# 图例

背景知识 Background

这是一个背景知识。

例子 Example

这是一个例子。

不变量 Invariant

这里会写一些性质。

这里可能会有一些人话版。

定义 Definition

这是一个定义。

通常底下会有一句人话版。

# 聊点基础的东西

## 计算机存储体系和数据结构

讲一点内存的性质，因为数据结构通常都是设计被放在内存里的， 也是根据内存的存储特性进行设计的。

计算机存储金字塔 Background 1

计算机的存储体系被设计成一个金字塔结构。

计算机里的数据都是从外存读到内存，再从内存读到寄存器/缓存， 然后 CPU 才会从寄存器/缓存拿到的。 这样的话，对同一个数据进行反复操作的时候， 就不用反复读写很慢的硬盘了。

内存有几个很重要的特性（与机械硬盘对比）：

- **随机访问**：内存是一个按字节编址的线性空间， 它访问任何一个地址的时间基本一致。
- **空间局部性**：现代 CPU 取东西的时候会顺便把旁边的东西也预取出来， 因此访问比较近的内存比访问更远的会快一些。（这也是链表为什么慢一点）
- 易失性：断电数据就丢了。

讲到内存地址通常记作形如 `0x30000000` 的 16 进制数字的时候， 发现需要补一下关于数制的概念。

数制 Background 2

简单来说，n 进制就是“满 n 进 1”的计数规则。在 n 进制中， 人们使用 n 个符号表示一个数字。如在二进制中，使用了 `0`, `1` 两个符号。

位值制 Example 3

这里写一下同一个数字（十进制下的 13）在不同进制下的表示。

| 进制 | 按权展开式的意义 (权重的组合) | 在该进制下的表示 |
|---|---|---|
| 2 | $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$ | $(1101)_2$ |
| 10 | $1 \cdot 10^1 + 3 \cdot 10^0$ | $(13)_{10}$ |
| 16 | $13 \cdot 16^0 \to \mathrm{D} \cdot 16^0$ | $(\mathrm{D})_{16}$ |

使用十六进制通常是为了让人方便看， 因为十六进制的 1 个字符一一对应于二进制的 4 个字符。比如当说到 $(\mathrm{A3})_{16}$ 的时候，这个数字其实是 $(10100011)_2$。

总是写 $()_{16}$ 也很麻烦，所以也用 `0xA3` 表示 $(\mathrm{A3})_{16}$。 这里的 x 是十六进制 (hexadecimal) 的意思。

## 抽象数据类型

然后说说为什么有**抽象数据类型** (Abstract Data Type, ADT) 这个东西。

因为不同计算机或者语言之间差别很大， 而写程序的人希望关心数据本身的性质， 而不是在每个不同的地方它们的表现。

比如对于矩阵

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

在 C 语言里，数据在内存上是这么放的（Recall：内存是一个**线性模型**）：

`1 2 3 4 5 6 7 8 9`

在 Fortran 语言里，则是这么放的：

`1 4 7 2 5 8 3 6 9`

**矩阵的存储方式** Background 4

上面介绍的两种存储矩阵的方式没有对错之分。 像 C 语言这样逐行存储的做法被称作**行主序**， 而像 Fortran 这样的做法被称作**列主序**。

如果我们想要计算这个矩阵第三列的和， 那在 C 中就要做

`a[0][2] + a[1][2] + a[2][2] = a[2] + a[5] + a[8]`

而在 Fortran 中则是

`a[2][0] + a[2][1] + a[2][2] = a[6] + a[7] + a[8]`

这个显然是挺恶心的一件事，但是数据本身的结构是清晰的， 所以如果我们抽象出一个叫做**矩阵**的东西， 在计算的时候就在两边都只关心 `sum matrix.col(3)` 就可以了。由这个抽象数据类型的实现者负责把 `col(3)` 这个东西翻译到两个语言里不同的实现去。

## 编程范式

简单说一下这个东西， 是因为数据结构这个东西本身是带有强烈的**面向对象**的风格的。

| C 风格（过程式，结构化编程） | C++ 风格（面向对象编程） |
| --- | --- |
| `写字(cuso4d, 字)` | `cuso4d.写字(字)` |

在面向对象编程中，程序员认为程序是一系列相互作用的对象， 对象有自己的性质和能做的事情（方法）。 这个和数据结构想做的事情非常类似： 每个抽象数据类型都有自己能做的事情。

## 算法

**算法** Definition 5

**算法**是为了解决某类问题而规定的一个**有限长**的**操作序列**。

人话版的重点：能执行完，每一步都能让计算机知道怎么做的，操作序列。

**算法的一堆性质** Invariant 6

一个算法必须满足以下 5 个重要特性。

- **有穷性**。一个算法必须总是在执行有穷步后结束，且每一步都必须在有穷时间内完成。
- **确定性**。对于每种情况下所应执行的操作，在算法中都有确切的规定， 不会产生二义性，算法的执行者或阅读者都能明确其含义及如何执行。
- **可行性**。算法中的所有操作都可以通过将已经实现的基本操作运算执行有限次来实现。
- **输入**。一个算法有 0 个或多个输入。
- **输出**。一个算法有一个或多个输出，它们是算法进行信息加工后得到的结果，无输出的算法没有任何意义。

---

- **有穷性**。不能死循环。
- **确定性**。不能写“加少许盐”，要写“加 2g NaCl”。
- **可行性**。不能写“预测明天的彩票号码”，因为计算机做不到。
- **输入**。算法是用来处理东西的。
- **输出**。算法处理完得有个结果。

衡量算法好坏的直观看法就是算法能跑多快， 但是算法输入有时候大有时候小，怎么做呢？ 有一个办法就是观察“数据量翻倍的时候，时间会怎么翻倍”。

**基本语句** Definition 7

基本语句”指的是算法中重复执行次数和算法的执行时间成正比的语句， 它对算法运行时间的贡献最大。

---

就是这个算法里单次运行耗时比较大的语句，它之后会被当成 $O(1)$。

**随便举个例子** Example 8

我回家之后可能会开灯，打游戏。 这两个玩意放一起的时候，打游戏就是那个基本语句。 因为通常来说，我开 1 次灯打 1 次游戏要用 2h， 我开 100 次灯再打 1 次游戏要用 2h 1min。 但是我要是开 1 次灯打 2 次游戏那就要用 4h 了。

### 时间复杂度

Definition 9

一般情况下，算法中基本语句重复执行的次数是问题规模 $n$ 的某个函数$f(n)$， 算法的时间量度记作:

$$T(n) = O(f(n)).$$

它表示随着问题规模 $n$ 的增大，算法执行时间的增长率和 $f(n)$ 的增长率相同，称作算法的渐近时间复杂度，简称时间复杂度。

就是说随着输入规模变大的时候，从长远来看（不管小输入的时候如何）， 需要的时间会增加得多快。

#### 用数字举个例子

Example 10

| 输入规模 $n$ | $n = O(n)$ | $n^2 = O(n^2)$ | $10n^2 + 100 = O(n^2)$ | $2^n = O(2^n)$ |
|---|---|---|---|---|
| 10 | 10 | 100 | 1,100 | 1,024 |
| 100 | 100 | 10,000 | 100,100 | $1.27 \times 10^{30}$ |
| 1000 | 1000 | 1,000,000 | 10,000,100 | $1.07 \times 10^{301}$ |

在这个例子里，$10n^2$ 的部分虽然最开始和 $2^n$ 差不多， 但是数据量增大的时候很快就被甩开了，然后它和 $n^2$ 越来越趋于一致。

因此在粗粒度考虑的时候，只关注 $O(f(n))$ 就可以了， 常系数和后面阶数更低的项不是很需要关心。

### 矩阵乘法

Example 11

以这个算法为例子，分析它的时间复杂度。

```
for i in range(n):                      # n + 1
    for j in range(n):                  # (n + 1) * n
        sum = 0                         # n * n
        for k in range(n):              # (n + 1) * n * n
            sum += A[i][k] * B[k][j]    # n * n * n
        C[i][j] = sum                   # n * n
```

这里的基本语句是乘法和赋值那句，所以

$$T(n) = n^3 = O(n^3).$$