算机科学如何改善生活环境。
2. 整理媒体算法。例如文字处理、声音处理、图像处理、计算机绘图、计算机视觉以及各种进阶领域，拓展信息人的视野。

预计工作项目

1. 将网站译为其它语言，让更多人可以获取网站资源。

2. 汇整各种计算机科学领域所对应的软件公司，藉此促进人才媒合，也藉此让在学学生了解产业现况、培养志向。
3. 探讨如何运用计算机科学改善社会问题、增进民众福祉。

资助算法笔记

站长家庭经济状况不佳，日子过得清澹。如果每个月有稳定薪资，就不必烦恼经济问题——然而没有人提供编写算法笔记的工作，站长必须常常分神去从事其它工作。

理想的方式，是由政府或富人，每月固定资助新台币 35000 元做为薪资，让站长可以全心投入算法笔记。愿意资助的朋友，请来信 `algorithm.notes@gmail.com`。

站长的眼睛已经渐渐好转，现在可以长时间使用计算机，不过写程序的话还是太吃力了。因此麻烦大家别再推荐程序开发的工作了！

目录

第一部分	Algorithm	11
第一章	Algorithm	13
1.1	Algorithm	13
1.1.1	算法是什么	13
1.1.2	计算机只会算数字	13
1.1.3	程序用来比对数字、改变数字、储存数字	14
1.1.4	数学和程序这么复杂，为什么要用计算机解决现实问 题?	15
1.2	Algorithm	15
1.2.1	算法是什么?	15
1.2.2	如何记载一个算法?	16
1.2.3	如何实作一个算法?	17
1.2.4	时间复杂度、空间复杂度	18
1.2.5	解决问题的成效	20
1.3	Algorithm	20
1.3.1	学习程序语言	20
1.3.2	学习算法	20
1.3.3	算法设计、算法分析	21
1.4	Algorithm Class	21
1.4.1	Offline Algorithm / Online Algorithm	21
1.4.2	Static Algorithm / Dynamic Algorithm	21
1.4.3	Exact Algorithm / Approximation Algorithm	21
1.5	Time Complexity	22
1.5.1	时间复杂度	22
1.5.2	时间复杂度标记法	22
1.5.3	测试数据	22
1.5.4	算法的步骤数目不是固定的	23

1.5.5	内存	23
1.5.6	当今计算机计算能力的极限	23
1.5.7	最佳排列、最佳组合	23
1.6	P versus NP	24
1.6.1	示意图	24
1.6.2	P 问题	24
1.6.3	NP 问题	24
1.6.4	NP-complete 问题	25
1.6.5	NP-hard 问题	25
1.6.6	$P = NP$?	26
1.6.7	介于 P 与 NP-complete 之间的问题	26
第二章	Algorithm Design	27
2.1	Incremental Method	27
2.1.1	Incremental Method	27
2.1.2	范例：加总数字	27
2.1.3	范例：复制字符串	28
2.1.4	范例：选择排序法 (Selection Sort)	29
2.1.5	范例：印出直角三角形	30
2.1.6	范例：人潮最多的时段 (Interval Partitioning Problem)	30
2.1.7	范例：储存坐标	32
2.1.8	范例：印出转换成小写的字符串	32
2.1.9	范例：对调数字	34
2.1.10	范例：对调数组	34
2.2	Memoization	36
2.2.1	Memoization	36
2.2.2	范例：数组大小	36
2.2.3	范例：加总数字	37
2.2.4	范例：统计字母数量	38
2.2.5	范例：统计数字数量	39
2.2.6	范例：求 1 到 n 的全部整数的立方和，n 的范围由 1 到 10。	40
2.2.7	范例：印出方框	43
2.2.8	范例：拆开循环 (Loop Unrolling)	44
2.3	Enumeration	44
2.3.1	Enumeration	44

2.3.2	范例: 枚举一百个平方数	45
2.3.3	范例: 寻找数组里的最小值	46
2.3.4	范例: 寻找数组里的特定数字	47
2.3.5	范例: 寻找二维数组里的特定数字	48
2.3.6	范例: 平面上距离最近的两个点 (Closest Pair Problem)	50
2.3.7	范例: 字符串匹配 (String Matching)	51
2.3.8	范例: 统计字母数量	53
2.3.9	范例: 反转字符串	54
2.3.10	范例: 寻找总和为 10 的区间	55
2.3.11	范例: 寻找数组之中的最小值, 数组已经由小到大排序	57
2.3.12	范例: 寻找数组之中的特定数字, 数组已经由小到大排序	57
2.3.13	范例: 平面上距离最近的两个点 (Closest Pair Problem)	58
2.3.14	范例: 英文单字从单数变复数	59
2.3.15	范例: 小画家倒墨水 (Flood Fill Algorithm)	60
2.3.16	Straightforward Method / Trial and Error	62
2.3.17	范例: 暴力攻击 (Brute Force Attack)	63
2.3.18	范例: 单向函数 (One-way Function)	63
2.4	Iterative Method	64
2.4.1	Iterative Method	64
2.4.2	范例: 字符串变整数	64
2.4.3	范例: 秦九韶算法 (Horner's Rule)	66
2.4.4	范例: $3n+1$ 猜想 (Collatz Conjecture)	66
2.4.5	范例: 除法	66
2.4.6	范例: 牛顿法 (Newton's Method)	66
2.4.7	范例: 十分逼近法	67
2.4.8	范例: 书塔 (Book Stacking Problem)	67
2.4.9	范例: 生命游戏 (The Game of Life) (Cellular Automata)	68
2.4.10	范例: 兰顿的蚂蚁 (Langton's Ant)	69
2.4.11	范例: 以试除法建立质数表	69
2.4.12	范例: 数学归纳法 (Mathematical Induction)	70
2.5	Recursive Method	70
2.5.1	Recursive Method	70

2.5.2	范例：碎形 (Fractal)	70
2.5.3	范例：质因子分解 (Integer Factorization)	71
2.5.4	范例：L 形磁砖	71
2.5.5	范例：辗转相除法 (Euclid's Algorithm)	72
2.5.6	递推法、递归法，一体两面，同时存在。	73
2.5.7	范例：秦九韶算法 (Horner's Rule)	74
2.5.8	范例：爬楼梯	75
2.5.9	范例：格雷码 (Gray Code)	77
2.6	Divide and Conquer	77
2.6.1	Divide and Conquer	77
2.6.2	范例：分解动作	77
2.6.3	范例：方格法求面积	78
2.6.4	范例：分类数数	78
2.6.5	Recursive Method	79
2.6.6	范例：归并排序法 (Merge Sort)	80
2.6.7	范例：快速排序法 (Quicksort)	80
2.6.8	范例：不重复组合 (Combination)	80
2.6.9	范例：汉诺塔 (Tower of Hanoi)	81
2.6.10	Prune and Search	82
2.6.11	范例：二分搜索法 (Binary Search)	83
2.6.12	范例：寻找数组里第 k 大的数	83
2.6.13	范例：寻找假币 (Counterfeit Coin Problem)	84
2.6.14	Marriage before Conquest	84
2.6.15	Recurrence	86
2.6.16	范例：爬楼梯	86
2.7	Dynamic Programming	86
2.7.1	Recurrence	86
2.7.2	范例：爬楼梯	86
2.7.3	范例：不重复组合 (Combination)	87
2.7.4	范例：河内塔 (Tower of Hanoi)	88
2.7.5	Dynamic Programming = Divide and Conquer + Memoization	88
2.7.6	Dynamic Programming State Space Search	88
2.7.7	用 Dynamic Programming 设计算法时的步骤	88
2.7.8	范例：爬楼梯	89
2.7.8.1	recurrence	89

2.7.8.2	state space	90
2.7.8.3	lookup table	90
2.7.8.4	computational sequence	90
2.7.8.5	initial states / boundary	90
2.7.9	实作	91
2.7.9.1	Top-down	92
2.7.9.2	Bottom-up	94
2.7.10	小结	95
2.7.11	范例：阶乘 (Factorial)	96
2.7.12	范例：帕斯卡尔三角形 (Pascal's Triangle)	96
2.7.13	Staircase Walk	97
2.7.13.1	Staircase Walk	97
2.7.13.2	Recurrence	97
2.7.13.3	复杂度	98
2.7.13.4	程序代码	98
2.7.13.5	节省内存空间	99
2.7.13.6	往其它方向走的话?	101
2.7.13.7	双向都可以递归	101
2.7.13.8	一般公式解	101
2.7.14	Largest Empty Interval	101
2.7.14.1	问题描述 (离散版本)	101
2.7.14.2	Recurrence	102
2.7.14.3	复杂度	102
2.7.14.4	程序代码：求出最长空白的长度	102
2.7.14.5	程序代码：求出最长空白的长度	103
2.7.14.6	程序代码：求出最长空白的位置	105
2.7.14.7	程序代码：求出其中一个最长空白的位置	105
2.7.15	Largest Empty Rectangle	106
2.7.15.1	问题描述 (离散版本)	106
2.7.15.2	如果使用穷举法	106
2.7.15.3	尝试切成条状, Divide and Conquer	107
2.7.15.4	算法	107
2.7.15.5	程序代码	107
2.7.15.6	程序代码	108
2.7.15.7	程序代码	109
2.7.15.8	复杂度	111

2.7.15.9 更好的方法	111
2.7.15.10 讨论	112
2.7.15.11 时间复杂度	112
2.7.15.12 程序代码: Largest Empty Rectangle 的面积	112
2.7.15.13 程序代码: Largest Empty Rectangle 的位置	114
2.7.15.14 最好的方法	114
2.7.16 范例	115
2.7.17 范例: Matrix Chain Multiplication	115
2.7.18 范例: 文章换行	116
2.7.19 范例: Longest Increasing Subsequence	117
2.7.20 范例: 鸡蛋耐力测试	117
2.7.21 范例: 二进制数	117
2.7.22 范例: 节省内存	117
2.7.23 Dynamic Programming 的递归关系是 DAG	117

第一部分

Algorithm

第一章 Algorithm

1.1 Algorithm

1.1.1 算法是什么

算法是资讯工程系非常重要的基础科目。简单来说，算法就是用计算机算数学的学问（古代人用算盘算、现代人用计算机算），可以说是数学科目。

想要解决现实生活当中的各种问题，计算机科学家就把现实问题对应到数学问题，然后设计公式、把公式写成程序，让计算机执行程序计算答案——这些公式就叫做算法了。

尽管这里用了「公式」这个字眼来形容算法，然而并不是各位印象中的数学公式。由于计算机能够执行繁复的计算，所以公式可以设计成好几十行、好几百行，甚至用到很多数学理论。

因此呢，就算学习过算法的人，也不见得懂得设计算法；因为数学、程序的东西实在太复杂了。想把现实问题对应到数学问题，那就更复杂了。

1.1.2 计算机只会算数字

回过头来，计算机又是什么？计算机是个很潮的中文翻译，不过实际上计算机的原意是「计算机」。计算机的英文叫做 computer，而计算的英文就叫做 compute。计算机是一台计算器，只会计算、判断、储存数字。又快又准。程序是一连串计算、判断、储存数字的步骤。计算机只会处理数字（二进制数）。计算机里的每一个文字、每一种颜色、每一种声音，其实都有相对应的数字。打个比方，我们规定：用 1 代表「一」，用 2 代表「乙」，用 3 代表「人」，……。一个数字对应一个中文字。计算机里面的所有中文字，都依循人为规定，变作了数字。再打个比方，「人」这个字，呈现计算机屏幕上是个「人」样。计算机屏幕的画面，是由许多小光点组成的；计算机屏幕上的「人」也是由许多小光点组成的。我们以「人」的左下角为坐标原点，横向为 X 轴，直向为 Y 轴，那么「人」其实是 (0,1)、(1,2)、(2,3)、

... 这些坐标画上黑点后所形成的。「人」这个字的形状，在计算机中变作了一连串的数字。

同样的道理，呈现在计算机屏幕画面上的文字、颜色、图片、影像、声音，全部都可以化作数字。一切事物在计算机里面都是数字。计算机并没有想象中的那么神奇。不过计算机最厉害的地方并不是计算机本身，而是在于计算机可以接上各式各样的设备。接上摄影机与屏幕，就可以把色彩变成数字、把数字变成色彩；接上麦克风与耳机，就可以把声音变成数字、把数字变成声音。

计算机一旦接上了设备，就额外有用处。接上话机和基地台，就可以互通有无；接上数字相机和打印机，就可以制造回忆；接上重量仪和筛子，计算机也会拣土豆；接上车厢、接上警示灯、再杂七杂八接上一堆东西，就变成了大众运输系统。

若要用计算机解决现实问题，通常要考虑两个方面：一、计算机应该接上那些设备？如何用计算机控制这些设备？二、现实问题如何对应到数学问题？如何设计算法？

1.1.3 程序用来比对数字、改变数字、储存数字

举个例子，我们希望把屏幕上的「人」变成斜体字。过程大略是这样——首先呢，把「人」的形状 $(0,1)$ 、 $(1,2)$ 、 $(2,3)$ 、... 这些数字拿出来；然后呢，位置越高的坐标，就往右移动多一点，如此一来就成为斜体字了。想让坐标往右移动，就是让计算机做数字加法计算，然后把相加结果储存起来。

再举个例子，用鼠标点选一个文件夹，文件夹的颜色会反白。过程大略是这样——首先呢，计算机侦测到鼠标点击的坐标之后，把坐标转换成数字；然后呢，再把屏幕画面的数据拿出来，看看屏幕上每个东西的坐标，是哪一个与鼠标的坐标相符合；噢，原来是一个文件夹的图标，把文件夹的显示颜色给反白过来。

再举个例子，计算机据说会拣选土豆。过程大略是这样——把每一颗土豆拿出来，利用特殊的仪器，把形状、重量、色泽、气味统统转换成数字，储存在计算机里面；然后呢，用计算机比较这些数字，找出优良的土豆，如此一来就有绵绵松松的土豆了！编写程序，计算数字，这就是程序设计师的工作。

1.1.4 数学和程序这么复杂，为什么要用计算机解决现实问题？

计算机的计算速度可说是非常的快，一秒钟可以进行好几千万次。就算文字多么的多，图片多么的大，计算机处理起来，也是轻松写意，顺畅无比。

打开计算机里的任何一份文件，用鼠标卷动一下文件画面，眼睛都还没眨一下，正确画面马上就呈现在屏幕上了。事实上在卷动画面的时候，计算机已经经过几千万次的计算，仅使用了极短的时间，就把屏幕上应该呈现的数据全部计算好了。

人类会想要用计算机解决问题，正是仰赖计算机的计算速度、正确性，以及计算机自动按照程序计算的特性。程序设计师只要花心思写出一支好程序，接下来的工作就可以让计算机代劳了。计算机做的比人类更快更好，计算机做得到人类做不到的事情；尽管数学和程序很复杂，还是有很多人选择使用计算机解决问题。

1.2 Algorithm

1.2.1 算法是什么？

算法由三个部分组成：输入、计算步骤、输出。介绍这件事情的时候，有人连结到函数的概念，也有人连结到黑箱白箱的概念。

输出、输入是一堆数字。实务上是将这些数据放在数据结构，例如 array、linked list。

输入数据的来源，通常是硬盘里面储存的档案，或者是藉由硬件装置撷取到的数据，例如数字相机、麦克风等等。输出数据的去处，通常是硬盘里面储存的档案，或者是藉由硬件装置转换之后以其它型态呈现，例如数字电视、数字音响等等。

加法的算法

a — [] —> c

b —

例如

5 — [] —> 8

3 —

计算步骤有两种类型，一类是运算，例如数学运算加减乘除、逻辑运算且或非、比较运算大于等于小于。另一类是读写，例如读取某处的数字、储存数字至某处，就跟计算器的 MR、M+ 按键的意义相似。

古人定义算法，规定计算步骤的数量是必须是有限步，不是无限步。用程序语言的术语来说就是：算法不能有无穷循环。

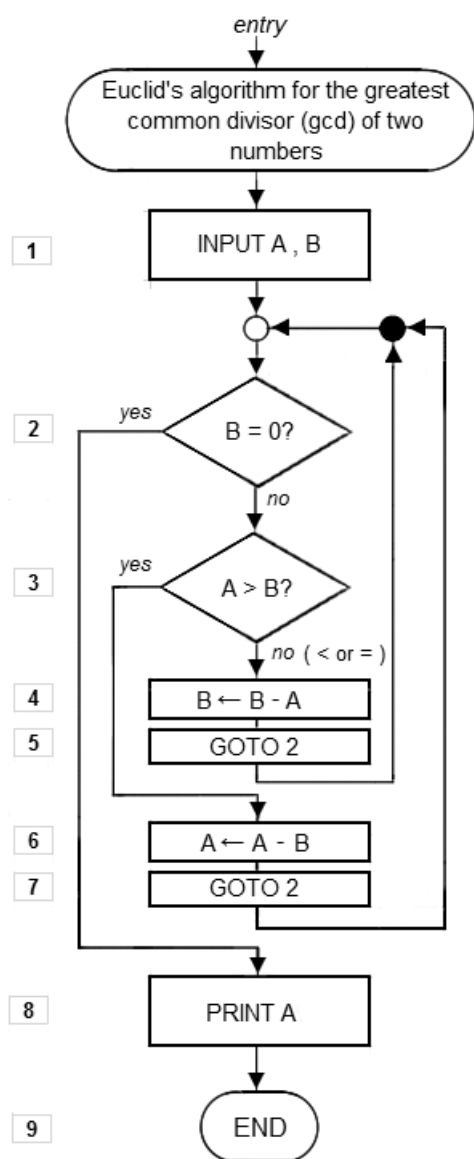
古人当初规定有限步，是为了方便统计总步数。但是实务上，很多计算机程序是开启之后就保持执行状态，直到当机、重开机，例如网络传输的算法。因此实务上可以是无限步。

1.2.2 如何记载一个算法？

有人用伪码来记载一个算法。如要设计计算机程序，伪码是比较恰当的。

```
1: procedure GREATEST_COMMON_DIVISOR( $a, b$ )
2:   while  $a \neq b$  do
3:     if  $a > b$  then
4:        $a \leftarrow a - b$ 
5:     else
6:        $b \leftarrow b - a$ 
7:     end if
8:   end while
9:   return  $a$ 
10: end procedure
```

有人用流程图来记载一个算法。如要设计电子电路，流程图是比较恰当的。



大多数时候，我们无法光从伪码和流程图彻底理解算法，就如同我们无法光从数学公式彻底理解数学概念。想要理解算法，通常还是得藉由文字、图片的辅助说明。

1.2.3 如何实作一个算法？

实作的意思是：实际去操作、实际去运行。对于资工系学生来说，自然就是把算法撰写成计算机程序，例如 C 或者 C++ 程序语言，然后在个人计算机上面执行程序。

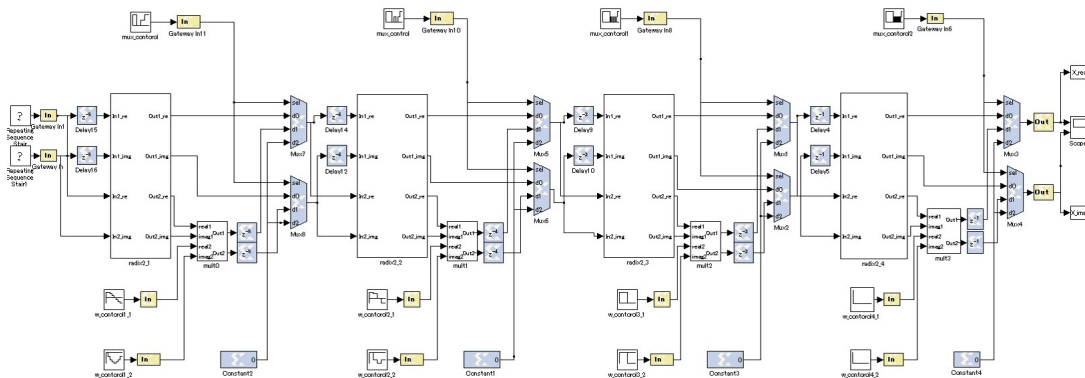
```

1 int gcd(int a, int b) {
2     while (a != b)
3         if (a > b)
4             a -= b;
5         else
6             b -= a;
7     return a;
8 }

```

对于电机系学生来说，自然就是把算法设计成电子电路，在面包板、印刷电路板、PLD 上面执行。

电子电路也有加法器、减法器、AND 逻辑闸、OR 逻辑闸等等，所以也可以用电子电路实作算法。例如电子表、随身听、悠游卡等等，都是直接将算法做死在芯片上面。在个人计算机、智能型手机还没流行之前，以往都是用电子电路实作算法。



电子电路的执行速度是飞快的，计算机程序的执行速度慢了一点。制作电子电路的过程相当麻烦，需要精密的设备、复杂的制程、大量的人力和经费，而且制成之后就无法修改；但是写程序就简单轻松多了。相对地，在计算机上面很容易调整程序代码，又可以储存很多程序代码，最主要的是家家户户都有计算机。

1.2.4 时间复杂度、空间复杂度

要评断一个算法的好坏，最基本的指标是时间和空间。

最直觉的方式，就是测量程序的执行时间、程序的内存使用量。但是由于同一个算法在不同计算机上面的执行时间稍有差异，又由于每个人实作算法所采用的程序语言、程序设计技巧都不一样，所以执行时间、内存使用量不是一个稳定的评断标准。

数学家于是计算步骤数量。

```

1: procedure BUBBLESORT( $A$ )
2:   for  $i \leftarrow 0, \text{length}(A) - 1$  do
3:     for  $j \leftarrow 0, \text{length}(A) - i - 1$  do
4:       if  $A[j] < A[j + 1]$  then
5:         swap  $A[j]$  and  $A[j+1]$ 
6:       end if
7:     end for
8:   end for
9: end procedure

```

```

1: procedure BUBBLESORT( $A, n$ )
2:   for  $i \leftarrow 0, n - 1$  do  $\triangleright n$ 
3:     for  $j \leftarrow 0, n - i - 1$  do  $\triangleright \frac{n(n-1)}{2}$ 
4:       if  $A[j] < A[j + 1]$  then  $\triangleright \frac{n(n-1)}{2}$ 
5:          $\text{temp} \leftarrow A[j]$   $\triangleright \frac{n(n-1)}{2}$ 
6:          $A[j] \leftarrow A[j + 1]$   $\triangleright \frac{n(n-1)}{2}$ 
7:          $A[j + 1] \leftarrow \text{temp}$   $\triangleright \frac{n(n-1)}{2}$ 
8:       end if
9:     end for
10:  end for
11: end procedure

```

$$\begin{aligned}
 \text{total} &= n + \frac{5n(n-1)}{2} \\
 &= n + 2.5n^2 - 2.5n \\
 &= 2.5n^2 - 1.5n \\
 &= O(n^2)
 \end{aligned}$$

数学家把步骤数量写成代数式子。例如当数据有 $n = 1000$ 笔，步骤数量一共是 $2.5 \times 1000^2 - 1.5 \times 1000 = 2498500$ 步。

有了步骤数量之后，还可以进一步粗估执行时间。假设一个步骤需要 10 个 clock，而计算机中央处理器 CPU 的频率是 2GHz：每秒钟执行 2000000 个 clock，那么程序执行时间大约 12.4925 秒。

但是这并不是精准的步伐数量。由于实作的关系，系数很容易变动，所以系数的意义不大。因此数学家只取出代数式子的最高次方，并且规定 n 必须足够大（类似微积分的极限概念）。尽管这是非常不精准的估算方式，不过还是可以对常见的算法进行简易分类，粗略地比较快慢。

	time*	space
bubble sort	$O(n^2)$	$O(n)$
insertion sort	$O(n^2)$	$O(n)$
merge sort	$O(n \log(n))$	$O(n)$
quicksort	$O(n^2)$	$O(n)$
heapsort	$O(n \log(n))$	$O(n)$
counting sort	$O(n + r)$	$O(n + r)$

*worst case

空间的计算方式与时间类似，就不多提了。由于空间复杂度不会超过时间复杂度，而且空间复杂度通常不会呈指数成长，所以大家比较不在意空间复杂度。教科书上面通常比较强调时间复杂度。

1.2.5 解决问题的成效

算法的好坏除了时间和空间的用量以外，主要还是看算法解决问题的成效如何。

数学问题，通常可以明定解答好坏，例如数字越大越好。通常这种情况，有多种算法可以求得正解，那么这些算法的成效是一样好的。

真实世界的问题，通常难以界定绝对的好坏优劣，例如美丑、乐音噪音、喜怒哀乐、是非对错等等，此时算法的成效，就由人自行判断，利用两两比较、投票表决等等方式来决定成效。

1.3 Algorithm

1.3.1 学习程序语言

学习程序语言，有两个层次：一、程序语言本身的语法；二、把想法转换成程序代码。

第一个层次即是「程序语言 Programming Language」。终极目标是背熟规格书、灵活运用程序语言。可参考本站文件「C/C++」。

第二个层次即是「程序设计 Programming」。终极目标是设计程序代码解决问题。可参考本站文件「Programming」。

1.3.2 学习算法

学习算法，有两个层次：一、算法本身的运作过程；二、把想法转换成算法。

第一个层次即是「算法 Algorithm」。终极目标是灵活运用各个算法。可参考本站首页的各大字段，例如图论、计算几何、字符串学等等。

第二个层次即是「算法设计 Algorithm Design」。终极目标是设计数学计算解决问题。可参考本站首页的 Algorithm Design 字段。

1.3.3 算法设计、算法分析

算法又可以细分为两个不同的方向。

算法设计，是制造相对应的算法。算法设计目前已经有一些经典手法，例如 Dynamic Programming、Greedy 等等。读者可以参考《Algorithm Design》这本书。

算法分析，是针对特定算法，精确计量时间复杂度和空间复杂度。算法分析会用到很多数学知识。读者可以参考《An Introduction to the Analysis of Algorithms》这本书。

1.4 Algorithm Class

1.4.1 Offline Algorithm / Online Algorithm

「离线算法」是一口气输入所有数据之后，才能开始运行的算法。例如 Bubble Sort。

「在线算法」是不需等待所有数据到达，就可以分时分段处理输入的算法。例如 Insertion Sort。

有些「在线算法」甚至可以实时提供目前所有输入的正确输出。例如 Insertion Sort。

1.4.2 Static Algorithm / Dynamic Algorithm

「静态算法」是无法随时修改、增加、减少原本的输入数据，无法随时查询输出的算法。例如 Dijkstra's Algorithm。

「动态算法」是可以随时修改、增加、减少原本的输入数据，可以随时查询输出的算法。例如 Binary Search Tree。

1.4.3 Exact Algorithm / Approximation Algorithm

「精确算法」是计算结果绝对正确的算法。

「近似算法」是计算结果拥有误差的算法。

有许多问题无法快速计算正确答案。为了追求速度，就会设计「近似算法」。

1.5 Time Complexity

1.5.1 时间复杂度

想要描述一个算法执行速度有多快，最直观的方式是测量算法计算时间，另一种方式是统计算法步骤数目。由于执行时间深受机械规格与实作方式影响，难以放诸四海皆准，因此学术上倾向于统计算法步骤数目。一般都是统计加减乘除的次数。

1.5.2 时间复杂度标记法

时间复杂度的标记法，是几十年前的数学家发明的一种方式：大写的英文字母 O 函数代表算法执行步骤数目上限，大写的希腊字母 Ω 函数代表下限，大写的希腊字母 Θ 函数代表同时满足上限与下限（也就是不多不少刚刚好）。这些都是假设 N 无限大的情况，又由于 N 无限大，所以我们只需比较函数的最高次方项，另外我们省略了最高次方项的系数。

N 无限大。无限大对数学家来说是司空见惯，然而对程序设计师来说却是天方夜谭。什么时候程序设计师才会遇到无限大的测试数据呢？遇不到。真实世界中根本不可能把无限大的测试数据输入到程序之中。不管是什么坚忍不拔、屹立不摇的程序，总还是有那么一天，发生了停电、当机、人为更新设备，而把程序中止了，造成程序没时间吃进无限多的测试数据。真实世界也没有那么大的内存能够一口气读进无限多笔数据。 N 无限大是不可能的，但是有可以模拟为 N 无限大的情况。例如操作系统的程序，例如网络应用程序，持续执行个一年半载都不停，含辛茹苦不眠不休地处理数据。一有测试数据就赶快解决掉，当作好像没有发生过一样，乍看是无限多笔测试数据，实际上却是同一个步骤执行无限多次。这时候用时间复杂度的标记法，用来判断算法快慢，是一个不错的指标。然而还是要小心，当两个算法时间复杂度一样，两者的速度也可能相去甚远，因为最高次方项的系数根本就被忽略了。一般在单机上跑没几秒钟就会结束的程序，只喂那少少的测试数据，要拿时间复杂度来评定算法快慢，那就有点扯了。 $N = 5$ 的情况下，说不定 $O(N^3)$ 的算法表现的比 $O(N^2)$ 好。设计一个 $O(N)$ 的算法，在 $N = 5$ 的情况下反而跑的比 $O(N^2)$ 的算法还慢。两个同为 $O(N)$ 的算法可能不一样快， N 大时甲快、 N 小则乙快。

1.5.3 测试数据

当测试数据很乱，那我们可以说平均的时间复杂度多少；当测试资料很整齐，那我们可以说最佳与最坏的时间复杂度为多少。例如 Quicksort，

最佳 $O(N)$ ，平均 $O(N \log N)$ ，最差 $O(N^2)$ 。另外还想讲一件事：最佳、平均、最坏跟 ω 、 θ 、 O 没有关系，不知道为什么很多人觉得它们是对应的。

Smoothed Analysis 则是分析测试数据有多少机率是整齐的、多少机率是乱的。

1.5.4 算法的步骤数目不是固定的

Probabilistic Analysis 和 Amortized Analysis 和 Competitive Analysis。

1.5.5 内存

再者，时间复杂度的标记法，完全忽略了处理 I/O 和内存管理的问题。要是数据结构复杂一点、庞大一点，读取数据就会变得比较慢，就算是时间复杂度比较低的算法，也可能慢得吓死人。时间复杂度的标记法也没有考虑程序语言特性和平台特性。平平同一个算法，用 C 写的通常就比用 Java 的跑得快。

时间复杂度标记法再怎么不可靠，也比不上实作的不可靠。平平同一个算法，不同人写出来的程序代码也可能执行效率不一样，差十倍都是有可能的。

1.5.6 当今计算机计算能力的极限

也许各位已经听闻过当今七大数学难题之一「 $P=NP$ 问题」。目前的计算机运算能力其实差强人意，绝大多数的问题都没办法快速地求解。就算找来大量计算机实施并行计算，依然没办法快速地求解。

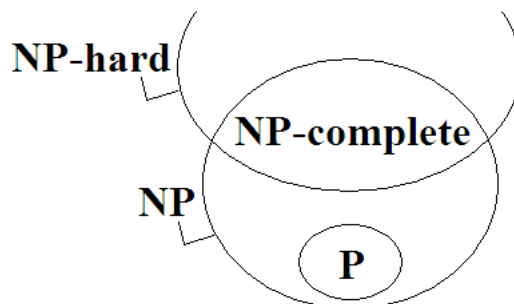
然而，现代人类对于计算机有着神祇般的依赖，各种日常生活问题都祈望计算机能够帮上忙。于是近似算法出现了，用来求得一个马马虎虎差不多的答案。

1.5.7 最佳排列、最佳组合

「穷举所有排列」目前不存在多项式时间的算法。「穷举所有组合」目前有伪多项式时间的算法。

1.6 P versus NP

1.6.1 示意图



1.6.2 P 问题

用多项式时间算法能够计算答案的问题。

「找出一群数字当中最大的数字」是 P 问题。

P 的全名是 Polynomial time。通常以「P」表示所有 P 问题构成的集合。

1.6.3 NP 问题

用指数时间算法能够计算答案的问题，同时，用多项式时间算法能够验证答案的问题。

由于 P 问题也可以改用指数时间算法计算答案、当然可以用多项式时间验证答案，故 P 问题都属于 NP 问题。

「找出一张图的一条 Hamilton Path」是 NP 问题。

计算答案：

穷举所有可能的路线，一条一条验证。

是指数时间算法。

验证答案：

给定一条可能的路线，就照着路线走，看看能不能走到每一点。

是多项式时间算法。

「找出一张图成本最小的那条 Hamilton Path」不是 NP 问题。

计算答案：

穷举所有可能的路线，一条一条验证。

是指数时间算法。

验证答案：

就算给定一条可能的路线，

还是必须穷举所有路线，一条一条验证，才知道哪条路线成本最少。

是指数时间算法。

值得一提的是，每一个 NP 问题，都可以设计出多项式时间算法，转换成另一个 NP 问题。换句话说，所有 NP 问题都可以用多项式时间算法彼此转换。

NP 的全名是 Non-deterministic Polynomial time，定义颇复杂，此处省略之。通常以「NP」表示所有 NP 问题构成的集合。

1.6.4 NP-complete 问题

所有 NP 问题当中，最具代表性、层次最高、最难的问题。

NP-complete 问题的各种特例，涵盖了所有 NP 问题。只要有办法解决 NP-complete 问题，就有办法解决 NP 问题。

各个 NP-complete 问题都等价、都一样难，可以用多项式时间算法彼此转换。现今已经找出上百个 NP-complete 问题了。

Complete 的意义为：能够代表整个集合的子集合。举例来说，它就像是一个线性空间（linear space）的基底（basis）。

「判断一张图是否存在 Hamilton Path」已被证明是 NP-complete 问题。

1.6.5 NP-hard 问题

用多项式时间算法转换 NP 问题所得到的问题，同时，必须是跟 NP-complete 问题一样难、还要难的问题。

NP-hard 问题可能是：甲、NP-complete 问题（是 NP 问题），乙、超出 NP 问题的复杂度，是更难的问题。

「找出一张图成本最小的 Hamilton Path」是 NP-hard 问题。

由「找出一张图的一条 Hamilton Path」这个 NP 问题，
用多项式时间把每条边加上成本而得。

而且「找出一张图成本最小的 Hamilton Path」至少比 NP-complete 问题还难。

1.6.6 $P = NP$?

这是信息科学界的一个悬案。大意是说：到底 NP 问题能不能用多项式时间算法解决呢？如果可以的话，那么 NP 问题就都变成了 P 问题了。这意味着有一些花上几十年几百年算不出答案的问题，变得可以在几分几秒内计算完毕、得到答案。

有一个解决这个悬案的方向是：尝试发明一个多项式时间算法，解决某一个 NP-complete 问题。一旦找到了一个多项式时间算法能够算出某一个 NP-complete 问题的答案，我们可以将此 NP-complete 问题进行特例化得到所有 NP 问题，如此一来，所有 NP 问题就一定可以用多项式时间算法算出答案了。

很多人声称自己已经成功证明了，但是至今还没有一个让所有人都信服证明：

<http://www.win.tue.nl/~gwoegi/P-versus-NP.htm>

1.6.7 介于 P 与 NP-complete 之间的问题

<http://cstheory.stackexchange.com/questions/79/>

第二章 Algorithm Design

2.1 Incremental Method

不积跬步，无以至千里。不积小流，无以成江海。《荀子》

2.1.1 Incremental Method

「递增法」是符合计算机运作特性的方法。计算机执行程序，一次只做一个动作，完成了一件事才做下一件事。当一个问题太大太多时，化整为零、一个一个解决吧！

合抱之木，生于毫末；九层之台，起于累土；千里之行，始于足下。谨以此句与大家共勉。

2.1.2 范例：加总数字

无论计算机再怎么强，还是得一个一个累加数字。

$$3 + 6 + 9 + (-8) + 1 = 11$$

1. $0 + 3 = 3$
2. $3 + 6 = 9$
3. $9 + 9 = 18$
4. $18 + (-8) = 10$
5. $10 + 1 = 11$

```
1 void summation()
2 {
3     int array[5] = {3, 6, 9, -8, 1};
4     int sum = 0;
5     for (int i=0; i<5; i++)
6         sum += array[i];
7     cout << "总和是" << sum;
8 }
```

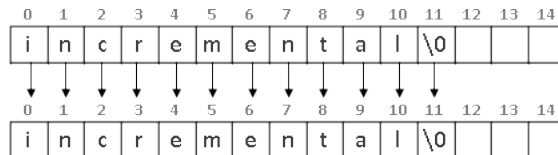
```

1 int summation(int array[], int n)
2 {
3     int sum = 0;
4     for (int i=0; i<n; i++)
5         sum += array[i];
6     return sum;
7 }

```

2.1.3 范例：复制字符串

无论计算机再怎么强，还是得逐字复制。



```

1 void copy()
2 {
3     char s[15] = "incremental";
4     char t[15];
5
6     int i;
7     for (i=0; s[i] != '\0'; ++i)
8         t[i] = s[i];
9     t[i] = '\0';
10
11     cout << " 原本字符串 " << s;
12     cout << " 复制之后的字符串 " << t;
13 }

```

```

1 void copy(char* s, char* t)
2 {
3     int i;
4     for (i=0; s[i]; ++i)
5         t[i] = s[i];

```

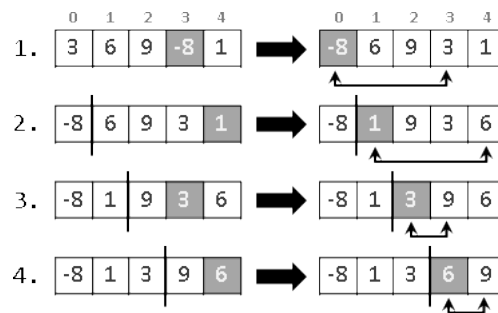
```

6     t[i] = '\0';
7 }

```

2.1.4 范例：选择排序法 (Selection Sort)

找到第一小的数字，放在第一个位置；再找到第二小的数字，放在第二个位置。一次找一个数字，如此下去就会把所有数值按照顺序排好了。



```

1 void selection_sort() {
2     int array[5] = {3, 6, 9, -8, 1};
3
4     for (int i=0; i<5; ++i)
5     {
6         // 从尚未排序的数字当中，找到第 i 小的数值。
7         int min_index = i;
8         for (int j=i+1; j<5; ++j)
9             if (array[j] < array[min_index])
10                min_index = j;
11
12        // 把第 i 小的数值，放在第 i 个位置。
13        swap(array[i], array[min_index]);
14    }
15
16    // 印出排序结果。
17    for (int i=0; i<5; ++i)
18        cout << array[i];
19 }

```

2.1.5 范例：印出直角三角形

多字成行，多行成直角三角形。由细微的东西开始，一件一件组起来。



```

1 // 多字成行
2 void print_line(int n) // n 是一行的长度
3 {
4     for (int i=1; i<=n; i++) cout << '@';
5     cout << '\n';
6 }
7
8 // 多行成直角三角形
9 void print_triangle(int n) // n 是行数
10 {
11     for (int i=n; i>=1; i--) print_line(i);
12 }

```

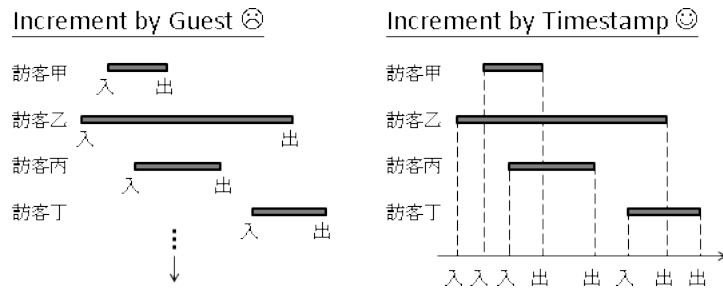
UVa 488 10038 10107
10370

2.1.6 范例：人潮最多的时段 (Interval Partitioning Problem)

一群访客参加宴会，我们询问到每一位访客的进场时刻与出场时刻，请问宴会现场挤进最多人的时段。

换个角度想，想象会场门口装着一支监视器。有访客进入，会场就多一人；有访客离开，会场就少一人。如此就很容易统计会场人数。递增的标的是时刻，而不是访客。

【注：这个技巧在中文网络上昵称为「离散化」。】



```

1 struct Guest {int arrival, leave;} g[10];
2
3 bool cmp(const int& i, const int& j)
4 {
5     return abs(i) < abs(j);
6 }
7
8 void maximum_guest()
9 {
10     vector<int> time;
11     for (int i=0; i<10; ++i)
12     {
13         time.push_back(+g[i].arrival);
14         time.push_back(-g[i].leave);
15     }
16
17     sort(time.begin(), time.end(), cmp);
18
19     int n = 0, maximum = 0;
20     for (int i=0; i<time.size(); ++i)
21     {
22         if (time[i] >= 0)
23             n++;
24         else
25             n--;
26
27         maximum = max(maximum, n);
28     }
29     cout << "人潮最多的时段有 " << maximum << "人";

```

30 }

此处仅找出人数。找出人潮最多的时段，就留给各位自行尝试吧。

UVa 688 972 10613
10585 10963
UVa 308 837

2.1.7 范例：储存坐标

递增的标的，主为点，次为坐标轴。

```
1 struct Point {float x, y;} p[5] =
2 {
3     {0, 1}, {1, 2}, {3, 0}, {2, 2}, {3, 1}
4 };
```

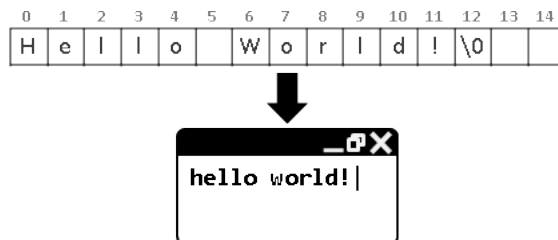
递增的标的，主为坐标轴，次为点。

	0	1	2	3	4
x	0	1	3	2	3

	0	1	2	3	4
y	1	2	0	2	1

```
1 float x[5] = {0, 1, 3, 2, 3};
2 float y[5] = {1, 2, 0, 2, 1};
```

2.1.8 范例：印出转换成小写的字符串



有需要改变的，只有大写字母——如果是大写字母，就转换成小写字母并且印出；如果不是大写字母，就直接印出。

```
1 void print_lowercase()
2 {
3     char s[15] = "Hello World!";
4
5     char t[15];
6
7     // 第一波：复制字符串
8     for (int i=0; s[i]; i++)
```



```
9         t[i] = s[i];
10
11     // 第二波：换成小写
12     for (int i=0; s[i]; i++)
13         if (t[i] >= 'A' && t[i] <= 'Z')
14             t[i] = t[i] - 'A' + 'a';
15
16     // 第三波：印出字符串
17     cout << t;
18 }
```

```
1 void print_lowercase()
2 {
3     char s[15] = "Hello World!";
4
5     char t[15];
6
7     // 每一波的程序代码可以自行包装成为函数，
8     // 亦可套用内建函数库。
9     my_copy(s, t);        // 复制字符串
10    my_lowercase(t);       // 换成小写
11    cout << t;            // 印出字符串
12 }
```

第一种解法称作 one-pass，数据只会读取一遍。读取数据的同时，也一口气处理掉所有事情。

第二种解法称作 multi-pass，数据会重复读取许多遍。所有事情划分成数个阶段，逐步处理，每个阶段只专心处理一件事情。

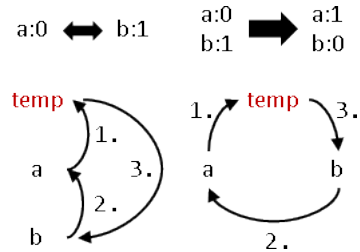
one-pass 的优点是：程序代码简短、执行时间也短。缺点是：程序代码不易编修。

multi-pass 的优点是：程序代码一目了然，容易编修、测试、调试；程序代码可以包装成为函数，也有机会套用内建函数库。缺点是：需要额外的暂存内存。

这两种方式各有利弊。程序员必须自行取舍。

2.1.9 范例：对调数字

利用一个变量，暂存其中一个数字，以便对调。



```

1 void swap_int()
2 {
3     int a = 0, b = 1;
4
5     // 交换 a 与 b
6     int temp = a;
7     a = b;
8     b = temp;
9
10    cout << a << ' ' << b;
11 }

```

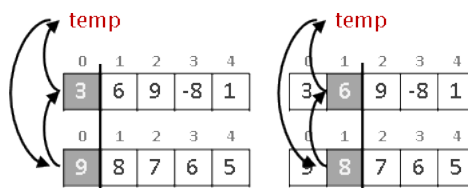
```

1 void swap_int(int& a, int& b)
2 {
3     int temp = a;
4     a = b;
5     b = temp;
6 }

```

2.1.10 范例：对调数组

节省内存的方法：采用递增法，逐一对调数字。

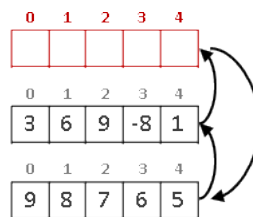


```

1 void swap_int_array()
2 {
3     int a[5] = {3, 6, 9, -8, 1};
4     int b[5] = {9, 8, 7, 6, 5};
5
6     // one-pass
7     for (int i=0; i<5; ++i)
8     {
9         int temp = a[i];
10        a[i] = b[i];
11        b[i] = temp;
12    }
13 }

```

浪费内存的方法：建立一个数组，暂存其中一个数组。



```

1 void swap_int_array()
2 {
3     int a[5] = {3, 6, 9, -8, 1};
4     int b[5] = {9, 8, 7, 6, 5};
5
6     // multi-pass
7     int temp[5];
8     for (int i=0; i<5; ++i) temp[i] = a[i];
9     for (int i=0; i<5; ++i) a[i] = b[i];
10    for (int i=0; i<5; ++i) b[i] = temp[i];
11 }

```

```

1 void swap_int_array()
2 {
3     int a[5] = {3, 6, 9, -8, 1};

```

```
4      int b[5] = {9, 8, 7, 6, 5};
5
6      // multi-pass
7      int temp[5];
8      my_copy(a, temp);
9      my_copy(b, a);
10     my_copy(temp, b);
11 }
```

2.2 Memoization

惟事事，乃其有备，有备无患。《书经》

2.2.1 Memoization

「记忆法」是符合计算机运作特性的方法。计算机拥有大量储存空间。只要将计算过的数值，储存于内存，往后就能直接使用内存储存的数据，不必再浪费时间重复计算一遍。

Memoization (Tabulation)

算法执行过程之中，实时更新数值，储存于内存。

例如堆栈的大小。

Preprocessing (Precalculation)

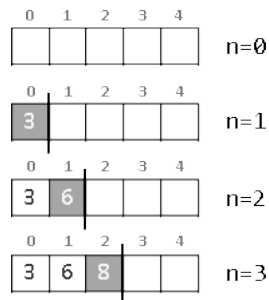
算法开始之时，预先计算数值，储存于内存。

例如圆周率、字符串的长度、质数的表格。

如果要储存大量的、同性质的数值，我们可以将这些数值整理成一个表格（通常是数组），以方便查阅——称作「查询表 lookup table」。例如质数表便是一个「查询表」。

2.2.2 范例：数组大小

使用一个变量，纪录数据数量，以便迅速地增加数据。



```

1 void array_size()
2 {
3     int array[100];
4     int n = 0;          // 使用一个变量，纪录数据数量。
5     array[n++] = 3;    // 以便迅速地增加数据。
6     array[n++] = 6;
7     array[n++] = 9;
8     cout << n;
9 }

```

C++ 程序语言的标准函数库的 `stack`，事实上也额外隐含了一个变量，纪录数据数量。当堆栈塞入数据、弹出数据的时候，也就是调用 `push` 函数、调用 `pop` 函数的时候，就默默更新数据数量。

```

1 void stack_size()
2 {
3     stack<int> s;        // C++ STL <stack>
4     s.push(1);          // 默默地 n++
5     s.pop();            // 默默地 --n
6     cout << s.size();   // 把 n 印出来
7 }

```

2.2.3 范例：加总数字

利用一个变量，累计数字的总和。

$$3 + 6 + 9 + (-8) + 1 = 11$$

- . 0
- 1. 3 (+3)
- 2. 9 (+6)
- 3. 18 (+9)
- 4. 10 (-8)
- 5. 11 (+1)

```

1 void summation()
2 {
3     int array[5] = {3, 6, 9, -8, 1};
4     int sum = 0;
5     for (int i=0; i<5; i++)
6         sum += array[i];
7     cout << "总和是" << sum;
8 }

```

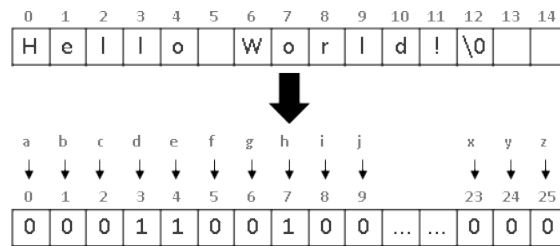
```

1 int summation(int array[], int n)
2 {
3     int sum = 0;
4     for (int i=0; i<n; i++)
5         sum += array[i];
6     return sum;
7 }

```

2.2.4 范例：统计字母数量

建立 26 格的数组，让字母 a 到 z 依序对应数组的每一格，作为 lookup table。一边读取字符串，一边累计字母出现次数。



```

1 void count_letter()
2 {
3     char s[15] = "Hello World!";
4     int c[26] = {0};
5
6     // 字母一律换成小写
7     for (int i=0; s[i]; i++)
8         if (s[i] >= 'A' && s[i] <= 'Z')

```

```

9           s[i] = s[i] - 'A' + 'a';
10
11        // 统计字母数量
12        for (int i=0; s[i]; i++)
13            if (s[i] >= 'a' && s[i] <= 'z')
14                c[s[i] - 'a']++;
15
16        // 印出统计结果
17        for (int i=0; i<26; i++)
18            cout << char('a'+i) << ':' << c[i] << '\n';
19    }

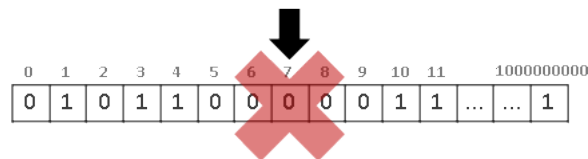
```

UVa 10260 10082
10222 12626

2.2.5 范例：统计数字数量

当数字范围太大，无法建立那么大的数组，可以改用 hash table、binary search tree 等等数据结构作为 lookup table。

1, 3, 4, 10, 11, 1000000000, 23, 99, 123, 514,



```

1 void count_number()
2 {
3     int array[10] =
4     {
5         1, 3, 4, 10, 11,
6         1000000000, 23, 99, 123, 514
7     };
8     // int c[1000000000] = {0};
9     map<int, int> c;    // binary search tree
10
11    // 统计数字数量
12    for (int i=0; i<10; i++)
13        c[array[i]]++;
14

```

```

15      // 印出统计结果
16      for (auto i=c.begin(); i!=c.end(); ++i)
17          cout << i->first << ':' << i->second << '\n';
18  }

```

UVa 11572 141

2.2.6 范例：求 1 到 n 的全部整数的立方和，n 的范围由 1 到 10。

$$\begin{array}{rcl}
 1^3 & = & 1 \\
 1^3 + 2^3 & = & 9 \\
 \vdots & & \vdots \\
 1^3 + 2^3 + 3^3 + \dots + 10^3 & = & 3025
 \end{array}$$

以直接的方式，累加每个立方数。（尽管这个问题有公式解，但是为了方便举例，所以这里不采用公式解。）

```

1  int sum_of_cubes(int n)
2  {
3      int sum = 0;
4      for (int i=1; i<=n; i++)
5          sum += i * i * i;
6      return sum;
7  }
8
9  void print_sum_of_cubes()
10 {
11     int n;
12     while (cin >> n && n > 0)
13         cout << sum_of_cubes(n);
14 }

```

使用 Memoization。建立 11 格的数组，每一格依序对应 0 到 10 的立方数，作为 lookup table。一旦计算完毕，就储存至表格；往后就直接读取表格，不需重复计算。

```

1  int sum_of_cubes(int n)
2  {
3      // 其值为 0 表示没有存入答案
4      static int answer[10 + 1] = {};

```



```
5
6    // 如果已经计算过，就直接读取表格的答案。
7    if (answer[n] != 0) return answer[n];
8
9    // 如果不曾计算过，就计算一遍，储存答案。
10   int sum = 0;
11   for (int i=1; i<=n; i++)
12       sum += i * i * i;
13   return answer[n] = sum;
14 }
15
16 void print_sum_of_cubes()
17 {
18     int n;
19     while (cin >> n && n > 0)
20         cout << sum_of_cubes(n);
21 }
```

使用 Preprocessing 。

```
1 void print_sum_of_cubes()
2 {
3     // 预先建立立方数表格
4     int cube[10 + 1];
5     for (int i=1; i<=10; ++i)
6         cube[i] = i * i * i;
7
8     int n;
9     while (cin >> n && n > 0)
10    {
11        // 直接读取表格的立方数
12        int sum = 0;
13        for (int i=1; i<=n; ++i)
14            sum += cube[i];
15        cout << sum;
16    }
17 }
```

Preprocessing 当然也可以直接算答案啦。

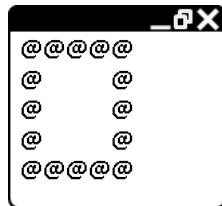
```
1  int sum_of_cubes(int n)
2  {
3      int sum = 0;
4      for (int i=1; i<=n; i++)
5          sum += i * i * i;
6      return sum;
7  }
8
9  void print_sum_of_cubes()
10 {
11     // 预先计算所有答案
12     int answer[10 + 1];
13     for (int i=1; i<=10; ++i)
14         answer[i] = sum_of_cubes(i);
15
16     // 直接读取表格的答案
17     int n;
18     while (cin >> n && n > 0)
19         cout << answer[n];
20 }
```

最后是 Preprocessing 的极致。

```
1  void print_sum_of_cubes()
2  {
3      // 预先计算答案，写死在程序代码里面。
4      int answer[10 + 1] =
5      {
6          0, 1, 9, 36, 100, 225,
7          441, 784, 1296, 2025, 3025
8      };
9
10     // 直接读取表格的答案
11     int n;
12     while (cin >> n && n > 0)
13         cout << answer[n];
14 }
```

2.2.7 范例：印出方框

建立二维数组：数组的格子，依序对应窗口的文字。



	0	1	2	3	4
0	@	@	@	@	@
1	@				@
2	@				@
3	@				@
4	@	@	@	@	@

不直接印出方框，而是间接填至数组。不必数空格键，只需两条水平线和两条垂直线。

```

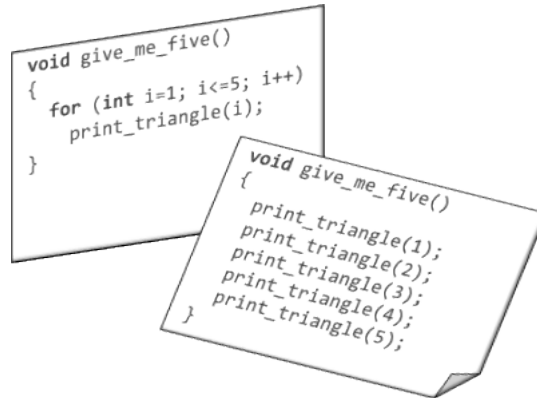
1 void print_square_border()
2 {
3     // 建立内存
4     char array[5][5];
5
6     // 预先填入空格键
7     for (int i=0; i<5; ++i)
8         for (int j=0; j<5; ++j)
9             array[i][j] = ' ';
10
11     // 填入方框：两条水平线、两条垂直线
12     // 即便相互重叠也无所谓
13     for (int i=0; i<5; ++i) array[0][i] = '@';
14     for (int i=0; i<5; ++i) array[4][i] = '@';
15     for (int i=0; i<5; ++i) array[i][0] = '@';
16     for (int i=0; i<5; ++i) array[i][4] = '@';
17
18     // 印出方框
19     for (int i=0; i<5; ++i)
20     {
21         for (int j=0; j<5; ++j)
22             cout << array[i][j];
23         cout << '\n';
24     }

```

25 }

UVa 105 706

2.2.8 范例：拆开循环（Loop Unrolling）



循环语法的功能是：一段指令，重复实施数次，但是每次都稍微变动一点点。

事实上，我们可以反璞归真，拆开循环，还原成数行指令。如此一来，就节省了循环每次累加变量的时间，也节省了循环每次判断结束条件的时间。

拆开循环是一种 Preprocessing，预先计算循环变量、预先计算循环结束条件。

拆开循环之后，虽然提高了程序的执行速度，但是降低了程序可读性。程序员必须自行取舍。

2.3 Enumeration

愚者千虑，必有一得。《史记》

2.3.1 Enumeration

「枚举法」利用了计算机无与伦比的计算速度。找到不确定的变量，枚举所有可能性，逐一判断正确性。

Enumerate

一笔一笔列出所有数据。

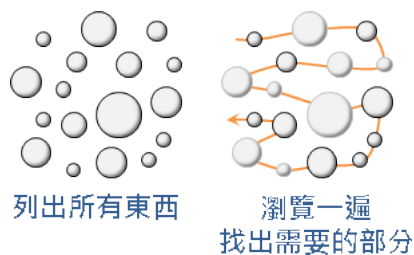
对应到程序语言的 for。

Search

浏览所有数据，找出需要的部份。

对应到程序语言的 for 加 if。

收集充分信息，就能解决问题。

**2.3.2 范例：枚举一百个平方数**

采用直接法：依序枚举数字 1 到 100；枚举过程当中，将数字平方得到平方数。

1	2	3	4	5	100
↓	↓	↓	↓	↓		↓
1	4	9	16	25	10000

```

1 void generate_squares()
2 {
3     for (int i=1; i<=100; i++)
4         cout << i*i << "是平方数";
5 }

```

采用试误法：依序枚举数字 1 到 ∞ ；枚举过程当中，判断数字是不是平方数。

1 ~~2~~ ~~3~~ 4 ~~5~~ ~~6~~ ~~7~~ ~~8~~ 9 ~~10~~ ~~11~~ ~~12~~ ~~13~~

```

1 void generate_squares()
2 {
3     for (int i=1; i<=100*100; i++)
4     {

```

```

5         int sqrt_i = sqrt(i);
6         if (sqrt_i * sqrt_i == i)
7             cout << i << " 是平方数 ";
8     }
9 }

```

2.3.3 范例：寻找数组里的最小值

由小到大枚举数组索引值，逐一比较数组元素。

	0	1	2	3	4	
	3	6	9	-8	1	寻找最小值
1.	3	6	9	-8	1	目前最小值是 3
2.	3	6	9	-8	1	目前最小值是 3
3.	3	6	9	-8	1	目前最小值是 3
4.	3	6	9	-8	1	目前最小值是 -8
5.	3	6	9	-8	1	目前最小值是 -8

```

1 void find_minimum()
2 {
3     int array[5] = {3, 6, 9, -8, 1};
4
5     int min = 2147483647;
6     for (int i=0; i<5; i++) // 枚举索引值
7         if (array[i] < min) // 比较元素
8             min = array[i]; // 随时纪录最小值
9
10    cout << " 最小的数字是 " << min;
11 }

```

```

1 int find_minimum(int array[], int n)
2 {
3     int min = 2147483647;
4     for (int i=0; i<n; i++) // 枚举索引值
5         if (array[i] < min) // 比较元素
6             min = array[i]; // 随时纪录最小值

```

```

7     return min;
8 }

```

2.3.4 范例：寻找数组里的特定数字

找到所有特定数字：浏览一遍所有数字。

	0	1	2	3	4	
	3	6	9	-8	1	寻找 6
1.	3	6	9	-8	1	不是 6
2.	3	6	9	-8	1	是 6
3.	3	6	9	-8	1	不是 6
4.	3	6	9	-8	1	不是 6
5.	3	6	9	-8	1	不是 6

```

1 void find_all_number()
2 {
3     int array[5] = {3, 6, 9, -8, 1};
4
5     for (int i=0; i<5; i++) // 枚举
6         if (array[i] == 6) // 搜索
7             cout << i << ':' << array[i] << '\n';
8 }

```

找到其中一个特定数字：一旦找到，立即停止浏览，以节省时间。

	0	1	2	3	4	
	3	6	9	-8	1	寻找 6
1.	3	6	9	-8	1	不是 6
2.	3	6	9	-8	1	找到了 6，停止

```

1 bool find_number()
2 {
3     int array[5] = {3, 6, 9, -8, 1};
4
5     for (int i=0; i<5; i++) // 枚举
6         if (array[i] == 6) // 搜索
7             {

```

```

8             cout << i << ':' << array[i];
9             return true;
10        }
11        return false;
12    }

```


```

1 int find_number(int array[i], int n, int num)
2 {
3     for (int i=0; i<n; i++)
4         if (array[i] == num)
5             return i;
6     return -1;
7 }

```

2.3.5 范例：寻找二维数组里的特定数字

	0	1	2	3	4
0	3	6	9	-8	1
1	2	4	6	8	0
2	7	5	3	2	1



	0	1	2	3	4
0	3	6	9	-8	1
1	2	4	6	8	0
2	7	5	3	2	1

多个元素成为一个横条、多个横条成为一个数组。内层先枚举元素，外层再枚举横条，就能枚举所有元素。

方才是由内而外、由小到大进行思考，其实也可以由外而内、由大到小进行思考：外层先枚举每一个横条，内层再枚举一个横条的每一个元素，就能枚举所有元素。

```

1 bool find(int n)
2 {
3     int array[3][5] =
4     {
5         {3, 6, 9, -8, 1},
6         {2, 4, 6, 8, 10},
7         {11, 7, 5, 3, 2}
8     };
9
10    // 外层枚举每一个横条
11    for (int i=0; i<3; i++)

```



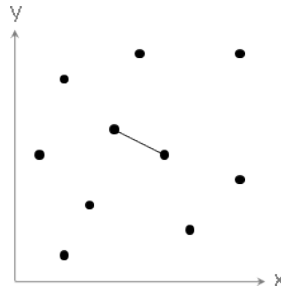
```
12         // 内层枚举一个横条的每一个元素
13         for (int j=0; j<5; j++)
14             // 就能枚举所有元素
15             if (array[i][j] == n)
16                 return true;
17     return false;
18 }
```

此处再介绍一种特别的思考方式：第一层枚举每一个横条，第二层枚举每一个直条，就能枚举所有直条与横条的交错之处。

虽然前后两个思考方式完全不同，但是前后两支程序代码却完全相同。

```
1 bool find(int n)
2 {
3     int array[3][5] =
4     {
5         {3, 6, 9, -8, 1},
6         {2, 4, 6, 8, 10},
7         {11, 7, 5, 3, 2}
8     };
9
10    // 第一层枚举每一个横条
11    for (int i=0; i<3; i++)
12        // 第二层枚举每一个直条
13        for (int j=0; j<5; j++)
14            // 就能枚举所有横条与直条交错之处
15            if (array[i][j] == n)
16                return true;
17    return false;
18 }
```

2.3.6 范例：平面上距离最近的两个点（Closest Pair Problem）



第一层枚举第一个点，第二层枚举第二个点。为了避免重复枚举相同的一对点，第二层只枚举索引值更高的点。

```

1 void closest_pair()
2 {
3     float point[10][2] =
4     {
5         {3, 3}, {1, 5}, {4, 6}, {2, 8}, {9, 9},
6         {2, 1}, {7, 2}, {6, 5}, {9, 4}, {5, 9}
7     };
8
9     // 距离最近的两个点的距离
10    float d = 1e9;
11
12    // 枚举第一点
13    for (int i=0; i<10; i++)
14        // 枚举第二点
15        for (int j=i+1; j<10; j++)
16        {
17            // 计算第一点到第二点的距离
18            float dx = point[i][0] - point[j][0];
19            float dy = point[i][1] - point[j][1];
20            float dij = sqrt(dx * dx + dy * dy);
21
22            // 纪录最短的距离
23            if (dij < d) d = dij;
24        }
25    cout << "距离是" << d;

```

26 }

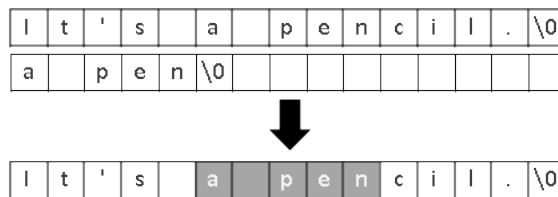
可以把计算距离的程序代码，抽离出来成为一个函数。好处是程序代码变得清爽许多，增加程序代码可读性。坏处是大量调用函数，导致执行速度变慢。

```
1 struct Point {float x, y;};
2
3 // 计算两点之间的距离
4 float dist(Point& a, Point& b)
5 {
6     float dx = a.x - b.x;
7     float dy = a.y - b.y;
8     return sqrt(dx * dx + dy * dy);
9 }
10
11 void closest_pair()
12 {
13     Point point[10] =
14     {
15         {3, 3}, {1, 5}, {4, 6}, {2, 8}, {9, 9},
16         {2, 1}, {7, 2}, {6, 5}, {9, 4}, {5, 9}
17     };
18
19     float d = 1e9;
20     for (int i=0; i<10; i++)
21         for (int j=i+1; j<10; j++)
22             // 纪录最短的距离
23             d = min(d, dist(point[i], point[j]));
24
25     cout << "距离是" << d;
26 }
```

鱼与熊掌不可兼得，这两种程序代码各有优缺点，没有绝对的好坏。程序员必须自行取舍。

2.3.7 范例：字符串匹配 (String Matching)

从长字符串之中，找到短字符串的出现位置。



第一层先枚举所有可以匹配的位置，第二层再枚举所有需要匹配的字符。

```

1 void string_matching()
2 {
3     char text[15] = "It's_a_pencil.";
4     char pattern[6] = "a_pen";
5
6     // 枚举所有可以匹配的位置
7     for (int i=0; i<14; i++)
8     {
9         // 枚举所有需要匹配的字符
10        bool match = true;
11        for (int j=0; j<5; j++)
12            if (text[i+j] != pattern[j])
13                match = false;
14
15        if (match)
16            cout << "短字符串出现在第 " << i << " 个字符 ";
17    }
18 }

```

因为短字符串不会超出长字符串末段，所以第一层枚举范围可以再略微缩小。

因为只要一个相异字符，就足以表明匹配位置错误，所以第二层的枚举过程可以提早结束。

```

1 void string_matching()
2 {
3     char text[15] = "It's_a_pencil.";
4     char pattern[6] = "a_pen";
5
6     // 仔细估量枚举范围

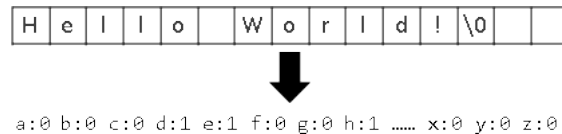
```

```

7     for (int i=0; i<14-6+1; i++)
8     {
9         bool match = true;
10        for (int j=0; j<5; j++)
11            if (text[i+j] != pattern[j])
12            {
13                match = false;
14                break;
15            }
16
17        if (match)
18            cout << "短字符串出现在第 " << i << " 个字符";
19    }
20 }

```

2.3.8 范例：统计字母数量



第一层先枚举 26 种英文字母，第二层再枚举字符串的所有字符，计算一种字母的数量。

```

1 void count_letter()
2 {
3     char s[15] = "Hello World!";
4
5     // 字母统一换成小写
6     for (int i=0; s[i]; i++)
7         if (s[i] >= 'A' && s[i] <= 'Z')
8             s[i] = s[i] - 'A' + 'a';
9
10    // 枚举 26 种英文字母
11    for (int i=0; i<26; i++)
12    {
13        // 枚举字符串的所有字符
14        int c = 0;

```

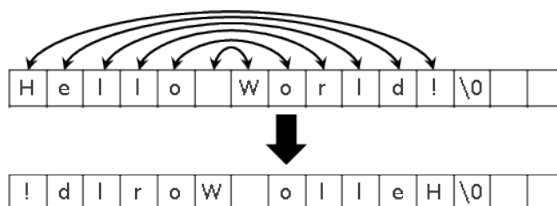
```

15         for (int j=0; s[j]; j++)
16             if (s[j] == i)
17                 c++;
18
19         // 印出一种字母的数量
20         cout << (char)i << ':' << c;
21     }
22 }

```

先前曾经介绍过统计字母数量的范例。先前范例当中，虽然耗费内存空间，但是执行速度快——简单来说就是空间大、时间小。此处范例当中，则是空间小，时间大，恰恰相反。这两种方式各有优缺点，程序员必须自行取舍。

2.3.9 范例：反转字符串



两个枚举，一个从头到尾，一个从尾到头，步调相同，逐步对调字符。虽然是两个枚举，却只有一个循环。

```

1 void reverse_string()
2 {
3     char s[15] = "Hello World!";
4
5     // 两个枚举，一个从头到尾，一个从尾到头。
6     for (int i=0, j=12; i<j; i++, j--)
7         swap(s[i], s[j]);
8
9     cout << "反转之后的字符串是" << s;
10 }

```

```

1 void reverse(char* s)
2 {

```

```

3     int n = strlen(s);
4     for (int i=0; i<n/2; i++)
5         swap(s[i], s[n-1-i]);
6 }

```

2.3.10 范例：寻找总和为 10 的区间

假设数组元素只有正数。

	0	1	2	3	4	
	3	6	1	7	2	尋找總和為10的區間
1.	3	6	1	7	2	目前總和是 3
2.	3	6	1	7	2	目前總和是 9
3.	3	6	1	7	2	目前總和是 10
4.	3	6	1	7	2	目前總和是 17
5.	3	6	1	7	2	目前總和是 14
6.	3	6	1	7	2	目前總和是 8
7.	3	6	1	7	2	目前總和是 10

两个枚举，枚举区间左端以及枚举区间右端，都是从头到尾，保持一左一右，视情况轮流枚举。虽然是两个枚举，却只有一个循环。

```

1 void find_interval()
2 {
3     int array[5] = {3, 6, 1, 7, 2};
4
5     int sum = 0;
6     for (int i=0, j=-1; j<5; ) // 枚举区间 [i, j]
7     {
8         if (sum > 10)
9         {
10             // 总和太大，区间左端往右缩短。
11             sum -= array[i];
12             i++;
13         }
14         else if (sum < 10)
15         {
16             // 总和太小，区间右端往右伸长。
17             j++;

```

```

18         sum += array[j];
19     }
20     else if (sum == 10)
21     {
22         // 总和刚好,
23         // 区间左端往右缩短,
24         // 亦得区间右端往右伸长。
25         // 任选一种皆可。
26     //     sum -= array[i];
27     //     i++;
28         j++;
29         sum += array[j];
30     }
31
32     if (sum == 100)
33         cout << '[' << i << ', ' << j << ']'<
34     }
35 }

```

```

1 void find_interval(int array[], int n, int num)
2 {     int sum = 0;
3     for (int i=0, j=0; j<=n; ) // 枚举区间 [i, j)
4     {
5         if (sum > num)
6             sum -= array[i++];
7         else
8             sum += array[j++];
9
10        if (sum == num)
11            cout << '[' << i << ', ' << j-1 << ']'<
12    }
13 }

```

读者可以想想看：数组元素若有零、有负数，是否要调整枚举方式？

2.3.11 范例：寻找数组之中的最小值，数组已经由小到大排序

找到其中一个最小值：经常整理房间，寻找东西就快；预先排序数据，搜索速度就快。

1.

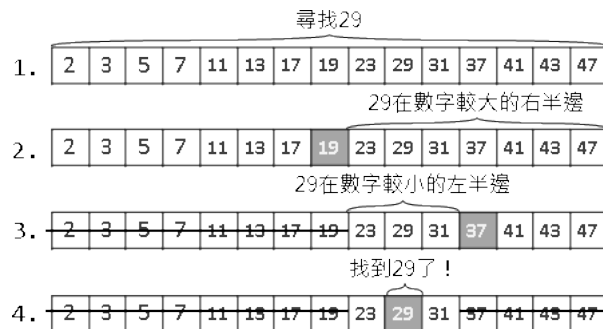
3	3	6	6	9
---	---	---	---	---

```
1 void find_minimum()
2 {
3     int array[5] = {3, 3, 6, 6, 9};
4     cout << "最小的数字是" << array[0];
5 }
```

找到所有最小值：读者请自行尝试。

2.3.12 范例：寻找数组之中的特定数字，数组已经由小到大排序

找到其中一个特定数字：首先找到数组中央的数字，依其数字大小，继续搜索左半段或者右半段。



```
1 void find_number()
2 {
3     int array[15] =
4     {
5         2, 3, 5, 7, 11,
6         13, 17, 19, 23, 29,
7         31, 37, 41, 43, 47
8     };
9
10    int left = 0, right = 15-1;
11    while (left < right)
```

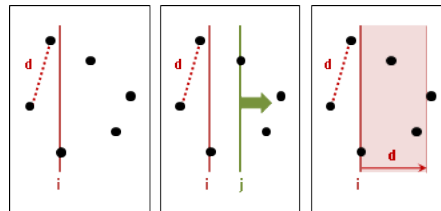
```

12     {
13         int mid = (left + right) / 2;
14         if (array[mid] < 29)
15             left = mid + 1;        // 继续搜索剩下的右半段
16         else if (array[mid] > 29)
17             right = mid - 1;       // 继续搜索剩下的左半段
18         else if (array[mid] == 29)
19         {
20             // 找到了其中一个数字
21             cout << mid << ':' << array[mid];
22             return;
23         }
24     }
25 }

```

找到所有特定数字：读者请自行尝试。

2.3.13 范例：平面上距离最近的两个点 (Closest Pair Problem)



找到距离最近的其中一对点：预先依照 X 坐标排序所有点，搜索得以略过大量情况。

```

1  struct Point {float x, y;};
2
3  // 计算两点之间的距离
4  float dist(Point& a, Point& b)
5  {
6      float dx = a.x - b.x;
7      float dy = a.y - b.y;
8      return sqrt(dx * dx + dy * dy);
9  }
10

```

```
11 bool cmp(const Point& i, const Point& j)
12 {
13     return i.x < j.x;
14 }
15
16 void closest_pair()
17 {
18     Point point[10] =
19     {
20         {3, 3}, {1, 5}, {4, 6}, {2, 8}, {9, 9},
21         {2, 1}, {7, 2}, {6, 5}, {9, 4}, {5, 9}
22     };
23
24     // 依照X坐标排序所有点
25     sort(point, point+10, cmp);
26
27     float d = 1e9;
28     for (int i=0; i<10; i++)
29         for (int j=i+1; j<10; j++)
30         {
31             // 两个点的X坐标已经相距太远，直接略过，
32             // 继续枚举下一个左端点。
33             if (p[j].x - p[i].x > d) break;
34             d = min(d, dist(point[i], point[j]));
35         }
36
37     cout << "距离是" << d;
38 }
```

找到距离最近的每一对点：读者请自行尝试。

2.3.14 范例：英文单字从单数变复数

枚举各种情况，写成大量判断式。

1. -y → -ies
2. -s → -ses
3. -x → -xes
4. -ch → -ches
5. -sh → -shes
6. -man → -men

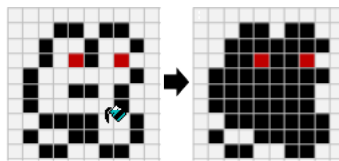
```

1 void plural(string s)
2 {
3     int n = s.length();
4     if (s.back() == 'y')
5         cout << s.substr(0, n-1) << "ies";
6     else if (s.back() == 's' || s.back() == 'x')
7         cout << s << "es";
8     else if (s.substr(n-2) == "sh" || s.substr(n-2) == "ch")
9         cout << s << "es";
10    else if (s.substr(n-3) == "man")
11        cout << s.substr(0, n-3) << "men";
12    else
13        cout << s << 's';
14 }

```

2.3.15 范例：小画家倒墨水 (Flood Fill Algorithm)

计算机图片可以想成是一张方格纸，每个方格都填着一种颜色。现在要实现小画家倒墨水的功能：以某一格为起点，只要相邻方格颜色一样，就染成同一个颜色。



运用大量指令，枚举上下左右四个方向；运用递归，枚举相邻同色方格。必须避免已经枚举过的方格又重复枚举，否则程序在有生之年都不会结束。

```

1 int image[10][10];           // 图片的大小为 10x10
2
3 void flood(int x, int y, int new_color, int old_color)

```

```
4 {
5     if (x>=0 && x<10 && y>=0 && y<10)    // 不能超出边界
6         if (image[x][y] == old_color)    // 同色方格才枚举
7             {
8                 // 染色
9                 image[x][y] = new_color;
10                // 枚举上下左右四个方向
11                flood(x+1, y, new_color, old_color);
12                flood(x-1, y, new_color, old_color);
13                flood(x, y+1, new_color, old_color);
14                flood(x, y-1, new_color, old_color);
15            }
16 }
17
18 void ink()
19 {
20     // 在坐标 (7,6) 的方格，淋上 1 号颜色。
21     flood(7, 6, 1, image[7][6]);
22 }
```

大量指令，亦得写成一个循环。

```
1 void flood(int x, int y, int new_color, int old_color)
2 {
3     if (x>=0 && x<10 && y>=0 && y<10)
4         if (image[x][y] == old_color)
5             {
6                 image[x][y] = new_color;
7
8                 // 写成一个循环
9                 for (int i=0; i<4; i++)
10                    {
11                        static int dx[4] = {1, -1, 0, 0};
12                        static int dy[4] = {0, 0, 1, -1};
13                        flood(x + dx[i], y + dy[i], new_color, old_color);
14                    }
15            }
16 }
```

多层判断式，亦得拆解成一层一层的判断式。

```

1 void flood(int x, int y, int new_color, int old_color)
2 {
3     if (!(x>=0 && x<10 && y>=0 && y<10)) return;
4     if (image[x][y] != old_color) return;
5
6     image[x][y] = new_color;
7
8     // 写成一个循环
9     for (int i=0; i<4; i++)
10    {
11        static int dx[4] = {1, -1, 0, 0};
12        static int dy[4] = {0, 0, 1, -1};
13        flood(x + dx[i], y + dy[i], new_color, old_color);
14    }
15 }
```

UVa 260 280 352 469
572 601 657 776 782
784 785 871 10267
10336 10946
ICPC 4792 5130

2.3.16 Straightforward Method / Trial and Error

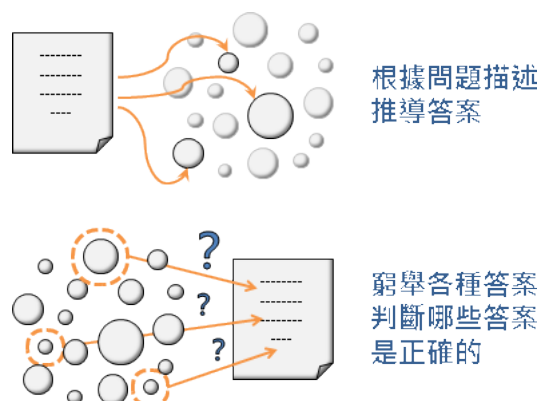
「直接法」，直接算出答案。例如按照流程进行得到答案、套用公式计算答案、直接印出答案。

UVa 488 10055 10370
10878 10929

「尝试错误法」、「试误法」，针对答案进行 Enumerate 与 Search。有些困难的问题，难以直接推导答案，既然推导不出来，就慢慢测试答案、慢慢验算吧——确立答案的范围，穷举所有可能的答案，再从中搜索正确答案。

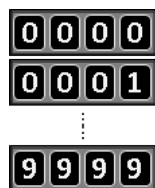
UVa 10167 10125 296
846 714

直接法和试误法刚好相反。直接法是由题目本身下手，推导答案；试误法则是从答案下手，让答案迎合题目需求。



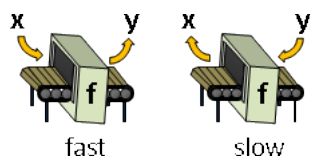
2.3.17 范例：暴力攻击（Brute Force Attack）

破解密码最简单的方法叫做「暴力攻击」。不知道密码规律的情况下，无法直接推导正确密码，只好以试误法一一检验所有可能的密码，从中找出正确密码。



2.3.18 范例：单向函数（One-way Function）

「单向函数」是一种特别的函数，给定输入很容易算出输出，但是给定输出却很难算出输入。



举例来说，令一个函数的输入是两个质数，输出是两个质数的乘积。给定两个质数可以轻易的在多项式时间内算出乘积，然而给定两质数的乘积却需要指数时间才能完成质因子分解。

如果给定一个单向函数的输入，求其输出，就适合用直接法，套用函数快速算得答案；如果给定一个单向函数的输出，求其输入，就适合用试误法，尝试各种输入并套用函数快速验证答案。

2.4 Iterative Method

道生一，一生二，二生三，三生万物。《老子》

2.4.1 Iterative Method

繁中「迭代法」、简中「递推法」。不断利用目前求得的数值，再求得新数值。

UVa 997

2.4.2 范例：字符串变整数

2 6 9 6 2 8 6 9 \0 → 26962869

直觉的方式是递增法。个、十、百、千、万、……，每个位数分别乘上 10 的次方，通通加起来。此处按照高位数到低位数的顺序进行处理，以符合字符串的储存顺序。

1.	2	6	9	6	2	8	6	9	\0	→	2×10^7
2.	2	6	9	6	2	8	6	9	\0	→	+ 6×10^6
3.	2	6	9	6	2	8	6	9	\0	→	+ 9×10^5
											⋮
8.	2	6	9	6	2	8	6	9	\0	→	+ 9×10^0

```

1 // 计算字符串长度
2 int string_length(char* s)
3 {
4     int n = 0;
5     while (s[n]) n++;
6     return n;
7 }
8
9 // 计算 10 的 exp 次方
10 int pow10(int exp)
11 {
12     int n = 1;
13     for (int i=0; i<exp; i++)
14         n *= 10;

```



```
15     return n;
16 }
17
18 void string_to_integer()
19 {
20     char s[10] = "26962869";
21
22     // 预先计算字符串长度。
23     int length = string_length(s);
24
25     // 依序处理高位数到低位数。
26     int n = 0;
27     for (int i=0; i<length; i++)
28         n += (s[i] - '0') * pow10(length - 1 - i);
29
30     cout << n;
31 }
```

更好的方式是递推法！由高位数到低位数、也就是由左到右读取字符串，每读取一个字符，就将数值乘以十、加上当前字符的对应数字。

```
1 void string_to_integer()
2 {
3     char s[10] = "26962869";
4
5     int n = 0;
6     for (int i=0; s[i]; i++)
7         n = n * 10 + s[i] - '0';
8
9     cout << n;
10 }
```

同一个问题，有着不同的解法。有着程序代码很长、执行速度很慢的方法，也有着程序代码很短，执行速度很快的方法。一支程序的好坏，除了取决于正确性和可读性之外，同时也取决于计算方法。

2.4.3 范例：秦九韶算法（Horner’s Rule）

多项式函数，代入数值。一乘一加，不断更迭，求得函数值。完全不需要次方运算。

$f(x) = 2x^3 + 6x^2 + 9x^1 + 6$ $f(10) = 2 \cdot 10^3 + 6 \cdot 10^2 + 9 \cdot 10^1 + 6 = ?$	
Incremental Method ☺	Iterative Method ☺
1. $0 + 2 \times 10^3 = 2000$	- . $0 \times 10 + 2 = 2$
2. $2000 + 6 \times 10^2 = 2600$	1. $2 \times 10 + 6 = 26$
3. $2600 + 9 \times 10^1 = 2690$	2. $26 \times 10 + 9 = 269$
4. $2690 + 6 \times 10^0 = 2696$	3. $269 \times 10 + 6 = 2696$

2.4.4 范例：3n+1 猜想（Collatz Conjecture）

猜想的内容是这样的：有一个整数，如果是偶数，就除以 2；如果是奇数，就乘以 3 再加 1。一个整数不断这样操作下去，最后一定会变成 1。这个操作的过程就是一种递推。

$$\begin{matrix} \times 3 + 1 & \div 2 & \times 3 + 1 & \div 2 & \div 2 & \div 2 & \div 2 \\ 3 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1 \end{matrix}$$

至今尚未有人能证明其正确性。有趣的是，目前也尚未检查出任何反例。

UVa 100 371 694

2.4.5 范例：除法

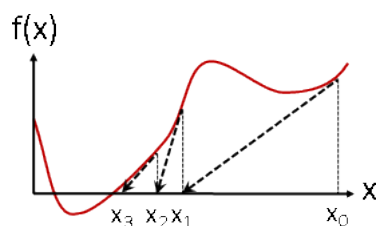
不断乘以十、除以除数，就是一种递推。

	001886	1	
53	10	↓	×10
	0	↓	%52
	100	10	×10
	53	↓	%52
	470	47	×10
	424	↓	%52
	460	46	×10
	424	↓	%52
	360	36	×10
	318	↓	%52
	42	↓	%52
		⋮	

2.4.6 范例：牛顿法（Newton’s Method）

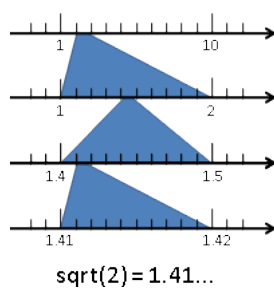
一个经典的递推法范例，微积分课程一定有教过。牛顿法用来求连续函数的其中一个根。一开始先随便设定一点，不断利用斜率求出下一点。

$$X_{n+1} = X_n - \frac{f(X_n)}{f'(X_n)}$$



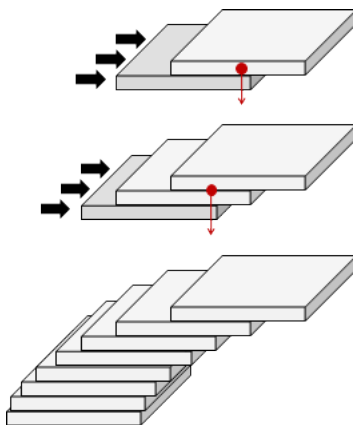
2.4.7 范例：十分逼近法

数线分割成十等份区间，从中找出正确区间，把对应的小数字数添到答案末端，然后不断十等分下去。



2.4.8 范例：书塔（Book Stacking Problem）

将书本一本一本迭起来，成为一座斜塔，越斜越好。



对于任何一本书来说，其上方所有书本的整体重心，必须落在这本书上，这本书才能平稳地支撑住上方所有书本。

将书本插入到书塔底部，让书塔的重心落在书本边缘，就可以让书塔最斜。插入书本到书塔底部之后，就更新书塔的重心位置，以便稍后插入下一本书本。

不断插入书本到书塔底部、更新书塔重心，运用先前的书塔求得新的书塔——这段过程就是一种递推。

2.4.9 范例：生命游戏（The Game of Life）（Cellular Automata）

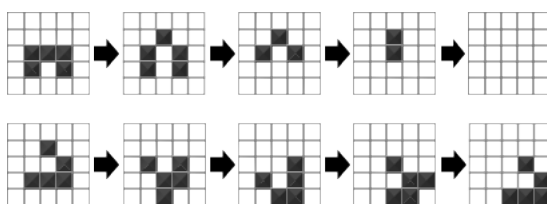
一个二维的方格平面，每个格子都有一个细胞，可能是活的，可能是死的。细胞的生命状况，随时间变动，变动规则如下：

复活：一个死的细胞，若是它的八个邻居，有三个细胞是活的，则在下一刻复活。

存活：一个活的细胞，若是它的八个邻居，有两个或三个细胞是活的，则在下一刻存活。

死于孤单：一个活的细胞，若是它的八个邻居，只有零个或一个细胞是活的，则在下一刻死亡。

死于拥挤：一个活的细胞，若是它的八个邻居，有四个以上的细胞是活的，则在下一刻死亡。



实作时，我们可以弄两张地图，第一张地图储存现在这个时刻的状态，第二张地图储存下一个时刻的状态。两张地图交替使用，以节省内存空间。

细胞的变动规则，包装成一个函数，让程序代码易读。

```

1 void go(int x, int y, bool map1[100][100], bool map2[100][100])
2 {
3     int n = 八个邻居中，还活着的细胞数目；
4
5     if (!map1[x][y])
6         if (n == 3) // 复活
7             map2[x][y] = true;
8         else // 仍旧死亡
9             map2[x][y] = map1[x][y];
10    else
11        if (n == 2 || n == 3) // 存活
12            map2[x][y] = true;

```

```

13         else if (n == 0 || n == 1) // 死于孤单
14             map2[x][y] = false;
15         else if (n >= 4)           // 死于拥挤
16             map2[x][y] = false;
17     }
18
19 void cellular_automata()
20 {
21     bool map[2][100][100];
22
23     map[0][50][50] = true; // 自行设定一些活的细胞
24     map[0][50][51] = true;
25     map[0][51][50] = true;
26
27     for (int t=0; t<100; ++t)
28         for (int x=0; x<100; ++x)
29             for (int y=0; y<100; ++y)
30                 go(x, y, map[t%2], map[(t+1)%2]);
31 }

```

UVa 447 457 10443
10507

2.4.10 范例：兰顿的蚂蚁 (Langton's Ant)

跟生命游戏相似，不过这个游戏更神奇。

- 一、格子有黑与白两种颜色。
- 二、蚂蚁走入白格则右转，走入黑格则左转。
- 三、蚂蚁离开格子时，格子颜色颠倒。

惊人的是，乍看完全没有规律的路线，却在 10647 步之后开始循环。原因至今不明。

UVa 11664

2.4.11 范例：以试除法建立质数表

从表面上来看是两层的枚举法：第一层先枚举正整数，一一试验是否为质数；第二层再枚举所有已知质数，一一试除。

但是从另一个角度来看，利用目前求得的质数，再求出更多质数，其实就是递推法。

2.4.12 范例：数学归纳法（Mathematical Induction）

数学归纳法的第二步骤，就是证明可不可以递推！第二步骤的证明过程中一定会用到递推！

1. 先证明 $n = 1$ 成立。（有时候不见得要从 1 开始。）
2. 假设 $n = k$ 成立，证明 $n = k + 1$ 也会成立。

当 1. 2. 得证，就表示 $n = 1 \cdots \infty$ 全部都成立。

2.5 Recursive Method

易有太极，是生两仪。两仪生四象，四象生八卦。《易传》

2.5.1 Recursive Method

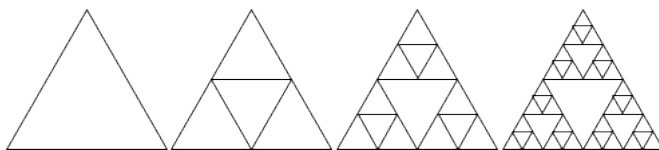
繁中「递归法」、简中「递归法」。重复运用相同手法，缩减问题范围，直到厘清细节。

UVa 10994 10212
10471 10922

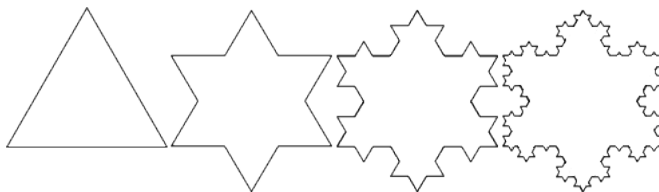
2.5.2 范例：碎形（Fractal）

利用相同手法绘图，绘图范围越来越精细。

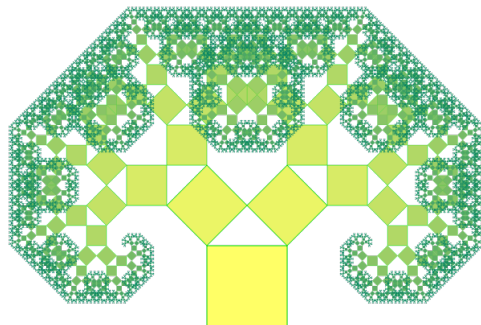
图中的碎形称作 Sierpinski triangle。凡是尖端朝上的正三角形，就在当中放置一个尖端朝下的正三角形；放置之后，图形就变得更细腻，范围就变得更小了。



图中的碎形称作 Koch snowflake。一条边三等分，去除中段，朝外补上两段，形成尖角。



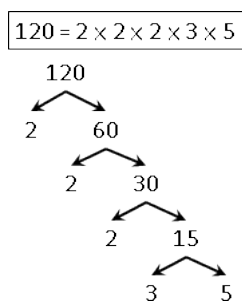
图中的碎形称作 Pythagorean tree 。不断绘制正方形、直角三角形，看起来像是一棵茂密的树。



UVa 177 10609

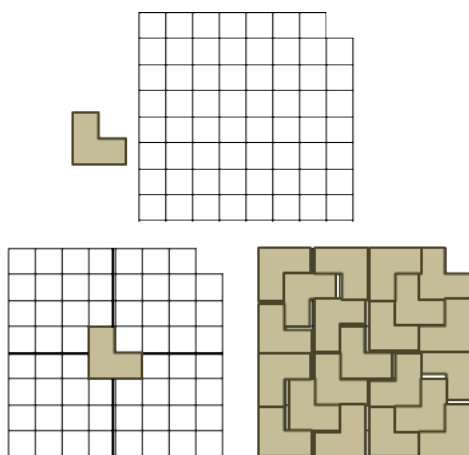
2.5.3 范例：质因子分解（Integer Factorization）

不断抽取出质因子，使数值不断变小，直到成为质因子。



2.5.4 范例：L 形磁砖

有一个边长为 2 的 3 次方的正方形，右上角缺了一角边长为 1 的正方形。现在要以 L 形磁砖贴满这个缺了一角的正方形，该如何贴呢？



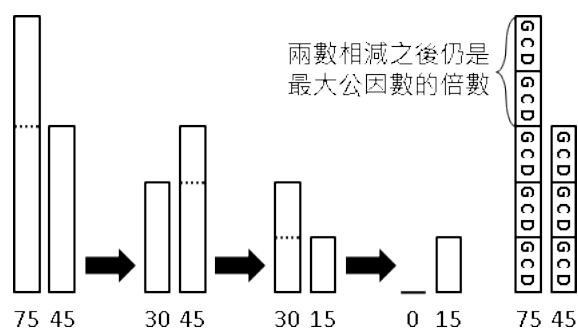
巧妙地将一块 L 形磁砖放在中央的位置，就顺利的把正方形切成四个比较小的、亦缺了一角的正方形。接下来只要递归处理四个小正方形，就解决问题了。

这个问题也可以改成缺口在任意一处，各位可以想想看怎么解。

UVa 10230

2.5.5 范例：辗转相除法（Euclid's Algorithm）

两个数字轮流相除、求余数，最后就得到最大公因子（greatest common divisor, gcd）。相信大家小时候都有学过。



我们可以把最大公因子想象成砖块、把两个数字都看成是最大公因子的倍数。

两数相减所得的差值，一定是最大公因子的倍数。更进一步来说，两数相除所得的余数，一定是最大公因子的倍数。辗转相除法的过程当中，两数自始至终都是最大公因子的倍数。

运用这个性质，我们把两数相除、求余数，使得原始数字不断缩小，直到得到最大公因子。真是非常巧妙的递归法！


```
1 // 运用程序语言的循环语法。
2 int gcd(int a, int b)
3 {
4     // 令 a 比 b 大，比较容易思考。
5     while (b != 0)
6     {
7         int t = a % b;
8         a = b;
9         b = t;
10    }
11    return a;
12 }
```

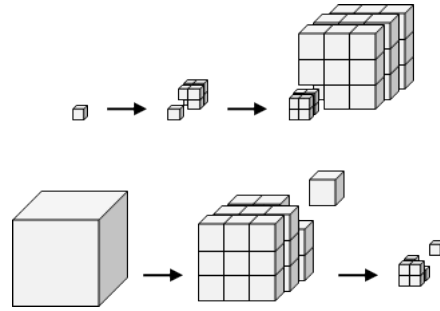
```
1 // 运用程序语言的递归语法。
2 int gcd(int a, int b)
3 {
4     // 令 a 比 b 大，比较容易思考。
5     if (b == 0)
6         return a;
7     else
8         return gcd(b, a % b);
9 }
```

注意到，递推法、递归法，不等于程序语言中的循环、递归。递推法、递归法是分析问题的方法，用来得到计算过程、用来得到算法。至于编写程序时，我们可以自由地采用循环或者递归。

2.5.6 递推法、递归法，一体两面，同时存在。

递推法与递归法恰好颠倒：递推法是针对已知，逐步累积，直至周全；递归法是针对未知，反复拆解，直至精确。

递推法是由小到大，递归法是由大到小。



2.5.7 范例：秦九韶算法（Horner's Rule）

递推法是不不断配 x ，扩增已知；递归法是不不断提 x ，减少未知。

$$ax^2 + bx + c$$

Iterative Method:

$$\{a\} * x^2 + b * x^1 + c$$

$$\{a, *x\} * x^1 + b * x^1 + c$$

$$\{a, *x, +b\} * x^1 + c$$

$$\{a, *x, +b, *x\} + c$$

$$\{a, *x, +b, *x, +c\}$$

Recursive Method:

$$\{a * x^2 + b * x^1 + c\}$$

$$\{a * x^2 + b * x^1\}, +c$$

$$\{a * x^1 + b\}, *x, +c$$

$$\{a * x^1\}, +b, *x, +c$$

$$\{a\}, *x, +b, *x, +c$$

虽然递推法与递归法的推理方向是相反的，但是递推法与递归法的计算方向是一样的，两者都是由小范围算到大范围。

Iterative Method:

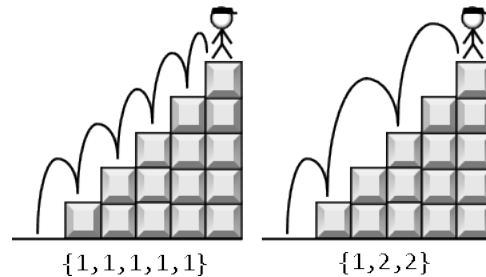
$$a, *x, +b, *x, +c$$

Recursive Method:

$$a, *x, +b, *x, +c$$

2.5.8 范例：爬楼梯

眼前有五阶楼梯，一次只能踏一阶或踏两阶，那么爬到五阶总共有哪几种踏法？例如 (1,1,1,1,1) 是其中一种踏法，(1,2,2) 是另一种踏法。



这个问题可以用递推法，也可以用递归法。

首先采用递推法。试着只爬少少的几阶楼梯，观察一下踏法。

爬到一阶的踏法：很明显的只有一种，(1)。

爬到两阶的踏法：有两种，(1,1) 和 (2)。

爬到三阶的踏法：因为一次只能踏一阶或踏两阶，所以只可能从第一阶或从第二阶踏上第三阶。只要综合 (爬到一阶的踏法,2) 与 (爬到两阶的踏法,1)，就是爬到三阶的踏法。

爬到四阶的踏法：同理，综合 (爬到两阶的踏法,2) 与 (爬到三阶的踏法,1) 即得。

递推下去，就可求出爬到五阶的踏法。

Forward Iterative Method:

- 爬到一阶 (1)
- 爬到两阶 (1,1) (2)
- 爬到三阶 即是 (爬到一阶,2) 与 (爬到二阶,1)
(1,2)
(1,1,1) (2,1)
- 爬到四阶 即是 (爬到二阶,2) 与 (爬到三阶,1)
(1,1,2) (2,2)
(1,2,1) (1,1,1,1) (2,1,1)
- 爬到五阶 即是 (爬到三阶,2) 与 (爬到四阶,1)
(1,2,2) (1,1,1,2) (2,1,2)
(1,1,2,1) (2,2,1) (1,2,1,1) (1,1,1,1,1) (2,1,1,1)

前面是采用上楼梯的顺序进行递推，由第一阶递推到第五阶。也可以采用下楼梯的顺序进行递推，由第五阶递推到第一阶。

Backward Iterative Method:

降到四阶 (1)
 降到三阶 (1,1) (2)
 降到二阶 即是 (2, 降到四阶) 与 (1, 降到三阶)
 (1,2)
 (1,1,1) (2,1)
 降到一阶 即是 (2, 降到三阶) 与 (1, 降到二阶)
 (1,1,2) (2,2)
 (1,2,1) (1,1,1,1) (2,1,1)
 降到平面 即是 (2, 降到二阶) 与 (1, 降到一阶)
 (1,2,2) (1,1,1,2) (2,1,2)
 (1,1,2,1) (2,2,1) (1,2,1,1) (1,1,1,1,1) (2,1,1,1)

有一些问题，比如爬楼梯问题，双向都可以递推。数值由小到大的方向称为「正向」或「顺向」(forward)，数值由大到小的方向称为「反向」或「逆向」(backward)。

接着采用递归法。由踏出的最后一步开始分析。

要「爬到五阶」，最后一步一定是踏上第五阶。要踏上第五阶，只可能从第四阶和第三阶踏过来，也就是综合 (爬到四阶的踏法,1) 与 (爬到三阶的踏法,2)。

但是我们尚不知如何「爬到四阶」和「爬到三阶」，所以只好再分别研究「爬到四阶」与「爬到三阶」。不断追究到「爬到一阶」与「爬到两阶」的时候，就能确认答案了！

Forward(?) Recursive Method:

爬到五阶 即是 (爬到四阶,1) 与 (爬到三阶,2)
 爬到四阶 即是 (爬到三阶,1) 与 (爬到二阶,2)
 爬到三阶 即是 (爬到二阶,1) 与 (爬到一阶,2)
 爬到两阶 (2) (1,1)
 爬到一阶 (1)

当然也可以双向递归。就不赘述了。

2.5.9 范例：格雷码 (Gray Code)

Iterative Method:

GrayCode(n-1) 的每个数字，最高位数加一个 0。

GrayCode(n-1) 的每个数字，高位数与低位数整个颠倒，然后在最高位数加一个 1。

两者衔接起来就是 GrayCode(n)。

Recursive Method:

GrayCode(n) 的每个数字，分成两类。

第一类最高位数是 0，把最高位数拿掉后，即形成 GrayCode(n-1)。

第二类最高位数是 1，把最高位数拿掉后，即形成 GrayCode(n-1)。

也可以用最低位数为主，进行递推、递归，生成顺序不同的 Gray Code。Gray Code 具有循环的特性，有多种递推、递归方式，不分正向与逆向。

2.6 Divide and Conquer

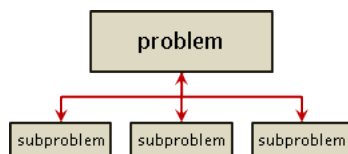
凡治众如治寡，分数是也。斗众如斗寡，形名是也。《孙子》

2.6.1 Divide and Conquer

「分治法」，分割问题、各个击破。将一个大问题，分割成许多小问题。如果小问题还是很难，就继续分割成更小的问题，直到问题变得容易解决。

分割出来的小问题，称作「子问题 subproblem」。解决一个问题，等价于解决所有子问题。

用树形图表达原问题与子问题的关系，最好不过！

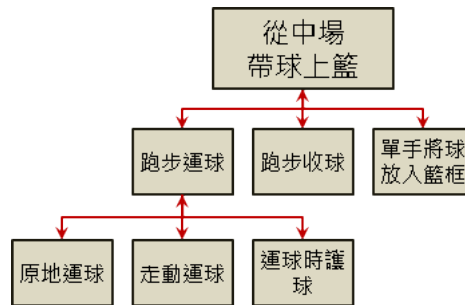


分治法着重分割问题的方式——要怎么分割问题，使得子问题简单又好算？各位读者可以藉由本文的范例，体会分割问题的方式。

2.6.2 范例：分解动作

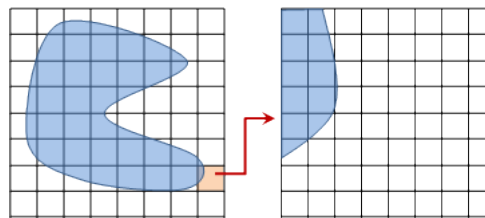
想要学习「从中场带球上篮」，我们可以将此动作分割为「跑步运球」、「跑步收球」、「单手将球放入篮框」等动作，分别学习。每一项动作都熟练之后，组合起来便是带球上篮了。

如果觉得「跑步运球」还是太难，可以更细分成「原地运球」、「走动运球」、「运球时护球」等动作，克服了之后便能够顺利解决「跑步运球」的问题了。



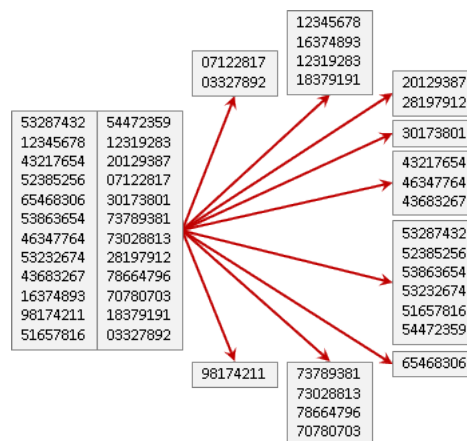
2.6.3 范例：方格法求面积

左边为原问题，右边放大并细分的图是其中一个子问题。



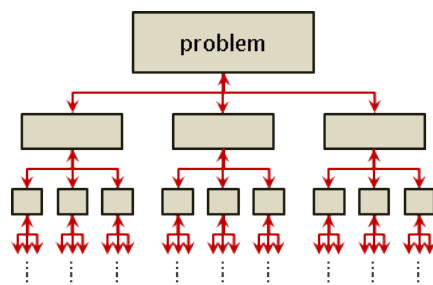
2.6.4 范例：分类数数

左边最大的框框是原问题，将原问题的数字进行分类后再统计，分类后的每一个框框都是一个子问题。

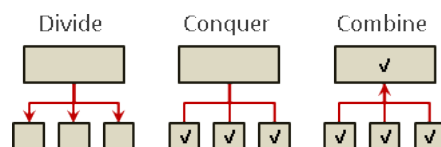


2.6.5 Recursive Method

在分治法当中，亦得递归地分割问题，其实就是递归法。



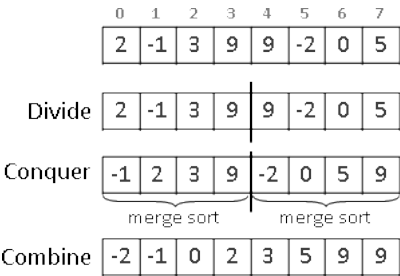
程序代码细分为三个阶段：Divide、Conquer、Combine。Divide 阶段是把原问题分割成小问题，Conquer 阶段是解决小问题，Combine 阶段是运用小问题的解答，整理出原问题的解答。



```

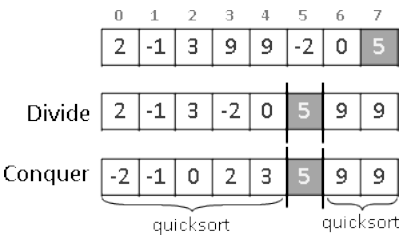
1  divide_and_conquer(原问题)
2  {
3      /* Divide */
4      先将原问题分割成许多小问题；
5
6      /* Conquer */
7      递归调用函数，求得子问题的解；
8      解答1 = divide_and_conquer(子问题1);
9      解答2 = divide_and_conquer(子问题2);
10     .....
11
12     /* Combine */
13     用小问题的解答，算出原问题的解答；
14     原问题解答 = 解答1 + 解答2 + .....;
15
16     return 原问题解答；
17 }
```

2.6.6 范例：归并排序法（Merge Sort）



Divide 阶段：数据分割成两堆。
Conquer 阶段：两堆资料各自从事 Merge Sort 。
Combine 阶段：两堆已排序过的数据，合并成一堆。

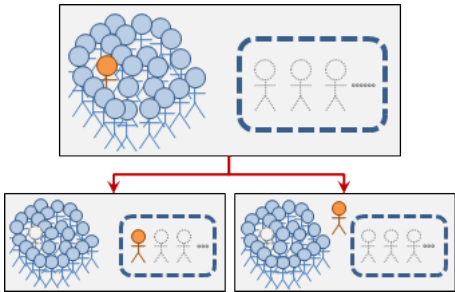
2.6.7 范例：快速排序法（Quicksort）



Divide 阶段：选择一个数值当作基准，把数据分割成左右两堆，使得左堆数值小于基准，右堆数值大于基准，基准数值置于左右两堆中间。
Conquer 阶段：左右两堆资料各自从事 Quicksort 。
Combine 阶段：不做任何事。

2.6.8 范例：不重复组合（Combination）

从 N 个人抓 M 个人出来组团，有哪些组合方式呢？



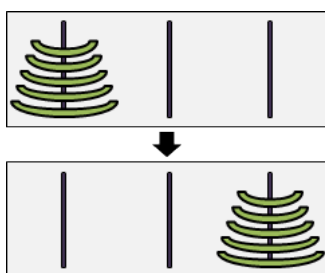
N 个人当中的其中一个人，叫做甲君好了，我们将原问题分割成两种情形：甲君在团中、甲君不在团中。

甲君在团中，演变成剩下 $N-1$ 个人要再抓 $M-1$ 个人出来组团。
甲君不在团中，演变成剩下 $N-1$ 个人仍要抓 M 个人出来组团。

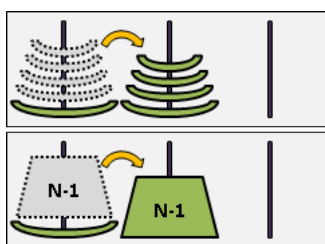
综合这两个子问题的组合方式，就得到答案。

2.6.9 范例：汉诺塔（Tower of Hanoi）

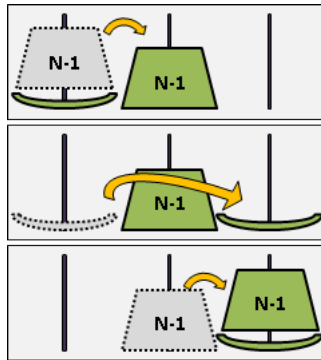
三根柱子、一迭盘子，盘子大小皆不同（盘子中间还得打个洞，这样盘子才能穿在柱子上）。所有盘子都迭在第一根柱子，大的在下面，小的在上面。现在要将整迭盘子移到第三根柱子，并且保持原来的大小顺序。每次只能搬动一个盘子到别根柱子，而且大的盘子一定要保持在小的盘子下面。



想要移动最大的盘子到第三根柱子，必须先挪开上方整迭盘子到第二根柱子。移动上方整迭盘子，正好与原问题相同、而少了一个盘子，可以视作子问题。



尝试以此子问题解决原问题，解题过程因而简化成三个步骤：一、上方整迭盘子移到第二根柱子；二、最大的盘子移到第三根柱子；三、方才的整迭盘子移到第三根柱子。



```

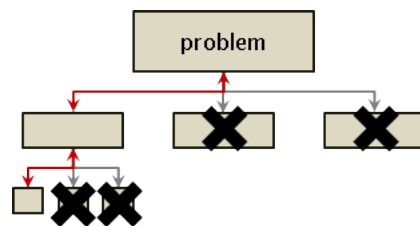
1  int p[5];    // 盘子所在的柱子。第i小的盘子放在第p[i]根柱子。
2
3  void move(int n, int t) // 将前n小的盘子移到第t根柱子
4  {
5      if (n == 0) return;
6      move(n-1, 6-p[n]-t);
7      cout << "从" << p[n] << "移到" << t;
8      p[n] = t;
9      move(n-1, t);
10 }
11
12 void tower_of_hanoi()
13 {
14     // 五个盘子，都迭在第一根柱子。
15     for (int i=1; i<=5; ++i) p[i] = 1;
16     // 五个盘子，从第一根柱子移到第三根柱子。
17     move(5, 3);
18 }

```

UVa 10017

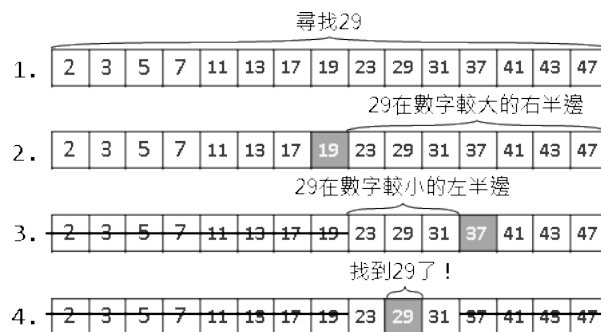
2.6.10 Prune and Search

「修剪搜索法」是分治法的特例。去除不重要的子问题，只搜索重要的子问题。



UVa 920

2.6.11 范例：二分搜索法 (Binary Search)



这是在已排序数组里面搜索数值的方法。数组由中央切成两边，一边数字较小、一边数字较大。这两边一定有一边不是我们所需要的，可以去除，只需要继续寻找其中一边。

UVa 10077

2.6.12 范例：寻找数组里第 k 大的数



运用 Quicksort 的分割手法，把数组切成两边，一边数字较小、一边数字较大。这两边一定有一边不是我们所需要的，可以去除，只需要继续寻找其中一边。

2.6.13 范例：寻找假币（Counterfeit Coin Problem）

一堆硬币，当中一枚硬币是假币，重量比真币轻，肉眼无法分辨差异。手边的工具仅有一台天平，但没有砝码，该如何藉由天平判断假币？

当硬币总数为一，那么该币就是假币。当硬币总数为二，那么就无解了。当硬币总数为三以上，一定有办法找出假币，以下介绍两种策略。

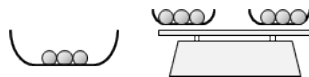
采用递增法。每次取两枚硬币，放在天平两端称重。当天平平平衡，表示这两枚硬币都是真币，接着继续处理剩余硬币。当天平倾斜，比较轻的硬币就是假币。



采用分治法。两枚硬币放在天平两端称重，当天平平平衡，表示这两枚硬币都是真币。接着只剩下 $N-2$ 枚硬币要寻找假币——问题递归缩小了！

剩下 $N-2$ 枚，太多了一点。一次取多一点硬币，同时放在天平两端秤，问题就能缩小更多了！

把所有硬币平均分成三份，取两份放在天平两端称重。当天平平平衡，表示剩下的一份含有假币，问题一次便缩小为 $1/3$ 。当天平倾斜，表示比较轻的一份含有假币，问题一次便缩小为 $1/3$ 。



读者可以想想看：如果硬币数量不是三的次方怎么办？如果一开始不知道假币与真币孰轻孰重怎么办？

2.6.14 Marriage before Conquest

Yu-Han 说道：

2014/1/11 at 9:00

分享一下最近看到的技巧 “marriage-before-conquest”

在做 Divide and conquer 中，递归求解的部份解答可以用来加速剩下的计算。

一个简单的例子是 <http://www.geeksforgeeks.org/sorted-linked-list-to-balanced-bst/> 把已排序的单向链接串行转成平衡的二元搜索树。

如果是用很直接的找中点然后分半递归求解，
时间复杂度是 $O(n \lg n)$ ，

但是实际上建立完左半边的时候，就可以直接得到中点，因此就可以使得时间复杂度降为 $O(n)$ 。

一个稍困难的例子是，给定二维空间中的点集合 S ，

我们称点 p 为 S 中的 maxima，

如果在 S 中没有任何点的 x 坐标以及 y 坐标同时都大于点 p 的 x 坐标与 y 坐标。

问题是要把所有 maxima 找出来。

基于 Divide and conquer 的技巧，利用 x 坐标等分成两半分别找出 maxima，然后把在子问题中是 maxima 但是在考虑整体时不是 maxima 的点删除，如此的时间复杂度可达到 $O(n \lg n)$ 。

使用 marriage-before-coquest 的技巧，

可以把时间复杂度降到 $O(n \lg h)$ ， h 为 maxima 的个数。

方法是先计算右半边的 maxima，

然后利用右半边的 maxima 把左半边中不可能成为全体的 maxima 的点先删除，

最后才计算左半边。

同样的技巧也可以用在 convex hull 上达到 $O(n \lg h)$ 的时间复杂度。

Yu-Han 说道：

2014/1/11 at 23:51

以 convex hull 为例子的话，

是 prune-and-search 和 marriage before conquest 的综合应用。

当有了右半边的 convex hull 的点，要删除左半边不可能为 convex hull 的点时，

需要使用线性规划（利用 prune-and-search），

或是自己设计一个 prune-and-search 的方法。

所以 marriage before conquest 和 prune-and-search 是不同的技巧。

除了这三个例子之外，我也找不太到其它 marriage-before-conquest 的范例了。

2.6.15 Recurrence

递归分割问题时，当子问题与原问题完全相同，只有数值范围不同，我们称此现象为 recurrence，再度出现、一再出现之意。

【注：recursion 和 recurrence，中文都翻译为「递归」，然而两者意义大不相同，读者切莫混淆。】

2.6.16 范例：爬楼梯

先前于递归法章节，已经谈过如何求踏法，而此处要谈如何求踏法数目。

2.7 Dynamic Programming

资之深，则取之左右逢其原。《孟子》

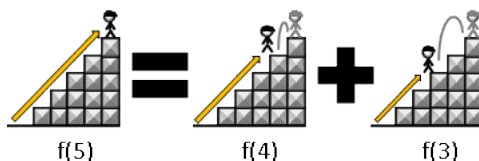
2.7.1 Recurrence

递归分割问题时，当子问题与原问题完全相同，只有数值范围不同，我们称此现象为 recurrence，再度出现、一再出现之意。

【注：recursion 和 recurrence，中文都翻译为「递归」，然而两者意义大不相同，读者切莫混淆。】

2.7.2 范例：爬楼梯

先前于递归法章节，已经谈过如何求踏法，而此处要谈如何求踏法数目。



踏上第五阶，只能从第四阶或从第三阶踏过去。因此「爬到五阶」源自两个子问题：「爬到四阶」与「爬到三阶」。

「爬到五阶」的踏法数目，就是总合「爬到四阶」与「爬到三阶」的踏法数目。写成数学式子是「 $f(5) = f(4) + f(3)$ 」，其中「 $f(\quad)$ 」表示「爬到某阶之踏法数目」。

依样画葫芦，得到「 $f(4) = f(3) + f(2)$ 」、「 $f(3) = f(2) + f(1)$ 」等式子。

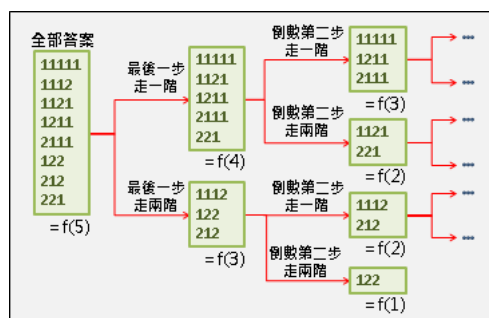
「爬到两阶」与「爬到一阶」无法再分割、没有子问题，直接得到「 $f(2) = 2$ 」、 $f(1) = 1$ 」等式子。

整理成一道简明扼要的递归公式：

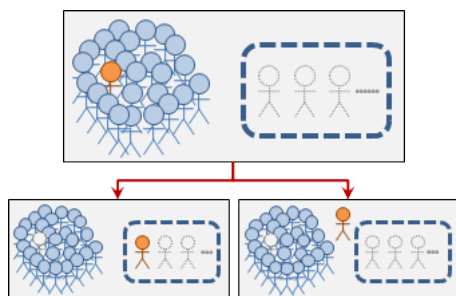
$$f(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2 & \text{if } n = 2 \\ f(n-1) + f(n-2) & \text{if } 3 \leq n \leq 5 \end{cases}$$

爬到任何一阶的踏法数目，都可以藉由这道递归公式求得， n 代入数值、递归计算即可。可以运用「Companion Matrix」或者「Dynamic Programming」迅速求得代入结果。

为什么分割问题之后，就容易计算答案呢？因为分割问题时，同时也分类了这个问题的所有可能答案。分类会使得答案的规律变得单纯，于是更容易求得答案。



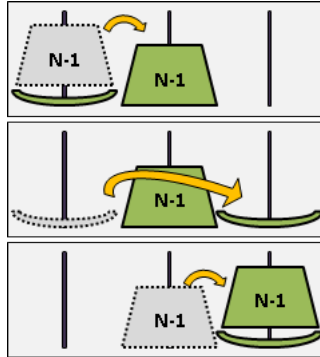
2.7.3 范例：不重复组合 (Combination)



两个子问题的组合数目加起来，就是原问题的组合数目。递归公式就是著名的帕斯卡尔公式 (Pascal's Formula)：

$$\binom{n}{m} = \begin{cases} \binom{n-1}{m-1} + \binom{n-1}{m} & \text{if } n > 1 \text{ and } m > 1 \text{ and } n \geq m \\ n & \text{if } m = 1 \\ 1 & \text{if } n = 1 \end{cases}$$

2.7.4 范例：河内塔（Tower of Hanoi）



$$f(n) = \begin{cases} f(n-1) + 1 + f(n-1) & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

2.7.5 Dynamic Programming = Divide and Conquer + Memoization

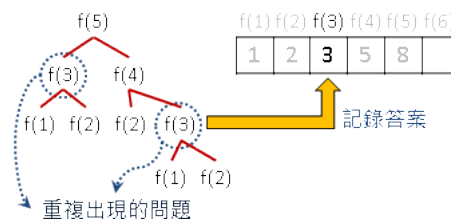
动态规划是分治法的延伸。当递归分割出来的问题，一而再、再而三出现，就运用记忆法储存这些问题的答案，避免重复求解，以空间换取时间。动态规划的过程，就是反复地读取数据、计算数据、储存数据。

Recurrence

$$f(n) = \begin{cases} 1 & , \text{ if } n = 1 \\ 2 & , \text{ if } n = 2 \\ f(n-2) + f(n-1) & , \text{ if } n \geq 3 \end{cases}$$

Divide and Conquer

Memoization



2.7.6 Dynamic Programming State Space Search

动态规划得模拟成 State Space Search：「问题」变「状态」，「全部问题」变「状态空间」，「递归公式」变「状态转移函数」。

2.7.7 用 Dynamic Programming 设计算法时的步骤

1. 把原问题递归分割成许多更小的问题。（recurrence）

- (a) 子问题与原问题的求解方式皆类似。(optimal sub-structure)
- (b) 子问题会一而再、再而三的出现。(overlapping sub-problems)

2. 设计计算过程:

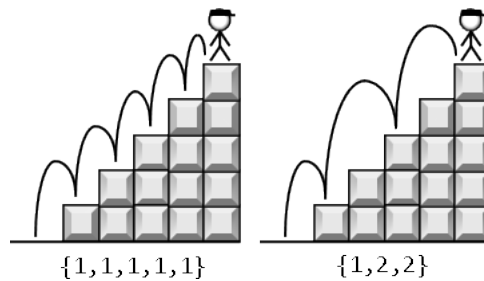
- (a) 确认每个问题需要哪些子问题来计算答案。(recurrence)
- (b) 确认总共有哪些问题。(state space)
- (c) 把问题一一对应到表格。(lookup table)
- (d) 决定问题的计算顺序。(computational sequence)
- (e) 确认初始值、计算范围。(initial states / boundary)

3. 实作，主要有两种方式:

- (a) Top-down
- (b) Bottom-up

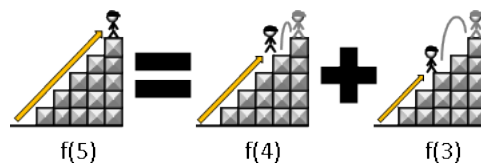
2.7.8 范例：爬楼梯

以先前提过的爬楼梯问题作为范例，说明 DP 的运用方式。



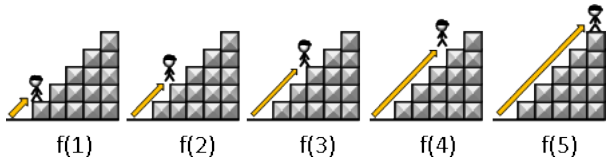
2.7.8.1 recurrence

$$f(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2 & \text{if } n = 2 \\ f(n-1) + f(n-2) & \text{if } n \geq 3 \end{cases}$$



2.7.8.2 state space

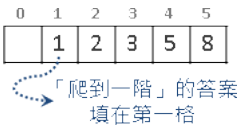
想要计算第五阶的踏法数目。
全部的问题是「爬到一阶」、「爬到二阶」、「爬到三阶」、「爬到四阶」、「爬到五阶」。



至于「爬到零阶」、「爬到负一阶」、「爬到负二阶」以及「爬到六阶」、「爬到七阶」没有必要计算。

2.7.8.3 lookup table

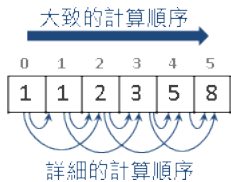
建立六格的数组，储存五个问题的答案。
表格的第零格不使用，第一格是「爬到一阶」的答案，第二格是「爬到二阶」的答案，以此类推。



如果只计算「爬完五阶」，也可以建立三个变量交替使用。

2.7.8.4 computational sequence

因为每个问题都依赖「阶数少一阶」、「阶数少二阶」这两个问题，所以必须由阶数小的问题开始计算。
计算顺序是「爬到一阶」、「爬到二阶」、……、「爬到五阶」。



2.7.8.5 initial states / boundary

最先计算的问题是「爬到一阶」与「爬到二阶」，必须预先将答案填入表格、写入程序代码，才能继续计算其它问题。心算求得「爬到一阶」的答案是 1，「爬到二阶」的答案是 2。

最后计算的问题是原问题「爬到五阶」。

为了让表格更顺畅、为了让程序代码更漂亮，可以加入「爬到零阶」的答案，对应到表格的第零格。「爬到零阶」的答案，可以运用「爬到一阶」的答案与「爬到两阶」的答案，刻意逆推而得。

0	1	2	3	4	5
	1	2	3	5	8

設定初始值

0	1	2	3	4	5
1	1	2	3	5	8

刻意逆推
「爬到零階」的答案

最后可以把初始值、尚待计算的部份、不需计算的部分，统整成一道递归公式： UVa 11069

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ f(n-1) + f(n-2) & \text{if } 2 \leq n \leq 5 \\ 0 & \text{if } n > 5 \end{cases}$$

2.7.9 实作

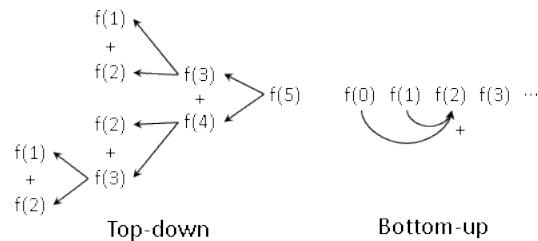
直接用递归实作，而不使用内存储存各个问题的答案，是最直接的方式，也是最慢的方式。时间复杂度是 $O(f(n))$ 。问题一而再、再而三的出现，不断呼叫同样的函数求解，效率不彰。刚接触 DP 的新手常犯这种错误。

```

1 int f(int n)
2 {
3     if (n == 0 || n == 1)
4         return 1;
5     else
6         return f(n-1) + f(n-2);
7 }
```

正确的 DP，是一边计算，一边将计算出来的数值存入表格，以后便不必重算。这里整理了两种实作方式，各有优缺点：

1. Top-down
2. Bottom-up



2.7.9.1 Top-down

```

1  int table[6];    // 表格，储存全部问题的答案。
2  bool solve[6];   // 纪录问题是否已经计算完毕
3
4  int f(int n)
5  {
6      // [Initial]
7      if (n == 0 || n == 1) return table[n] = 1;
8
9      // [Compute]
10     // 如果已经计算过，就直接读取表格的答案。
11     if (solve[n]) return table[n];
12
13     // 如果不曾计算过，就计算一遍，储存答案。
14     table[n] = f(n-1) + f(n-2); // 将答案存入表格
15     solve[n] = true;           // 已经计算完毕
16     return table[n];
17 }
18
19 void stairs_climbing()
20 {
21     for (int i=0; i<=5; i++)
22         solve[i] = false;
23
24     int n;
25     while (cin >> n && (n >= 0 && n <= 5))

```

```
26         cout << "爬到" << n << "阶," << f(n) << "种踏法";
27     }
```

```
1  int table[6];    // 合并 solve 跟 table, 简化程序代码。
2
3  int f(int n)
4  {
5      // [Initial]
6      if (n == 0 || n == 1) return table[n] = 1;
7
8      // [Compute]
9      // 用 0 代表该问题还未计算答案
10 // if (solve[n]) return table[n];
11     if (table[n]) return table[n];
12     return table[n] = f(n-1) + f(n-2);
13 }
14
15 void stairs_climbing()
16 {
17     for (int i=0; i<=5; i++)
18         table[i] = 0;
19
20     int n;
21     while (cin >> n && (n >= 0 && n <= 5))
22         cout << "爬到" << n << "阶," << f(n) << "种踏法";
23 }
```

这个实作方式的好处是不必斤斤计较计算顺序, 因为程序代码中的递归结构会迫使最小的问题先被计算。这个实作方式的另一个好处是只计算必要的问题, 而不必计算所有可能的问题。

这个实作方式的坏处是程序代码采用递归结构, 不断调用函数, 执行效率较差。这个实作方式的另一个坏处是无法自由地控制计算顺序, 因而无法妥善运用内存, 浪费了可回收再利用的内存。

2.7.9.2 Bottom-up

订定一个计算顺序，然后由最小的问题开始计算。特色是程序代码通常只有几个循环。这个实作方式的好处与坏处与前一个方式恰好互补。

首先建立表格。

```
1 int table[6];
```

```
1 int table[5 + 1];
```

心算「爬到零阶」的答案、「爬到一阶」的答案，填入表格当中，作为初始值。分别填到表格的第零格、第一格。

```
1 table[0] = 1;
```

```
2 table[1] = 1;
```

尚待计算的部份就是「爬到两阶」的答案、……、「爬到五阶」的答案。通常是使用循环，按照计算顺序来计算。

计算过程的实作方式，有两种迥异的风格。一种是「往回取值」，是常见的实作方式。

列表 2.1: 往回取值

```
1 int table[6];
2
3 void dynamic_programming()
4 {
5     // [Initial]
6     table[0] = 1;
7     table[1] = 1;
8
9     // [Compute]
10    for (int i=2; i<=5; i++)
11        table[i] = table[i-1] + table[i-2];
12 }
```

另一种是「往后补值」，是罕见的实作方式。

列表 2.2: 往后补值

```
1 int table[6];
```

```
2
```

```

3 void dynamic_programming()
4 {
5     // [Initial]
6     for (int i=0; i<=5; i++) table[i] = 0;
7     table[0] = 1;
8     // table[1] = 1;    // 刚好可以被算到
9
10    // [Compute]
11    for (int i=0; i<=5; i++)
12    {
13        if (i+1 <= 5) table[i+1] += table[i];
14        if (i+2 <= 5) table[i+2] += table[i];
15    }
16 }

```

计算完毕之后，最后印出答案。

```

1 void stairs_climbing()
2 {
3     dynamic_programming();
4
5     int n;
6     while (cin >> n && (n >= 0 && n <= 5))
7         cout << "爬到 " << n << "阶， " << f(n) << "种踏法 ";
8 }

```

UVa 495 900 10334

2.7.10 小结

第一。先找到原问题和其子问题们之间的关系，写出递归公式。如此一来，便可利用递归公式，用子问题的答案，求出子问题的答案；用子问题的答案，求出原问题的答案。

第二。确认可能出现的问题全部总共有哪些，这样才能知道要计算哪些问题，才能知道总共花多少时间、多少内存。

第三。有了递归公式之后，就必须安排出一套计算的顺序。大问题的答案，总是以小问题的答案来求得的，所以，小问题的答案是必须先算的，否则大问题的答案从何而来呢？

一个好的安排方式，不但会使程序代码容易撰写，还可重复利用内存空间。

第四。记得先将最小、最先被计算的问题，心算出答案，储存入表格，内建于程序代码之中。一道递归公式必须拥有初始值，才有办法计算其它项。

第五。实作 DP 的程序时，会建立一个表格，在表格存入所有大小问题的答案。安排好每个问题的答案在表格的哪个位置，这样计算时才能知道该在哪里取值。

切勿存取超出表格的元素，产生溢位情形，导致答案算错。计算过程当中，一旦某个问题的答案出错，就会如骨牌效应般一个影响一个，造成很难调试。

2.7.11 范例：阶乘（Factorial）

N 阶乘， $N!$ ， $1 \times 2 \times 3 \times \cdots \times N$ ，1 到 N 的连乘积。

$$\underbrace{1 \times 2 \times 3 \times \cdots \times (N-2) \times (N-1)}_{(N-1)!} \times N$$

$N!$ 问题依赖 $(N-1)!$ 问题，如此就递归分割问题了。

时间复杂度：总共 N 个问题，每个问题花费 $O(1)$ 时间，总共花费 $O(N)$ 时间。

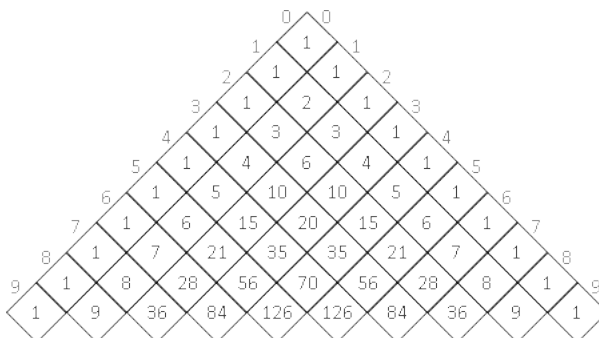
空间复杂度，有两种情况。

求 $1!$ 到 $N!$ ：总共 N 个问题，用一条 N 格数组储存全部问题的答案，空间复杂度为 $O(N)$ 。

求 $N!$ ：用一个变量累积乘积，空间复杂度为 $O(1)$ 。

UVa 623 568 10220
10323

2.7.12 范例：帕斯卡尔三角形（Pascal's Triangle）



帕斯卡尔三角形左右对称，可以精简掉对称部分。帕斯卡尔三角形逆时针转 45，视觉上就可以一一对应至表格。

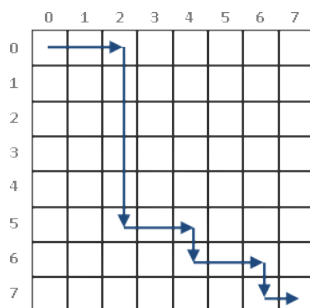
时间复杂度为 $O(N^2)$ ，空间复杂度为 $O(N^2)$ 。

UVa 369 485 10564

2.7.13 Staircase Walk

2.7.13.1 Staircase Walk

一个方格棋盘，从左上角走到右下角，每次只能往右走一格或者往下走一格。请问有几种走法？

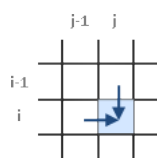
UVa 10599 825 926
ICPC 4787

2.7.13.2 Recurrence

对于某个位置的方格来说，只可能「从左走来」或者「从上走来」，得到递归公式：

$$count(i, j) = \begin{cases} 0 & \text{if } i < 0 \text{ or } j < 0 \\ 1 & \text{if } i = 0 \\ 1 & \text{if } j = 0 \\ count(i-1, j) + count(i, j-1) & \text{if } 0 < i < 8 \text{ and } 0 < j < 8 \\ 0 & \text{if } i \geq 8 \text{ or } j \geq 8 \end{cases}$$

$count(i, j)$: 从格子 $(0, 0)$ 走到格子 (i, j) 的走法数目。



若了解递归关系，就不必强记递归公式。若了解图片意义，就不必强记数学符号。

2.7.13.3 复杂度

时间复杂度分析：令 X 和 Y 分别是棋盘的长和宽。计算一个问题需要 $O(1)$ 时间（计算两个子问题的时间）。总共 $X * Y$ 个问题，所以计算所有问题需要 $O(XY)$ 时间。

空间复杂度分析：总共 $X * Y$ 个问题，所以需要 $O(XY)$ 空间，简单来说就是二维数组啦！如果不需要储存所有问题的答案，只想要得到其中一个特定问题的答案，那只需要一维数组就够了，也就是 $O(\min(X, Y))$ 空间。

2.7.13.4 程序代码

```
1  const int X = 8, Y = 8;
2  int count[X][Y];
3
4  void staircase_walk()
5  {
6      // [Initial]
7      for (int i=0; i<X; i++) count[i][0] = 1;
8      for (int j=0; j<Y; j++) count[0][j] = 1;
9
10     // [Compute]
11     for (int i=1; i<X; i++)
12         for (int j=1; j<Y; j++)
13             count[i][j] = count[i-1][j] + count[i][j-1];
14
15     // 输出结果
16     cout << "由(0,0)走到(7,7)有" << count[7][7] <<
        种走法;
17 // cout << "由(0,0)走到(7,7)有" << count[X-1][Y-1]
        << 种走法;
18
19     int x, y;
20     while (cin >> x >> y)
21         cout << "由(0,0)走到(x,y)有" << count[x][y]
            << 种走法;
22 }
```

2.7.13.5 节省内存空间

如果只打算求出一个问题，那么只需要储存最近算出来的问题答案，让计算过程可以顺利进行就可以了。

使用两条数组，就足够储存最近算出来的问题答案、避免 `count[i-1][j]` 超出数组范围。这个实作技巧在中文网络上称作「滚动数组」，「数组」是数组的意思，「滚动」也就是两条数组轮替使用的意思。

```

1  const int X = 8, Y = 8;
2  int count[2][Y];    // 两个数组，储存最近算出来的问
   题答案。
3
4  void staircase_walk()
5  {
6      // [Initial]
7      for (int j=0; j<Y; ++j) count[0][j] = 1;
8
9      // [Compute]
10     for (int i=1; i<X; i++)
11         for (int j=1; j<Y; j++)
12             // 只是多了 mod 2,
13             // 外观看起来就像两个数组轮替使用。
14             count[i % 2][j] = count[(i-1) % 2][j] +
               count[i % 2][j-1];
15
16     // 输出结果
17     cout << "由(0,0)走到(7,7)有" << count[7 % 2][7]
           << 种走法;
18 //  cout << "由(0,0)走到(7,7)有" << count[(X-1) %
           2][Y-1] << 种走法;
19 }
```

不过事实上，一个数组就够了。也不能再少了。

```

1  const int X = 8, Y = 8;
2  int count[Y];    // 一个数组就够了
3
4  void staircase_walk()
5  {
```

```

6      // [Initial]
7      for (int j=0; j<Y; ++j) count[j] = 1;
8
9      // [Compute]
10     for (int i=1; i<X; i++)
11         for (int j=1; j<Y; j++)
12             count[j] += count[j-1];
13
14     // 输出结果
15     cout << "由(0,0)走到(7,7)有" << count[7] << 种
        走法;
16 //  cout << "由(0,0)走到(7,7)有" << count[Y-1] <<
        种走法;
17 }

```

```

1  const int X = 8, Y = 8;
2  int count[Y];    // 一个数组就够了
3
4  void staircase_walk()
5  {
6      // [Initial]
7      count[0] = 1;    // 部分步骤移到 [Compute]
8
9      // [Compute]
10     for (int i=0; i<X; i++) // 从零开始!
11         for (int j=1; j<Y; j++)
12             count[j] += count[j-1];
13
14     // 输出结果
15     cout << "由(0,0)走到(7,7)有" << count[7] << 种
        走法;
16 //  cout << "由(0,0)走到(7,7)有" << count[Y-1] <<
        种走法;
17 }

```

2.7.13.6 往其它方向走的话？

如果某些格子上有障碍物呢？其实也很简单，如果某格有障碍物，在计算过程中，遇到障碍物就把此格的 $c(i, j)$ 设为零就可以了。

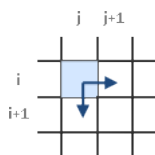
如果也可以往右下斜角走呢？那么递归公式就再修改一下，多加一项 $c(i-1, j-1)$ 就行了。

如果可以往上下左右走呢？那么就可以不断绕圈子，走法就成了无限多种了。写成递归公式的话，就会产生无穷递归，永远也不会结束。

如果也可以往右上斜角走呢？因为不会产生无穷递归，所以这是可以解的！

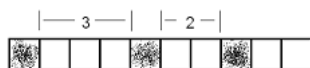
2.7.13.7 双向都可以递归

对某个位置的方格来说，只可能「向右走出」或者「向下走出」，如此得到另一道递归公式。

**2.7.13.8 一般公式解**

除了递归公式之外，其实也有一般公式：¹

$$\binom{m+n}{m} = \frac{(m+n)!}{m!n!}$$

2.7.14 Largest Empty Interval**2.7.14.1 问题描述（离散版本）**

一个数组，有些格子已被放上障碍物。最长的、连续的空白格子在哪里？

¹<http://mathworld.wolfram.com/StaircaseWalk.html>

2.7.14.2 Recurrence

$$length(i) = \begin{cases} 0 & \text{if } i < 0 \\ 0 & \text{if } i = 0 \text{ and } array[i] = 0 \\ 1 & \text{if } i = 0 \text{ and } array[i] = 1 \\ 0 & \text{if } i > 0 \text{ and } array[i] = 0 \\ length(i-1) + 1 & \text{if } i > 0 \text{ and } array[i] = 1 \end{cases}$$

$length(i)$: 以第 i 格作为最右端的连续空白的长度。

$array[i]$: 障碍物为 0, 空白为 1。

2.7.14.3 复杂度

时间复杂度为 $O(N)$, 空间复杂度为 $O(N)$, N 为数组长度。

如果只想计算一个特定问题的答案, 那么空间复杂度可以精简成 $O(1)$, 这个部份就不多提了, 交给各位来处理。

2.7.14.4 程序代码: 求出最长空白的长度

```

1  int array[10] =
2  {
3      0, 1, 1, 1, 0,
4      1, 1, 0, 1, 1
5  };
6
7  int length[10];
8
9  void largest_empty_interval()
10 {
11     // initial
12     if (array[0] == 0)
13         length[0] = 0;
14     else
15         length[0] = 1;
16
17     // compute
18     for (int i=1; i<10; i++)

```

```
19         if (array[i] == 0)
20             length[i] = 0;
21         else
22             length[i] = length[i-1] + 1;
23
24     // 输出结果
25     int max_length = 0;
26     for (int i=0; i<10; i++)
27         if (length[i] > max_length)
28             max_length = length[i];
29
30     cout << "最长空白的长度是" << max_length;
31 }
```

2.7.14.5 程序代码：求出最长空白的长度

为了让程序代码更清爽，这里把 `array[]`、`length[]` 里面的数值都往右移动一格，如此就可以省略掉第零格的判断式，也避免了 `length[]` 会溢出边界。

```
1  int array[10 + 1] =
2  {
3      0,
4      0, 1, 1, 1, 0,
5      1, 1, 0, 1, 1
6  };
7
8  int length[10 + 1];
9
10 void largest_empty_interval()
11 {
12     // initial
13     length[0] = 0;
14
15     // compute
16     for (int i=1; i<=10; i++)
17         if (array[i] == 0)
```

```
18         length[i] = 0;
19     else
20         length[i] = length[i-1] + 1;
21
22     // 输出结果
23     ...
24 }
```

为了让程序代码更清爽，这里也把 `length[]` 都初始化为 0，如此就不必特别处理 `array[i] == 0` 的情况了，相当巧妙。

```
1  int array[10 + 1] =
2  {
3      0,
4      0, 1, 1, 1, 0,
5      1, 1, 0, 1, 1
6  };
7
8  int length[10 + 1];
9
10 void largest_empty_interval()
11 {
12     // initial
13     memset(length, 0, sizeof(length));
14
15     // compute
16     for (int i=1; i<=10; i++)
17         if (array[i] == 1)
18             length[i] = length[i-1] + 1;
19
20     // 输出结果
21     ...
22 }
```

这两个技巧是经常使用的的实作技巧，不仅简化了程序代码的结构，也增加了程序的效率。一定要学会！

2.7.14.6 程序代码：求出最长空白的位置

求出最长空白的长度之后，在最后加上一段程序代码就可以了。当然可以再改进，就交给各位了。

```

1 void largest_empty_interval()
2 {
3     .....
4
5     // 求出所有最长空白的位置
6     for (int i=1; i<=10; i++)    // 从1开始
7         if (length[i] == max_length)
8         {
9             // 记得减回1
10            cout << "有一个最长空白的位置是"
11                << "从" << (i - max_length + 1) - 1
12                << "到" << (i
13                    1;
14            }
15 }
```

2.7.14.7 程序代码：求出其中一个最长空白的位置

也有人会一边计算表格，一边纪录最大值。这种写法也是很好的，不过只能求出其中一个最长空白的位置。

如果只需要求出随便一种最长空白的位置，那么这种写法就非常适合。

```

1 void largest_empty_interval()
2 {
3     // initial
4     memset(length, 0, sizeof(length));
5
6     // compute
7     int max_length = 0;
8     int index = 0;
9
10    for (int i=1; i<=10; i++)
11        if (array[i] == 1)
12        {
```

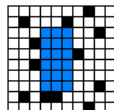
```

13         length[i] = length[i-1] + 1;
14
15         if (length[i] > max_length)
16         {
17             max_length = length[i];
18             index = i;
19         }
20     }
21
22     // 输出结果
23     cout << "最长空白的长度是" << max_length;
24     cout << "有一个最长空白的位置是"
25         << "从" << (index - max_length + 1) - 1
26         << "到" << (index
27     }

```

2.7.15 Largest Empty Rectangle

2.7.15.1 问题描述（离散版本）



一张方格纸，有许多格子填上了黑色。请找出不包含黑格子的矩形，并且令矩形面积尽量大。

矩形的顶点，可以直接想做是一整个格子，而不是想做直线与横线的交叉点。

UVa 10074 10502
10667

2.7.15.2 如果使用穷举法

最简单的方法就是用穷举法。矩形总共四个顶点，只要穷举所有可能的顶点位置，就可以找出答案来。纸的长宽为 H 和 W 的话，总共 $H * W$ 个位置可以放上顶点；要穷举所有矩形，时间复杂度就是 $O((H * W)^4)$ 。另外还要确定矩形内部有没有包含黑格子，时间复杂度就变成了 $O((H * W)^5)$ 。

要确定一个矩形的大小和位置，其实只要对角线的两个顶点就够了；要穷举所有矩形，时间复杂度是 $O((H * W)^2)$ 。确定矩形内部有没有包含黑格子，就是 $O((H * W)^3)$ 。

要确定一个矩形的大小和位置，也可以利用矩形左上角的顶点、长、宽；要穷举所有矩形，时间复杂度是 $O((H * W)^2)$ 。确定矩形内部有没有包含黑格子，就是 $O((H * W)^3)$ 。

谈了一堆简单的做法后，接着来试试 Dynamic Programming 吧！

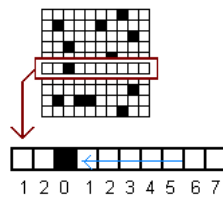
2.7.15.3 尝试切成条状，Divide and Conquer

因为原来的纸张又大又复杂，计算面积非常麻烦，所以我们可以试着把纸张切成小块小块，逐一处理。这里将纸张切成横条状（这个想法跟积分运算的道理是相同的），并套用上一篇文章所提到的 Largest Empty Interval 来计算每一条横条的面积；接着将所有横条合并起来，便能求出总面积。

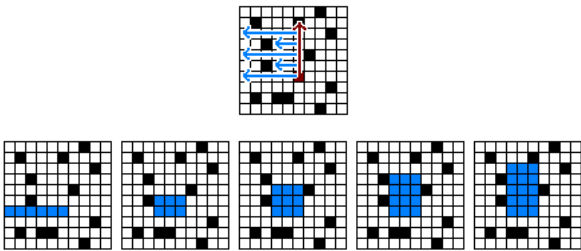
将纸张切成横条状，此即 Divide；每个横条用 Largest Empty Interval 来计算面积，此即 Conquer；将所有横条合并，此即 Combine。接着来看看要怎么找出 Largest Empty Rectangle 吧！

2.7.15.4 算法

首先将纸张切成横条状，针对每一横条，找出其中每个点往左可延伸的长度，即是在寻找 Largest Empty Interval。

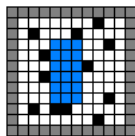


对纸张上的每个位置，都尝试作为矩形右下角的顶点位置（穷举所有矩形右下角的位置）。固定矩形右下角的顶点后，观察该处以上的每个横条（穷举所有矩形高度），往左可延伸的长度，便可以求得最大矩形面积。



2.7.15.5 程序代码

为了让边界计算不会溢位，于是将纸张的外面多围一圈。这是实作二维地图时很常用的方法。



```
1  bool array[10+2][10+2]; // 空敞处为 true, 有障碍物则
    为 false
```

2.7.15.6 程序代码

先设计出计算一个横条的程序代码——计算 Largest Empty Interval , 运用了 DP。(这段程序代码在计算 width 时, 每一格都会覆盖掉而不受旧值影响, 故重算时不必重新初始化。)

```
1  int width[10+2];
2  // 一个横条上, 每个位置往左可延伸的长度
3  // 如 width[2] 就是第二个位置往左可延伸的长度
4  // 初始化为零
5
6  void only_one_bar()
7  {
8      // 计算每个点往左可延伸的长度
9      for (int j=1; j<=10; ++j)
10         if (position j have blockade)
11             // 有障碍物, 长度为 0。
12             width[j] = 0;
13         else
14             // 没障碍物, 长度增加。
15             width[j] = width[j-1] + 1;
16 }
```

补足程序代码, 计算所有横条。

```
1  bool array[10+2][10+2];
2  int width[10+2][10+2];
3
4  void all_bars()
5  {
6      for (int i=1; i<=10; ++i)    // 计算每个横条
7          for (int j=1; j<=10; ++j)
```

```

8         if (array[i][j])
9             width[i][j] = width[i][j-1] + 1;
10        else
11            width[i][j] = 0;
12    }

```

2.7.15.7 程序代码

对纸张上的每个位置，都尝试作为矩形右下角的顶点位置。固定矩形右下角的顶点后，观察该处以上的每个横条，往左可延伸的长度，便可以求得最大矩形面积。先设计出计算一个位置的程序代码。

```

1  int only_one_point(int i, int j)
2  {
3      int area = 0;
4      int w = 1e9;
5
6      for (int h=1; i-h+1 >= 0; ++h)
7      {
8          // 已经窄到不能形成矩形了
9          if (width[i-h+1][j] == 0) break;
10
11         w = min(w, width[i-h+1][j]);
12         area = max(area, w*h);
13     }
14
15     return area;
16 }

```

判断矩形太窄的情形。

```

1  // (i,j)是矩形右下角顶点的位置
2  int only_one_point(int i, int j)
3  {
4      int area = 0;    // 最大矩形面积，先设为最小值。
5      int w = 1e9;    // 矩形的宽，设为无限大。
6
7      // 穷举矩形的高
8      for (int h=1; i-h+1 >= 0; ++h)

```

```
9      {
10          // 求出最窄处的宽度
11          w = min(w, width[i-h+1][j]);
12          // 最大矩形面积
13          area = max(area, w*h);
14      }
15
16      return area;
17 }
```

补足程序代码，穷举纸张上所有位置。

```
1  bool array[10+2][10+2];
2  int width[10+2][10+2];
3
4  int largest_empty_rectangle()
5  {
6      /* 计算所有横条当中，每个位置往左可延伸的长度。
7          */
8      for (int i=1; i<=10; ++i)
9          for (int j=1; j<=10; ++j)
10             if (array[i][j])
11                 width[i][j] = width[i][j-1] + 1;
12             else
13                 width[i][j] = 0;
14
15      /* 计算每个位置当作矩形右下角顶点时的最大矩形面
16          积。 */
17
18      // 最大矩形面积，初始化为最小值
19      int area = 0;
20
21      // 穷举矩形右下角顶点的位置
22      for (int i=1; i<=10; ++i)
23          for (int j=1; j<=10; ++j)
24              {
25                  int w = 1e9;
```

```

25         for (int h=1; i-h+1 >= 0; ++h)
26         {
27             if (width[i-h+1][j] == 0) break;
28
29             w = min(w, width[i-h+1][j]);
30             area = max(area, w*h);
31         }
32     }
33
34     return area;
35 }

```

2.7.15.8 复杂度

时间复杂度分析：首先计算了每个横条的 Largest Empty Interval，接着穷举矩形的右下角顶点位置，又穷举了矩形的各种高度，算出最大矩形面积。时间复杂度是 $O((H * W) * H)$ 。

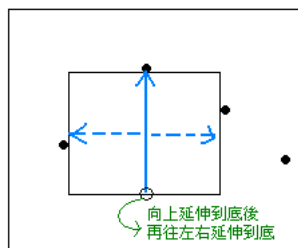
空间复杂度分析：储存全部问题的答案，空间复杂度是 $O(H * W)$ 。只想计算一个特定问题的答案，空间复杂度当然可以精简，这里就不多提了。

计算的方向是可以改变的。可以改为切直条，可以改为穷举矩形右上角顶点，道理都一样。

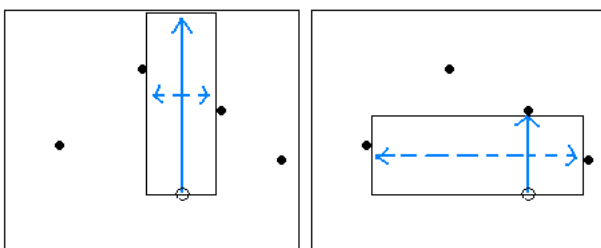
2.7.15.9 更好的方法

前面介绍的方法用了很多穷举，也重复计算了很多地方。所以，还可以更快。

这个方法是穷举纸张上每一个位置，每个位置都去计算以该点为长方形底部，往上延伸到底后，再往左右延伸到底的面积。



如果穷举一个横条上的所有位置，便可以得到以该横条为长方形底部的 Largest Empty Rectangle。



所以，只要穷举纸张上每个位置，就可以算出 Largest Empty Rectangle 了。

2.7.15.10 讨论

之前只将长方形往左延伸，故要穷举所有高度。现在改为同时往左右延伸，由于这种延伸方式可得到最大的矩形，便不必穷举所有高度。

2.7.15.11 时间复杂度

时间复杂度是 $O(H * W)$ 。

2.7.15.12 程序代码：Largest Empty Rectangle 的面积

计算过程满繁复的。大抵上和上一篇的方式差不多，我有点懒的说明，所以直接给程序代码吧。（懒散是不好的行为，请勿模仿。）

```

1  bool array[10+2][10+2];
2
3  int wl[10+2];    // 每一条横条往左可延伸的长度
4  int wr[10+2];    // 每一条横条往右可延伸的长度
5
6  int h[10+2];     // 矩形往上可延伸的高度
7  int l[10+2];     // 矩形往上延伸到底后，往左可延伸的
    距离。
8  int r[10+2];     // 矩形往上延伸到底后，往右可延伸的
    距离。
9
10 int largest_empty_rectangle()
11 {
12     // 最大矩形面积，初始化为最小值
13     int max_area = 0;
```



```
14
15     // 以每一个横条当作长方形底部
16     for (int i=1; i<=10; ++i)
17     {
18         // 往左可延伸的长度
19         for (int j=1; j<=10; ++j)
20             if (array[i][j])
21                 wl[j] = wl[j-1] + 1;
22             else
23                 wl[j] = 0;
24
25         // 往右可延伸的长度
26         for (int j=10; j>=1; --j)
27             if (array[i][j])
28                 wr[j] = wr[j+1] + 1;
29             else
30                 wr[j] = 0;
31
32         // 矩形往上可延伸的高度
33         for (int j=1; j<=10; ++j)
34             if (array[i][j])
35                 h[j] = h[j] + 1;
36             else
37                 h[j] = 0;
38
39         // 矩形往上延伸到底后，往左可延伸的距离。
40         for (int j=1; j<=10; ++j)
41             if (l[j] == 0)
42                 l[j] = wl[j];
43             else
44                 l[j] = min(wl[j], l[j]);
45
46         // 矩形往上延伸到底后，往右可延伸的距离。
47         for (int j=1; j<=10; ++j)
48             if (r[j] == 0)
49                 r[j] = wr[j];
```

```
50         else
51             r[j] = min(wr[j], r[j]);
52
53         // 计算 Largest Empty Rectangle 并纪录之
54         for (int j=1; j<=10; ++j)
55             max_area = max(max_area, (l[j] + r[j] -
56                             1) * h[j]);
57     }
58     return max_area;
59 }
```

2.7.15.13 程序代码: Largest Empty Rectangle 的位置

每当产生最大值之后, 就看看此时长方形往上、往左、往右可延伸的距离, 就能推敲出最大的长方形的位置。不过这种方式只能找出其中一个长方形的位置。

【待补程序代码】

2.7.15.14 最好的方法

最简洁的做法, 是利用 stack, 宛如判断括号对称一般, 将长方形的左右两边线找出来。特别要小心的地方, 是当 stack 的元素全部弹出之后, 之后出现的右边线还是有用处的, 不能把它想做是孤单的右括号。时间复杂度是 $O(H * W)$ 。

1. 切成直条。预先用 DP 计算每一条直线的 Empty Interval 高度。
2. 穷举每一个横条, 作为长方形的底线, 并利用 stack 算出最大矩形。

演练其中一段过程:

2.7.16 范例

Maximum Subarray
 1D p-Center Problem
 Longest Increasing Subsequence
 Longest Common Subsequence
 Longest Palindrome Substring
 0/1 Knapsack Problem

2.7.17 范例: Matrix Chain Multiplication

矩阵乘法具有结合率。在一连串的矩阵乘法中，可以从中任取两个相邻的矩阵相乘，先行结合成一个新矩阵，不会改变所有矩阵相乘之后的结果。

在一连串的矩阵乘法中，无论从何处开始相乘，计算结果都一样，然而计算时间却有差异。两个矩阵大小为 $a \times b$ 及 $b \times c$ ，其相乘需要 $O(a * b * c)$ 的时间（当然还可以更快，但是此处不讨论），那么一连串的矩阵乘法，需要多少时间呢？

从最后一次相乘的角度来看，原来的一连串矩阵，可以从最后一次相乘的地方分开，将原问题化作两串矩阵相乘。「Exponentiation」也有点类似，也是化作两个数字相乘；但是在本问题中，并非固定地对半分，而是同时考虑所有可能的分法。

$$f(i, k) = \min\{f(i, j) + f(j + 1, k) + r[i] * c[j] * c[k]\} \quad i \leq j < k$$

$f(i, k)$: 从第 i 个矩阵乘到第 k 个矩阵，最少的相乘次数。

$r[i]$: 第 i 个矩阵的 row 数目。

$c[i]$: 第 i 个矩阵的 column 数目。

```

1  int f[100][100];
2  int r[100], c[100];
3
4  void matrix_chain_multiplication()
5  {
6      memset(array, 0x7f, sizeof(array));
7      for (int i=0; i<N; ++i)
8          array[i][i] = 0;
9
10     for (int l=1; l<N; ++l)
```

```

11         for (int i=0; i+1<N; ++i)
12         {
13             int k = i + 1;
14             for (int j=i; j<k; ++j)
15                 f[i][k] = min(f[i][k], f[i][j] + f[
                        j+1][k] + r[i] * c[j] * c[k]);
16         }
17     }

```

计算顺序也可以调整成 online 版本。

```

1     for (int l=1; l<N; ++l)
2         for (int i=0; i+1<N; ++i)
3             for (int j=k-1; j>=i; --j)
4 //         for (int j=i; j<k; ++j)
5             f[i][k] = min(f[i][k], f[i][j] + f[
                        j+1][k] + r[i] * c[j] * c[k]);

```

当然也可以印出矩阵相乘的顺序。另外用一个数组，纪录最后一次的相乘位置就行了。

同是纪录区间的动态规划问题：

Matrix Chain Multiplication

Optimal Binary Search Tree

Optimal Alphabetic Code Tree

Minimum Weight Triangulation of Convex Polygon

Context-free Language: Cocke-Younger-Kasami Algorithm

现今已能在 $O(N \log N)$ 时间内解决 Matrix Chain Multiplication :
<http://historical.ncstrl.org/litesite-data/stan/CS-TR-81-875.pdf> 。

UVa 348 442 ICPC
6669

2.7.18 范例：文章换行

一大段的英文段落，适当的将文章换行，让文字不超过纸张边界，尽量美化排版，求得最佳的留白方式。

穷举行数，然后穷举一行挤入多少字数。

UVa 709 848 400

2.7.19 范例：Longest Increasing Subsequence

把解答编入状态之中。

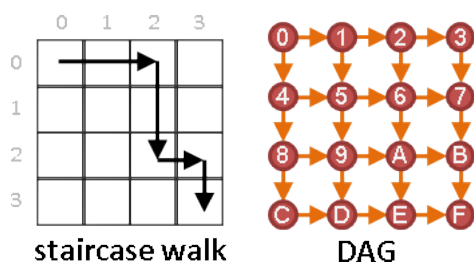
详见「Longest Increasing Subsequence」倒数两篇。

2.7.20 范例：鸡蛋耐力测试**2.7.21 范例：二进制数**

UVa 10934 882 ICPC
4554

2.7.22 范例：节省内存

ICPC 4833 5101

2.7.23 Dynamic Programming 的递归关系是 DAG

「DAG」是指图论的「有向无环图」。

DP 的各个子问题，就是 DAG 的每个点，一个子问题对应一个点。

DP 的计算顺序，就是 DAG 的拓扑顺序，一种计算顺序对应一种拓扑顺序。

DP 的递归关系不会循环——既然不会循环，显然是 DAG。

各种 DP 经典问题，诸如「Staircase Walk」、「Longest Increasing Subsequence」，各位读者可以试着改用 DAG 的观点来看待这些问题，试着改用 Graph Traversal 的观点设计算法。

ICPC 5104