20181025

第 1 章　绪论

🕮　1.1　计算

计算模型=计算机=信息处理工具
所谓算法，即特定计算模型下，旨在解决特定问题的指令序列。

算法特点：

1) 输入：待处理的信息(问题)
2) 输出：经处理的信息(答案)
3) 正确性：的确可以解决指定的问题
4) 确定性：任一算法都可以描述为一个由基本操作组成的序列
5) 可行性：每一基本操作都可实现，且在常数时间内完成
6) 有穷性：对于任何输入，经有穷次基本操作，都可以得到输出

……

有穷性举例：

$$Hailstone(n) = \begin{cases} \{1\}, & n \le 1 \\ \{n\} \cup Hailstone\left(\frac{n}{2}\right), & n\text{偶数} \\ \{n\} \cup Hailstone(3n+1), & n\text{奇数} \end{cases}$$

Hailstone Sequence：至今也没有证明对于任意 n，其序列是有穷或无穷。

什么是好算法：

1) 正确：符合语法，能够编译、链接
2) 健壮：能判别不合法的输入并做适当处理，而不致非正常退出
3) 可读：结构化代码，变量准确命名、注释…
4) 效率：速度尽可能快；存储空间尽可能少

Algorithms + Data Structure = Programs
(Algorithms + Data Structure) × Efficiency = Computation

Data Structure and Algorithms 简称 DSA，不同 DSA 性能有好坏优劣之别。
1) 引入理想、同一、分层次的尺度；
2) 运用该尺度，以测量 DSA 的性能。

🕮　1.2　计算模型

❁　1.2.1　问题规模

如果考察 $T_A(P)$ = 算法 A 求解问题实例 P 的计算成本，意义不大，因为可能的实例太多，如何归纳概括？使用划分等价类。
问题实例的规模，往往是决定计算成本的主要因素。

通常：规模接近，计算成本也接近；规模扩大，计算成本亦上升。

❁　1.2.2　最坏情况

令$T_A(n)$ = 算法 A 求解某一问题规模为n的实例，所需的计算成本。

讨论特定算法 A 时，简记为$T(n)$。
然而这样的定义仍有问题。因为对于同一问题，等规模的不同实例，计算成本不尽相同，甚至可能有实质性差别。

如平面上n个点，找到所形成三角形面积最小的三个点。以蛮力算法为例，最坏情况需要枚举所以C_n^3种组合，但运气好的话，一上来就找到。

因此，稳妥起见，取$T(n) = \max\{T(P) \mid |P| = n\}$
也就是在规模同为n的所有实例中，只关注**最坏**(成本最高)情况。

1.2.3 理想模型

同一个问题通常有多种算法，如何判断它们的优劣呢?
实验统计是最直接的方法，但不足够准确反映算法真正效率。
1) 不同算法，可能更适应于不同**规模/类型**的输入
2) 同一算法，可能由不同**程序员**、不同**编程语言**、经不同**编译器**实现；可能实现并运行于不同的**体系结构**、**操作系统**...

为了给出**客观**的评价，必须抽象出一个**理想**的模型，不依赖于上述各种因素，从而直接准确地描述、测量、评价算法。

1.2.4 图灵机模型

图灵机 TM (Turing Machine)
(1) Tape (纸带)：依次均匀地划分为单元格，各注有某一字符，默认为'#';
(2) Alphabet (字母表)：字符的种类有限
(3) Head (读写头)：总是对准某一单元格，并可读取和改写其中的字符，每经过一个节拍，可转向左侧或右侧的邻格
(4) State (状态表)：TM 总是处于有限种状态的某一种，每经过一个节拍，可(按照规则)转向另一种状态;

转换函数(Transition Function)：(q, c; d, L/R, p)
若当前状态为 q 且字符为 c，则将当前字符改写为 d；转向左侧/右侧的邻格，转入 p 状态；一旦转入特定的状态'h'，则停机。

实例：TM : Increase
功能：将二进制非负整数加一
算法：全'1'的后缀翻转为全'0'，原最低位的'0'或'#'翻转为'1'。

1.2.5 RAM 模型

RAM (Random Access Machine)与图灵机类似，都假设**无限空间**。
寄存器顺序编号，总数没有限制：R[0], R[1], R[2], R[3], …

每一基本操作只需常数时间：

R[i] <- c	R[i] <- R[j]
R[i] <- R[R[j]]	R[R[i]] <- R[j]
R[i] <- R[j] + R[k]	R[i] <- R[j] - R[k]
IF R[i] = 0 GOTO l	IF R[i] > 0 GOTO l
GOTO l	STOP

与 TM 模型一样，RAM 模型也是一般计算工具的简化与抽象，独立于具体平台，对算法的效率做出可信的比较与评判。

因为在这些模型中做了转换：
算法的运行时间 $\propto$ 算法需要执行的**基本操作次数**
$T(n) =$ 算法为求解规模为n的问题所需执行基本操作次数

实例：RAM : Floor
功能：向下取整的除法
对于$c \geq 0, d > 0$：

$$c/d = \max\{x \mid d \cdot x \leq c\} = \max\{x \mid d \cdot x < c + 1\}$$

算法：反复从 R[0]=1+c 中减去 R[1]=d，统计在下溢之前，所做减法次数x。

```
0    R[3] <- 1                r3 = 1
1    R[0] <- R[0] + R[3]      r0++
2    R[0] <- R[0] - R[1]      r0 -= r1
3    R[2] <- R[2] + R[3]      r2++
4    IF R[0] > 0 GOTO 2       if r0 > 0 goto 2
5    R[0] <- R[2] - R[3]      else r2-- and r0 = r2
6    STOP                     return r0
```

执行此过程，可以记录为一张表，表的行数就是执行基本指令的总次数。

🕮 1.3 大 O 记号

随着问题规模增长，计算成本如何增长?
这里更关心足够大的问题，注重考察成本的增长趋势。
渐进分析(asymptotic analysis)：当$n \gg 2$后，对于规模为 n 的问题，计算成本：
需执行的基本操作次数：$T(n) =$?
需占用的存储单元数：$S(n) =$?

1.3.1 大 O 记号 (big-O notation)

$T(n) = O(f(n))$ if $\exists c > 0$，当 n≫2 后，有$T(n) < c \cdot f(n)$

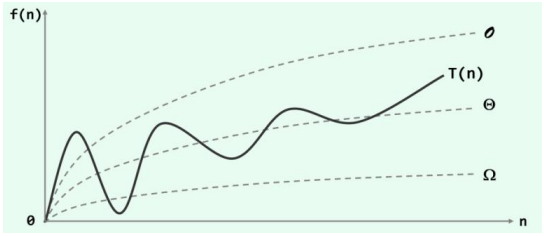
如$T(n) = \sqrt{5n \cdot [3n \cdot (n+2) + 4] + 6} < \sqrt{5n \cdot [6n^2 + 4] + 6} < \sqrt{35n^3 + 6}$

$< 6 \cdot n^{1.5} = O(n^{1.5})$

与$T(n)$相比，$f(n)$更为简洁，但依然能反映增长趋势
1) 常系数可以忽略：$O(f(n)) = O(c \cdot f(n))$
2) 低次项可以忽略：$O(n^3 + n^2) = O(n^3)$
考虑问题要**长远**(n足够大)、**主流**(常系数、低次项等非主流可以忽略)。

big-O 是T(n)的上界(upper bound)，可以认为 big-O 是对T(n)的悲观估计。

其他符号：
1) $\mathrm{T}(n)=\Omega(f(n))$：$\exists c>0$，当 n≫2 后，有$\mathrm{T}(n)>c\cdot f(n)$
也就是Ω是T(n)的下界。
2) $\mathrm{T}(n)=\Theta(f(n))$：$\exists c_1>c_2>0$，当 n≫2 后，有$c_1\cdot f(n)>\mathrm{T}(n)>c_2\cdot f(n)$
T(n)能被同一函数经过c_1,c_2放大倍后从上下两方界定。
认为Θ构成T(n)的确界。



❁ 1.3.2 高效解

$O(1)$：常数(constant function)

$$1=2018=2018\times 2018=O(1)$$

这类算法的效率最高。
不含转向(循环、调用、递归等)必顺序执行，即是$O(1)$

$O(\log n)$：对数
不写底数，因为常底数无所谓。
这类算法非常高效，复杂度无限接近于常数$O(1)$

❁ 1.3.3 有效解

$O(n^k)$：多项式 (polynomial function)

$$a_k n^k+a_{k-1}n^{k-1}+\cdots+a_1 n+a_0=O(n^k),a_k>0$$

$O(n)$：线性 (linear function)
很多编程习题主要在$O(n)$~$O(n^2)$之间。

这类算法的效率通常认为已令人满意，然而这个标准是不是太低了?
凡多项式复杂度，都认为可解，而不是难解。

❁ 1.3.4 难解

$O(2^n)$：指数 (exponential function)

$$\forall c>1,n^c=O(2^n)$$
$$n^{1000}=O(1.0000001^n)=O(2^n)$$
$$1.0000001^n=\Omega(n^{1000})$$

这类算法的计算成本增长极快，通常被认为不可忍受。
从$O(n^c)$到$O(2^n)$是从有效算法到无效算法的分水岭。
很多问题的$O(2^n)$算法往往显而易见，但设计出$O(n^c)$算法却很难。

例：S 包含n个正整数，$\sum S = 2m$。S 是否存在子集 T，满足$\sum T = m$?
直觉算法：逐一枚举 S 的每一子集，统计其中元素总和
然而需要迭代轮数：

$$\sum_{i=0}^{n} C_n^i = (1+1)^n = 2^n$$

该算法时间复杂度为$O(2^n)$，不好。

`2-Subset is NP-complete`：就目前的计算模型而言，不存在可在多项式时间内回答此问题的算法。也就是上面的直觉算法已属最优。

🕮 1.4 算法分析

❁ 1.4.1 算法分析主要任务

两个主要任务 = 正确性(不变性×单调性) + 复杂度

C++等高级语言的基本指令，均等效于常数条 RAM 基本指令；在渐近意义下，二者大体相当。
1) 分支转向：goto // 算法灵魂；出于结构化考虑被隐藏了
2) 循环迭代：for, while, ... // 本质上是: if + goto
3) 调用 + 递归 // 本质上也是 goto

复杂度分析主要方法：
1) 迭代：级数求和
2) 递归：递归跟踪 + 递归方程
3) 猜测 + 验证

❁ 1.4.2 级数

① 算术级数：与末项平方同阶

$$\mathrm{T}(n) = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} = \mathrm{O}(n^2)$$

② 幂方级数：比幂次高出一阶

$$\mathrm{T}_d(n) = 1 + 2^d + \cdots + n^d = \sum_{k=0}^{n} k^d \approx \int_0^n x^{d+1} dx = \frac{1}{d+1} n^{d+1} = O(n^{d+1})$$

③ 几何级数($a > 1$)：与末项同阶

$$\mathrm{T}_a(n) = a^0 + a^1 + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1} = O(a^n)$$

④ 收敛级数

$$\frac{1}{1\times 2} + \frac{1}{2\times 3} + \frac{1}{3\times 4} + \cdots \frac{1}{(n-1)n} = 1 - \frac{1}{n} = O(1)$$

$$1 + \frac{1}{2^2} + \cdots + \frac{1}{n^2} = O(1)$$

⑤ 级数未必收敛，但长度有限
调和级数与对数级数：

$$h(n) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \Theta(\log n)$$

$$\log 1 + \log 2 + \cdots + \log n = \log(n!) = \Theta(n \log n)$$

❀ 1.4.3 实例

① 取非极端元素

给定整数子集 S，$|S| = n \geq 3$，找出元素$a \in S, a \neq \max(S)\ and\ \ a \neq \min(S)$

算法：1) 从 S 中任取 3 个元素$\{x, y, z\}$
2) 确定并排除其中最大、最小值
3) 输出剩下的元素

无论输入规模n多大，上述算法需要执行时间不变：

$$T(n) = c = O(1) = \Omega(1) = \Theta(1)$$

② 起泡排序

给定n个整数，按非降序排列。

有序序列，任意一对相邻元素顺序；无序序列，总有一对相邻元素逆序

扫描交换：依次比较每一对相邻元素，如有必要，交换之。若一趟扫描都没有交换，排序完成；否则，进行下一趟扫描交换。

冒泡排序 Python 实现：

```python
def bubble_sort(arr):
    for i in range(len(arr)-1, 0, -1):
        exchange = False
        for j in range(i):  # 一趟扫描交换
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                exchange = True
        if not exchange:  # 如果一趟扫描没有一次交换，说明有序，直接结束
            return
```

最坏时间复杂度为$O(n^2)$。

❀ 1.4.4 算法的正确性证明

不变性：经过 k 轮扫描交换后，最大的 k 个元素必然就位

单调性：经过 k 轮扫描交换后，问题的规模缩减至 n - k

正确性：经至多 n 趟扫描后，算法必然终止，且能给出正确解答

⚙ 1.4.5 封底估算 (Back-Of-The-Envelope Calculation)

时间量的概念：

1 天 $= 24\text{hr} \times 60\text{min} \times 60\text{sec} \approx 25 \times 4000 = 10^5\text{sec}$

1 生 $\approx$ 1 世纪$= 100\text{yr} \times 365 \approx 3 \times 10^4\text{day} = 3 \times 10^9\text{sec}$

为祖国健康工作五十年 $= 1.6 \times 10^9\text{sec}$

三生三世 $= 300\text{yr} = 10^{10} sec$

宇宙大爆炸至今 $= 10^{21} sec = 10 \times (10^{10})^2\text{sec}$

例：考察对全国人口普查数据的排序$n = 10^9$

可从**硬件**机器上做改进和**算法**上做改进：

算法 \ 硬件	普通 PC 1GHz 10^9flops	天河 1A 10^{15}flops
bubble sort 10^{18}	10^9sec/30yr	10^3sec/20min
merge sort 30×10^9	30sec	3×10^{-5}sec/0.03ms

归并排序比冒泡排序效率提高大于3×10^7；天河 1A 比普通 PC 效率提高10^6。

普通 PC 使用归并排序比天河 1A 使用冒泡排序快得多。

可见，算法改进的威力十分巨大。

20181026

🕮 1.5 迭代与递归

Someone said: "To iterate is human, to recurse, divine." 迭代乃人工，递归方神通

然而递归的效率不如迭代。

例：数组求和 (迭代)

问题：计算任意 n 个整数之和

实现：逐一取出每个元素，累加之

```
int my_sum(int arr[], int n){
    int s = 0; // O(1)
    for (int i = 0; i < n; i++){ // O(n)
        s += arr[i]; // O(1)
    }
    return s; // O(1)
}
```

无论 arr 内容如何，都有：

$$T(n) = 1 + n \times 1 + 1 = n + 2 = O(n) = \Omega(n) = \Theta(n)$$

空间复杂度通常认为是除去输入本身所占空间外用于计算所需要的额外空间。

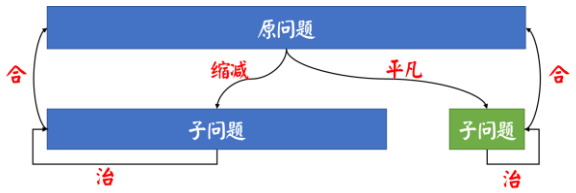
因此此处空间复杂度为$O(1)$。

⚙ 1.5.1 减而治之(decrease and conquer)

为求解一个大规模问题，可以将其划分为两个子问题：

其中一个是退化的平凡情况，另一规模缩减，但形式与原问题一样。

分别求解子问题，由子问题的解得到原问题的解。



例：数组求和 (线性递归)

```
int sum(int arr[], int n){
    return (n < 1) ? 0 : sum(arr, n-1) + arr[n-1];
}
```

递归跟踪(recursion trace)分析
检查每个递归实例，累计所需时间(调用语句本身，计入对应子实例)，其总和即算法执行时间。
递归调用，从前往后，值返回从后往前：

$$\text{main}() \to \text{sum}(\text{arr}, \text{n}) \to \text{sum}(\text{arr}, \text{n}-1) \to \cdots \to \text{sum}(\text{arr}, 1) \to \text{sum}(\text{arr}, 0)$$

本例单个递归只需O(1)时间，$\text{T}(n) = \text{O}(1) \times (n+1) = \text{O}(n)$

递归跟踪直观形象，但仅适用于简明的递归模式。对于复杂的递归模式，需要使用递推方程(recurrence)，其特点是间接抽象。

从递推角度看，为求解 sum(arr, n)需：
1) 递归求解规模为 n-1 的问题 sum(arr, n-1) // T(n-1)
2) 累加上 arr[n-1] // O(1)
3) 递归基：sum(arr, 0) // O(1)
递推方程：$\text{T}(n) = \text{T}(n-1) + \text{O}(1)$ // recurrence
递归基：　$\text{T}(0) = \text{O}(1)$ // base

$$\text{T}(n) = \text{T}(n-1) + \text{O}(1) = \text{T}(n-2) + 2 \times \text{O}(1) = \cdots$$
$$= \text{T}(0) + n \times \text{O}(1) = (n+1)\text{O}(1) = \text{O}(n)$$

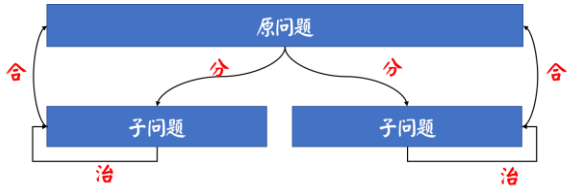
例：数组倒置

```
def reverse(arr, low, high): # 递归
    if low < high:  # 问题规模的奇偶性不变，需要两个递归基
        arr[low], arr[high] = arr[high], arr[low]
        reverse(arr, low+1, high-1)

def reverse1(arr, low, high): # 迭代
    while low < high:
        arr[low], arr[high] = arr[high], arr[low]
        low+=1
        high-=1
```

1.5.2 分而治之(divide and conquer)

为求解一个大规模问题，可以将其划分为若干(通常两个)子问题，规模大体相当，分别求解子问题，由子问题的解，得到原问题的解。



例：数组求和(二分递归)

```
def my_sum(arr, low, high):  # 区间范围 arr[low, high]
    if low == high:
```

```
        return arr[low]
    mid = (low + high) // 2
    return my_sum(arr, low, mid) + my_sum(arr, mid+1, high)
```

递归跟踪：$T(n)$ =各层递归实例所需时间之和

$= O(1) \times (2^0 + 2^1 + \cdots + 2^{\log n}) = O(1) \times (2^{\log n+1} - 1) = O(n)$

递推方程：从递推角度看，为求解 sum(arr, low, high)需：

1) 递归求解 sum(arr, low, mid)和 sum(arr, mid+1, high) // 2T(n/2)

2) 将子问题的解累加 // O(1)

3) 递归基：sum(arr, low, low) // O(1)

递推关系：$T(n) = 2 \times T(n/2) + O(1)$

递归基：$T(1) = O(1)$

$T(n) = k \times T(1) + (2^k - 1)O(1), k = \log_2 n$

故$T(n) = (\log n + (n-1))O(1) = O(n)$

例：Max2

从数组区间 arr[low, high)找出最大两个整数 arr[a]和 arr[b]

元素比较的次数要求尽可能少

```
def max2(arr, low, high):
    a, b = low, low
    # 扫描 arr[low, high)找出最大值 arr[a]
    for i in range(low+1, high):
        if arr[i] > arr[a]:
            a = i
    # 扫描 arr[low, a)
    for i in range(low+1, a):
        if arr[i] > arr[b]:
            b = i
    # 再扫描 arr(a, high), 找出次大值 arr[b]
    for i in range(a+1, high):
        if arr[i] > arr[b]:
            b = i
    return a, b

if __name__ == '__main__':
    arr = [randint(-10, 99) for x in range(10)]
    print(arr)
    print(max2(arr, 0, len(arr)))
```

结果：

```
[13, 99, 53, 97, -8, 83, 57, 58, 26, -4]
(1, 3)
```

无论如何，比较次数总是$\Theta(2n-3)$(最好最坏情况一样)

```
def max2_2(arr, low, high):
    a, b = low, low+1
    # 起始 arr[a]为前两个较大者,arr[b]为较小者
    if arr[b] > arr[a]:
        a, b = b, a
    for i in range(low+2, high):
        # 若发现元素比 arr[b]先给 b, 再与 arr[a]比较
        if arr[i] > arr[b]:
```

```
            b = i
            if (arr[b] > arr[a]):
                a, b = b, a
    return a, b

if __name__ == '__main__':
    arr = [randint(-10, 99) for x in range(10)]
    print(arr)
    a, b = max2_2(arr, 0, len(arr))
    print(f'最大:arr[{a}]={arr[a]}, 次大:arr[{b}]={arr[b]}')
```

结果：

```
[50, 69, 36, 95, 20, 25, 1, 46, 61, 54]
最大:arr[3]=95, 次大:arr[1]=69
```

最好情况(循环一次 if)：$1+(n-2)\times 1=n-1$

最坏情况(循环两次 if)：$1+(n-2)\times 2=2n-3$

最坏情况并没有实质改进。

二分递归：

```
def max2_3(arr, low, high):
    a, b = (low, low+1) if arr[low] > arr[low+1] else (low+1, low)
    if low+2 == high:  # T(2) = 1
        return a, b
    if low+3 == high:  # T(3) <= 3
        if arr[low+2] > arr[a]:
            a, b = low+2, a
        elif arr[low+2] > arr[b]:
            b = low+2
        return a, b

    mid = (low+high)//2
    la, lb = max2_3(arr, low, mid)  # 左边最大两个
    ra, rb = max2_3(arr, mid, high)  # 右边最大两个
    if arr[la] > arr[ra]:  # 左边最大>右边最大
        return la, lb if arr[lb] > arr[ra] else ra
    return ra, rb if arr[rb] > arr[la] else la
```

$\mathrm{T}(n)=2\times\mathrm{T}(n/2)+2\le 5n/3-2$

20181027

🕮 1.6 动态规划

Make it work, make it right, make it fast. - Kent Beck

从某种意义上说，动态规划是：用递归找出算法的本质，给出初步解后，再将其有效地转为迭代的形式。

例：fib()

$\mathrm{fib}(n)=\mathrm{fib}(n-1)+\mathrm{fib}(n-2):\{0,1,1,2,3,5,8,\dots\}$

① 递归

```
def fib(n): # 递归
    return n if n < 2 else fib(n-1)+fib(n-2)
```

使用 Python 运行，在 n=30 左右开始变得肉眼可见的慢...

复杂度：$\mathrm{T}(0)=\mathrm{T}(1)=1;\ \mathrm{T}(n)=T(n-1)+\mathrm{T}(n-2)+1, n>1$

令$\mathrm{S}(n)=[\mathrm{T}(n)+1]/2$

则$S(0)=1=\text{fib}(1), S(1)=1=\text{fib}(2)$

$2S(n)-1=2S(n-1)-1+2S(n-2)-1+1$

即：$S(n)=S(n-1)+S(n-2)=\text{fib}(n+1)$

所以：$T(n)=2S(n)-1=2\text{fib}(n+1)-1=O(\text{fib}(n+1))$

$$\text{fib}(n)\approx\Phi^n,\Phi=\frac{1+\sqrt{5}}{2}\approx 1.61803$$

∴ $T(n)=O(\Phi^n)=O(2^n)$

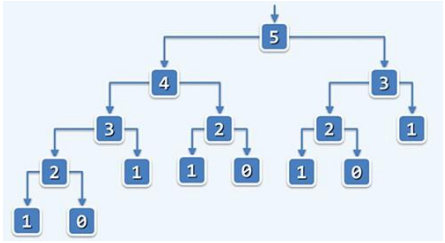
封底估算：$\Phi^{36}\approx 2^{25},\Phi^{43}\approx 2^{30}\approx 10^9\text{flo}\approx 1\text{sec}$

$\Phi^5\approx 10,\Phi^{67}\approx 10^{14}\text{flo}=10^5\text{sec}\approx 1\text{day}$

$\Phi^{92}\approx 10^{19}\text{flo}=10^{10}\text{sec}\approx 10^5\text{day}\approx 3\text{century}$

这说明该算法不好，严格说它甚至不是一个算法。

递归版 fib()低效的根源：各递归实例均被大量重复地调用。



② 迭代

1) 记忆(memoization)

将已计算的实例结果制表(全局数组)备查。

2) 动态规划(dynamic programming)

颠倒计算方向：由自顶而下递归，变为自底而上迭代

```
def fib_2(n):  # 迭代
    a, b = 0, 1
    while n > 0:
        a, b = b, a+b  # a, b 交替滚动向前走
        n -= 1
    return a
```

复杂度$T(n)=O(n)$，且只需$O(1)$空间。

例：最长公共子序列 LCS

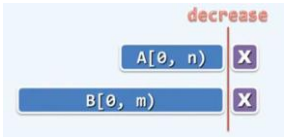
子序列(subsequence)：由序列中若干字符，按原相对次序构成

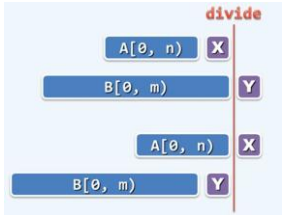
最长公共子序列(Longest Common Subsequence)：两个序列公共子序列中的最长者。(1) 可能有多个：如"education"和"advantage"的 LCS 可以是"data"或"dana"；(2) 可能有歧义：如"didactical"和"advantage"的 LCS 是"data"，但其中"d"是来自"didactical"第 1 个"d"还是第 2 个"d"?

简单版本为：只需计算 LCS 的长度

① 递归

对于序列 A[0, n]和 B[0, m]，LCS(A, B)有 3 种情况：
1) n = -1 或 m = -1，某个序列为空序列，则结果为空序列"" //递归基
2) 若 A[n]=B[m]='X'，则取 LCS(A[0, n), B[0, m))+'X' //减而治之
3) 若 A[n]≠B[m]，则在 LCS(A[0, n], B[0, m))与 LCS(A[0, n), B[0, m])取更长者






```python
def lcs(a, b):
    def lcs_rec(a, b, n, m):
        if n < 0 or m < 0:
            return 0
        if a[n] == b[m]:
            return lcs_rec(a, b, n-1, m-1)+1 # 最后字符相等，减而治之
        return max(lcs_rec(a, b, n, m-1), lcs_rec(a, b, n-1, m)) # 分而治之
    return lcs_rec(a, b, len(a)-1, len(b)-1)

if __name__ == '__main__':
    a = 'educational'
    b = 'advantage'
    print(lcs(a, b))  # 4
```

单调性：无论如何，每经一次比较，原问题的规模必可减小
具体地，作为输入的两个序列，至少其一的长度缩短一个单位

最好情况(只出现减而治之不出现分而治之)，只需$O(n+m)$时间
但问题在于，一旦有分而治之的情况出现，原问题将分解为两个子问题
更糟的是，它们的规模之和是原来的两倍，而且在随后进一步导出的子问题，也有类似情况，造成大量的雷同。

最坏情况：LCS(A[0, a], B[0, b])出现次数为：

$$C_{n+m-a-b}^{n-a} = C_{n+m-a-b}^{m-b}$$

特别地，LCS(A[0], B[0])的次数可多达：

$$C_{n+m}^{n} = C_{n+m}^{m}$$

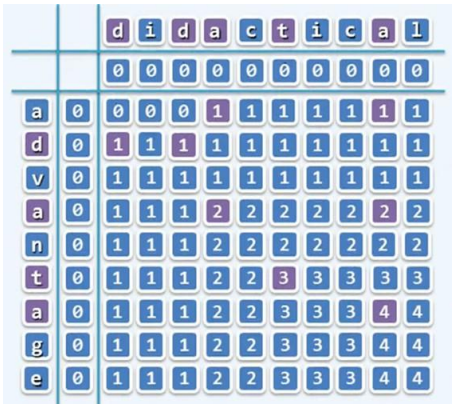
当$n = m$时，复杂度为$O(2^n)$。

② 迭代
与 fib()类似，LCS 的递归也有大量重复的递归实例(子问题)
最坏情况，共计出现$O(2^n)$个。

各子问题，分别对应于 A 和 B 的某个前缀组合，因此总数不过是$n \times m$种。
采用动态规划的策略，只需$O(n \times m)$时间即可计算出所有子问题。

1) 将所有子问题(假想地)列成一张表
2) 颠倒计算方向，从 LCS(A[0], B[0])出发，依次计算出所有项。

$$C[i][j]=\begin{cases}0 & i=0 \text{ or } j=0\\ C[i-1][j-1]+1 & i,j>0, A[i]=B[j]\\ \max\{C[i][j-1],C[i-1][j]\} & i,j>0, A[i]\neq B[j]\end{cases}$$




```python
def lcs_2(a, b):
    n, m = len(a), len(b)
    # 子问题假想表格(n+1)(m+1), 初始全部填 0
    c = [[0 for j in range(m+1)] for i in range(n+1)]
    # 从上到下从左往右遍历表格
    for i in range(1, n+1):
        for j in range(1, m+1):
            if a[i-1] == b[j-1]:  # 减而治之,左上角+1
                c[i][j] = c[i-1][j-1]+1
            # 分而治之,左边或上面较大者
            else:
                c[i][j] = max(c[i][j-1], c[i-1][j])
    # 表格右下角即为最终结果
    return c[n][m]

if __name__ == '__main__':
    a = 'hoshizora rin'
    b = 'nishikino maki'
    print(lcs(a, b))  # 6
    print(lcs_2(a, b))  # 6
```

总结：利用递归设计出可行且正确的解；在此基础上使用动态规划消除重复计算，提高效率。

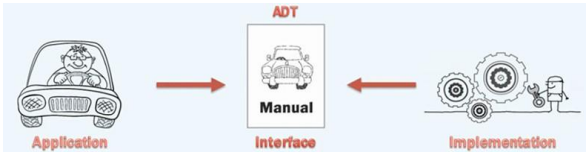
20181030

第 2 章　向量

🕮　2.1　接口与实现

抽象数据类型(Abstract Date Type)：数据模型+定义在该模型上的一组操作

数据结构(Data Structure)：基于某种特定语言，实现 ADT 的一整套算法



用户只关心提供的功能；实现者负责具体内部实现，说明书就是实现者与用户之间达成的协议、规范。

最基本的线性结构统称为序列(sequence)，根据数据项的逻辑次序与物理存储地址对应关系不同，分为向量(vector)和列表(list)。

C/C++语言，数组 A[]中的元素与[0, *n*)的编号一一对应。第 *n* 号元素虽然不一定存在，此处虚拟地放在后面，作为哨兵。

反之，每个元素均可由编号唯一指代，并可直接访问：

$A[i]$的物理地址$= A + i \times s$, s为单个元素所占空间

所以也称为线性数组(linear array)。

向量(vector)是 C++等高级语言中数组这种数据结构的推广和泛化，由一组元素按线性次序封装而成：

(1) 各元素与[0, n)内的秩(rank, 就是索引)一一对应 //循秩访问(call-by-rank)
(2) 元素的**类型**不限于基本类型
(3) 操作、管理、维护更加简化、同一、安全
(4) 更为方便地参与复杂数据结构的定制与实现

按照 ADT 的规范，向量结构必须提供一系列的操作接口：

操作	**功能**
size()	元素总数
get(r)	获取 r 号元素
put(r, e)	r 号元素替换为 e
insert(r, e)	r 号位插入 e，后继元素后移
remove(r)	删除并返回 r 号元素，后继元素前移
disordered()	判断是否有序(非降序)，返回逆序对数目
sort()	排序(非降序)
find(e)	查找目标元素 e，返回 rank，不存在返回-1
search(e)	查找目标元素 e，返回不大于 e 且 rank 最大的元素(有序向量)
deduplicate()	删除重复元素
uniquify()	删除重复元素(有序向量)
traverse()	遍历向量并统一处理所有元素，处理方法由函数对象指定

设计接口时，暂时不关系具体实现方法，只关心操作语义。

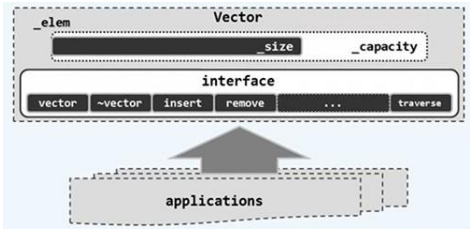
① Vector 模板类：

```
typedef int Rank; // 秩
#define DEFAULT_CAPACITY 3 // 默认初始容量(实际应用可设更大)
template <typename T> class Vector{ // 向量模板类
    private:
        Rank _size; // 元素个数
        int _capacity; // 容量
        T* _elem; // 数据区
    protected:
        // 内部函数
    public:
        // 构造函数
        // 析构函数
```

```
        // 只读接口
        // 可写接口
        // 遍历接口
};
```

定义一个元素类型为 T 的 Vector 模板类(//泛型?)。

Vector 类被封装，对用户提供一些接口。用户通过接口使用，而不需要了解内部实现：



② 构造函数和析构函数：

```
    public:
        // 构造函数
        Vector(int c=DEFAULT_CAPACITY){
            _elem = new T[_capacity=c]; // 申请长度为 c 类型为 T 的一段连续空间
            _size = 0; // 元素个数为 0
        }
        Vector(T const * A, Rank low, Rank high){ // 数组区间复制
            copyFrom(A, low, high); // 需要实现
        }
        Vector(Vector<T> const& V, Rank low, Rank high){ // 向量区间复制
            copyFrom(V._elem, low, high);
        }
        Vector(Vector<T> const& V){ // 向量整体复制
            copyFrom(V._elem, 0, V._size);
        }
        // 析构函数
        ~Vector(){ // 释放内部空间
            delete [] _elem;
        }
```

③ copyFrom()函数

//写在类外部，内部需要声明，该函数不是外部接口，可设为 protected

```
template <typename T> // 元素类型(T 为基本类型或已重载赋值操作符=)
void Vector<T>::copyFrom(T const * A, Rank low, Rank high){
    _elem = new T[_capacity=2*(high-low)]; // 分配 2 倍空间
    _size = 0;
    while (low < high){ // 将 A[low, high)元素逐一复制到_elem[0, high-low)
        _elem[_size++] = A[low++];
    }
}
```

🕮 2.2 可扩充向量

❀ 2.2.1 静态空间管理

开辟内部数据_elem[]并使用一段地址连续的物理空间。

_capacity：总容量；_size：当前元素个数

若采用静态空间管理策略，容量_capacity 固定，则有明显不足：
1) 上溢(overflow)：_elem[]不足以存放所有元素，尽管系统有足够空间
2) 下溢(underflow)：_elem[]中的元素寥寥无几，造成空间浪费
更糟的是，一般很难准确预测空间的需求量。

装填因子(load factor)：λ=_size / _capacity，是衡量空间利用率的重要指标。
也就是需要保证向量的装填因子既不超过 1，也不太接近于 0。

❀ 2.2.2 动态空间管理

蝉的哲学：身体每经过一段时间，以致无法为外壳容纳，即蜕去原先的外壳，代之以一个新的外壳。
同理，对于向量，在即将发生上溢时，适当地扩大内部数组的容量，使之足以容纳新的元素，即可扩充向量(extendable vector)。

扩容算法实现：expand()函数

```
template <typename T>
void Vector<T>::expand(){ // 向量空间不足时扩容
    if (_size < _capacity)
        return; // 未满不必扩容
    _capacity = max(_capacity, DEFAULT_CAPACITY); // 不低于最小容量
    T* oldElem = _elem; // 备份
    _elem = new T[_capacity <<= 1]; // 容量加倍
    for (int i = 0; i < _size; i++){ // 复制原来内容
        _elem[i] = oldElem[i];
    }
    delete [] oldElem; // 释放原来空间
}
```

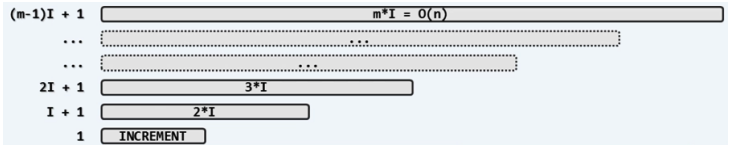
扩容后数据区地址由操作系统分配，与原来没有直接关系。此时，若直接引用数组，往往会导致共同指向原数组的其它指针失效，成为野指针(wild pointer)；而封装为向量后，可继准确地引用各元素，从而有效避免了野指针的风险。

❀ 2.2.3 扩容策略

① 容量递增策略

```
    T* oldElem = _elem;
    _elem = new T[_capacity += INCREMENT]; // 追加固定大小的容量
```

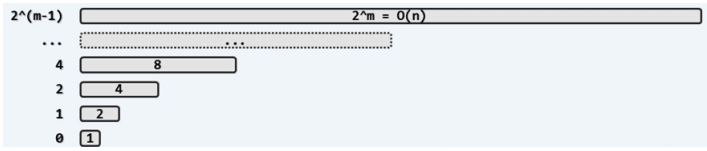
最坏情况在初始容量为 0 的空向量中，连续插入$n = m \times I$ (>>2)个元素：
在第 1、I + 1、2I + 1、3I + 1...次加入时都需要扩容。
即使不计申请空间操作，各次扩容过程复制原向量的时间成本依次为：
0、I、2I、3I...(m-1)I　//算术级数
总耗时$= I \times (m-1)/2 = O(n^2)$，每次扩容的分摊成本为$O(n)$。



② 容量加倍策略

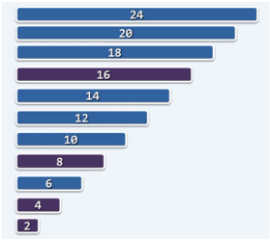
```
    T* oldElem = _elem; // 备份
    _elem = new T[_capacity <<= 1]; // 容量加倍
```

最坏情况：在初始容量为 1 的满向量中，连续插入$n = 2^m$ (>>2)个元素：
在第 1、2、4、8、16...次插入时都需要扩容。
各次扩容过程复制原向量的时间成本依次为：
1、2、4、8、16... 2^m //几何级数
总耗时$= O(n)$，每次扩容的分摊成本$O(1)$。



两种策略对比：

	递增策略	倍增策略
累计扩容时间	$O(n^2)$	$O(n)$
分摊扩容时间	$O(n)$	$O(1)$
装填因子	$\approx 100\%$	$> 50\%$



倍增策略在空间效率上做出适当牺牲，换取时间方面的巨大收益。

⚽ 2.2.4 分摊分析

平均复杂度(average/expected complexity)
根据数据结构各种操作出现概率的分布，将对应的成本加权平均。
各种可能的操作作为**独立事件**分别考察，割裂了操作之间的相关性和连贯性，往往不能准确评判数据结构和算法的真实性能。

分摊复杂度(amortized complexity)
对数据结构**连续**地实施**足够多**次操作，所需的总体成本分摊至单次操作。
从**实际可行**的角度，对一系列操作做整体考量，更加真实刻画了可能出现的操作序列，可以更精准地评判数据结构和算法的真实性能。

// 动态缩容算法实现：shrink()

20181031

🕮 2.3 无序向量

⚽ 2.3.1 元素访问

通过 get(r)和 put(r, e)接口可以读写向量元素；但就便捷性而言，远不如数组元素访问形式：A[r]。为了沿用下标访问，需要重载下标操作符[]：

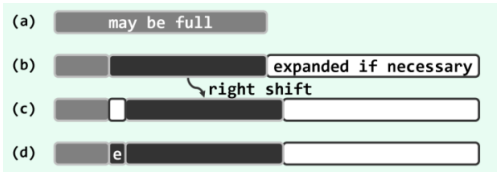
```
template <typename T>
T &Vector<T>::operator[](Rank r) const { //重载下标操作符,返回引用:T&,可作为左值
    // assert 0 <= r < _size; // 此处对于意外和错误简单处理,实际应用需严格
    return _elem[r];
}
```

main()函数测试：

```
int main(int argc, char const *argv[]){
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    Vector<int> myvector(arr, 0, 5); // 选取前 5 个
```

```
    myvector.show(); // [1, 2, 3, 4, 5]
    int a = myvector[1]; // 右值
    cout << "a = " << a << endl; //a = 2
    myvector[2] = 33; // 左值,因为返回引用,所以可以修改元素的值
    myvector.show();  //[1, 2, 33, 4, 5]
    return 0;
}
```

❁ 2.3.2 插入



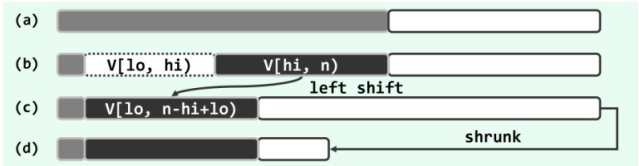

```
template <typename T>
Rank Vector<T>::insert(Rank r, T const &e) // r 处插入元素 e, 0<=r<=_size
{
    expand(); // 如果满了就扩容
    for (int i = _size; i > r; i--) // 自后向前,r 后的元素后移一个单元
        _elem[i] = _elem[i - 1];
    _elem[r] = e; // r 处插入 e
    _size++;      // 更新元素个数
    return r;     // 返回秩
}
```

main()函数测试：

```
int main(int argc, char const *argv[]){
    // ...接着上面继续测试
    myvector.insert(2, 99); // 2 号位插入 99
    myvector.show(); // [1, 2, 99, 33, 4, 5]
    cout << "size = " << myvector.size() << endl; // size = 6
    return 0;
}
```

❁ 2.3.3 删除

① 区间删除




```
template <typename T> //删除区间[low, high)
int Vector::remove(Rank low, Rank high){
    if (low == high)
        return 0;
    while (high < _size) // [high,_size)元素向前向后,往前移动 high-low 位
        _elem[low++] = _elem[high++];
    _size = low;       // 循环结束时 low 就是新的元素个数
    shrink();          // 如有必要则缩容
    return high - low; // 返回删除元素个数
}
```

动态缩容 shrink()：//一般不是必须的

```
template <typename T>
void Vector<T>::shrink(){ //装填因子过小时压缩所占空间
    if (_capacity < DEFAULT_CAPACITY << 1)
        return; //不会收缩到 DEFAULT_CAPACITY 以下
    if (_size << 2 > _capacity)
        return;                    //以 25%为界,装填因子>25%不缩容
    T *oldElem = _elem;             //备份
    _elem = new T[_capacity >>= 1]; //容量减半
    for (int i = 0; i < _size; i++)
        _elem[i] = oldElem[i]; //复制原来内容
    delete[] oldElem;          //释放原空间
}
```

测试：

```
int main(int argc, char const *argv[]){
    // ...接着上面继续测试
    myvector.remove(2, 4);                      // 删除[2, 4)号元素
    myvector.show();                            // [1, 2, 4, 5]
    cout << "size = " << myvector.size() << endl; // size = 4
    return 0;
}
```

② 单元素删除

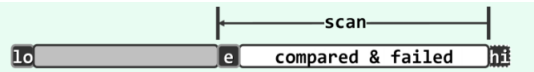
可以视为区间删除的特例。删除秩为 r 的元素，相当于区间删除[r, r+1)。

```
template <typename T> //删除秩为 r 的元素
T Vector<T>::remove(Rank r)
{
    T e = _elem[r];   // 记录删除元素
    remove(r, r + 1); // 调用区间删除
    return e;         // 返回被删除元素
}
```

所以说为什么不反过来，基于单元素删除，通过反复调用，实现区间删除呢?
单元素删除耗时主要是后继元素往前移操作，正比于删除元素后缀的长度$= n - r = O(n)$；循环次数为区间宽度$= high - low = O(n)$。
基于单元素删除的区间删除复杂度为$O(n^2)$；而之前的区间删除复杂度为$O(n)$。

2.3.4 查找

不是所有类型都支持判等和比较。假设：
无序向量：T 为可判等的基本类型，或已重载操作符==或!=
有序向量：T 为可比较的基本类型，或已重载操作符<或>




```
template <typename T> // 无序向量的逆序查找:返回元素 e 的最后位置,不存在返回 low-1
Rank Vector<T>::find(T const &e, Rank low, Rank high) const{
    while ((--high >= low) && (e != _elem[high])) // 从后向前
        ;
    return high; // 返回 high 是元素的秩(索引),如果元素不存在则 high=low-1
}
```

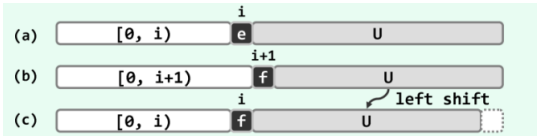
是否查找成功，交由上层调用者判断。
该查找算法，最好情况：一上来就命中$O(1)$；最坏情况：遍历完$O(n)$。
具体复杂度与输入相关，此类算法称为输入敏感(input-sensitive)的算法。

❀ 2.3.5 唯一化

去除向量中重复元素。

```
template <typename T> // 删除无序向量中重复元素
int Vector<T>::deduplicate()
{
    int oldSize = _size; // 记录原来元素个数
    Rank i = 1;          // 从_elem[1]开始查重
    while (i < _size)    // 从前向后考察每个元素
    {
        if (find(_elem[i], 0, i) < 0) // 该元素在前面有没有出现(至多一个)
            i++;                      // 没有重复,指针向后移
        else
            remove(i); // 有重复,删除该位置元素
    }
    return oldSize - _size; // 返回被删除元素个数
}
```

不变性：在循环中，当前元素_elem[i]的前缀_elem[0, i)中各元素彼此互异
初始 i=1 时，只有唯一前驱_elem[0]，自然成立。一般情况：



如图(a)，假设_elem[i]前缀元素互异。经过此步迭代有两种结果：
(1) e 与前缀_elem[0, i)任何元素不相同，如图(b)。i++后，新的前缀_elem[0, i)同样满足不变性，其规模增加一个单位。
(2) 前缀含有与 e 相同元素，如图(c)，由前面满足的不变性可知，相同的元素最多只有一个。删除 e 后，前缀_elem[0, i)依然保持不变性。

单调性：随着反复迭代：
(1) 当前元素前缀的长度单调非降，迟早增至_size
(2) 当前元素后缀的长度单调下降，迟早减至 0
故算法必然终止，且最多迭代$O(n)$轮。

复杂度：
find()对于前缀查找，remove()对于后缀元素往前移，故每轮迭代 find()和 remove()累计消耗$O(n)$时间，总体位为$O(n^2)$

进一步优化：
(1) 仿照有序向量的 uniquify()高效版思路，元素移动次数降至$O(n)$，但比较次数仍是$O(n^2)$，而且稳定性将被破坏。
(2) 先对需要删除的元素做标记，最后统一删除。稳定性保持，但查找长度更长，导致更多的比对操作。
(3) 先 sort()变为有序向量，再 uniquify()：简明实现最优$O(n \log n)$。

❀ 2.3.6 遍历

统一对各元素分别实施 visit 操作(事先约定的操作)。

如何指定 visit 操作?如何将其传递到向量内部?

① 利用函数指针机制，只读或局部性修改

```
template <typename T>
void Vector<T>::traverse(void (*visit)(T &)) //函数指针
{
    for (int i = 0; i < _size; i++)
        visit(_elem[i]);
}
```

② 利用函数对象机制，可全局性修改

```
template <typename T>
template <typename VST>
void Vector<T>::traverse(VST &visit) //函数对象
{
    for (int i = 0; i < _size; i++)
        visit(_elem[i]);
}
```

后者的功能更强，适用范围更广。

实例：将向量所有元素加一

首先需要实现一个可使单个 T 类型元素加一的类：

```
template <typename T> // 假设 T 可直接递增或已重载操作符++
struct Increase       // 简化起见使用 struct
{                    // 函数对象,通过重载操作符"()"实现,行为上类似于函数
    virtual void operator()(T &e) { e++; }
};
```

定义一个 increase()函数：

```
template <typename T>
void increase(Vector<T> &v){
    v.traverse(Increase<T>());
}
```

```
// 运行报错...
no known conversion for argument 1 from ‘Increase<int>’ to ‘Increase<int>&’
```


第 3 章