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。

算法 $O(f(n))$ Algorithm

算法相关的东西

ADT: 某个数据结构的名字

数据元素： **数据关系：**

基本操作：

> 某个方法的名字 $O(f(n))$ Operation
>
> 这个方法的介绍

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	1.27×10^{30}
1000	1000	1,000,000	10,000,100	1.07×10^{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 $O(n^3)$ 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

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

串

串也被翻译成字符串，它也是一个特殊的线性表。（Recall：栈和队列是被 ban 掉了某些操作的线性表）串是要求数据元素都得是字符的线性表。后文中使用 Σ 代表字符的集合。

做这个抽象我觉得有三个原因：

- 字符串这个东西比较常用；
- 字符串有一些很常用的操作不是线性表的基本操作。比如子串匹配。
- 处理字符和处理数值在计算机上的实现不同，因此会引入新的存储结构。

串 Definition 12

串是由零个或多个字符组成的有限序列。一般记为：

$$s = a_1 a_2 ... a_n . (n \geq 0)$$

若串由零个字符组成，称为**空串**，后文中记为 $s = \emptyset$。

重点就是 0+ 个字符（0 个也算），有限（不可以无限长），序列（有顺序，和集合相对）。

子串 Definition 13

串里任意个连续的字符组成的子序列称为该串的子串。

ADT: 串

数据元素：
$D = \{a_i \mid a_i \in \Sigma\}, i = 1, ..., n, n \geq 0.$

数据关系：
和线性表一样

基本操作：

`s.copy() -> str` $O(n)$ Operation 14

复制自己得到一个新串。

`s.compare(t: str) -> int` $O(n)$ Operation 15

比较自己和另一个串的大小。返回它们差了多远。

字典序 Background 16

先定义字符的顺序：`0 < 1 < ... < 9 < A < ... < Z < a < ... < z`。

字符串的序关系是：从最前面的字符开始比， 一直比到第一个不一样的字符，这时候谁的这个字符大谁就大。 如果有一个字符串更短，先结束了（即它是另一个的前缀），那先结束的那个小。 比如：*apple* < *apples*, *apple* < *applf*.

`s.length() -> int` $O(n)$ Operation 17

返回自己的长度。

`s.concat(s2: str) -> str` $O(n+m)$ Operation 18

返回自己和另一个串的拼接。

存储结构

首先串是特殊的线性表，所以和线性表一样有很基本的顺序存储和链式存储。

不过由于链式存储的时候如果一个节点只存一个字符那也太浪费空间了， 很多时候链串里的一个节点可以放好几个字符。这是通常线性表不会做的。

子串匹配

经常会遇到一个问题就是一个字符串是不是另一个字符串的子串。

这时候这个拿来检验的子串也叫做模式 (pattern) 串。

例子：在主串 `ABABABC` 中找是否存在子串 `ABABC`。

BF 算法进行子串匹配 $O(m*n)$ Algorithm 19

把子串和主串从第一位开始对齐，依次往后比， 出现失配的时候，就把子串往右挪一次再比。

直到如果某次子串全匹配了就是存在，如果到最后都没全匹配就是不存在。

```
a b a b a b c
a b a b c
= = = = ^

a b a b a b c
  a b a b c
  ^

a b a b a b c
    a b a b c
    = = = = =
```

KMP 算法 $O(m+n)$ Algorithm 20

别的和 BF 一样，但是如果出现不匹配的时候，利用之前比过的信息。

根据模式串的结构，在失配的位置，模式串的末尾和开头都是 `AB`， 那这个时候，虽然失配了，但是我们知道主串失配前那个位置都是匹配的， 也就是我们知道模式串的开头肯定和主串失配前的位置是匹配的。

所以模式串可以往后退很多，不用只挪一个格子。

```
 a b  [a b] a b c
[a b] [a b] c
 = =   = =  ^

a b [a b] a b c
    [a b] a b c
          = = =
```

KMP 算法为了实现我们刚刚说到的这个直觉，引入了一个 `next` 数组。 这个数组是用来分析模式串的结构的：当模式串在第 `j` 个位置失配时， 前面的子串里，最长的一模一样的头尾有多长。

对于上面的例子 `ABABC`， `next = [0, 0, 0, 1, 2]`， 也就是说，如果在 `C` 失配了，可以直接把模式串的 `[2]` 位置给对到主串的指针上去。

更强大的子串匹配

有些时候搜索的要求更高，不一定是子串， 但是可能是有一些特殊的结构的。

找到文本里的取件码 Example 21

取件码短信格式还挺多的。

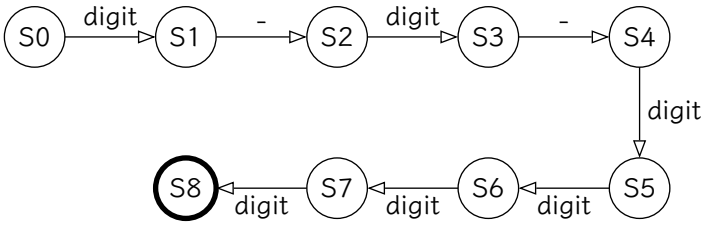
- 【菜鸟驿站】您的快递已到达，取件码 9-2-4015，请及时领取。
- [顺丰速运] 凭 1-2-3018 到店取件。

我们只知道取件码的格式是 数字-数字-4 个数字。 事实上有人创造出了一种表示这种需求的语言，称为**正则表达式**。

正则表达式 Background 22

每一条正则表达式代表着一类可能的字符串。刚才的取件码可以被表示为：`\d-\d-\d{4}`，其中 `\d` 表示数字，`{4}` 表示前边的东西要出现 4 次。

在实现的时候，上面的这个正则表达式会被转换成这样的一个状态机：



数组

数组是要求每个位置的数据类型相同的线性表。

为什么要单独抽象出这个结构？我觉得有以下几个原因：

- 强调**维度**的概念。之前提到的其他结构通常是一维的，而数组是递归定义的。
- 极致利用**随机访问**的特性。数组的存储方式是完全连续而且非常规整， 有利于快速寻址和利用缓存局部性。

n 维数组 Definition 23

1 维数组是一个线性表，它的每个元素的数据类型都是相同的，称为**原子类型**。

n 维数组是一个线性表，它的每个元素都是一个 n - 1 维数组，且它们的长度相等。

n 维数组其实就是数学中 n 维张量的对应。 1 维数组就是向量，2 维数组就是矩阵。 n 维数组就是像矩阵这样看起来方方的一个结构，所以放在内存里很好寻址。

ADT: 数组

数据元素：
原子类型都相同就行

数据关系：
在每一个维度方向上有前驱和后继

基本操作：

`init(b1, ..., bn) -> Array` $O(\prod b_i)$ Operation 24

初始化一个第 `i` 个维度的长度为 `bi` 的数组。

`value(i1, ..., in) -> ElemType` $O(1)$ Operation 25

取值。

`assign(value, i1, ..., in) -> None` $O(1)$ Operation 26

赋值。

关于数组，最需要知道的就是它在顺序存储下，怎么做到 $O(1)$ 访存。 只要能一下子算出来我想访问的内存的地址就可以了。

4 维数组的寻址 $O(1)$ Example 27

假设我有一个行主序的 4 维数组 A，我知道它的首地址是 0x30000000， 它的每个元素要占用 4 个字节。假设它的维度的长度为 (11, 13, 17, 19)，现在我想知道 A[2][3][5][7] 的值， 我要怎么做?

对于行主序的数组来说，数据在整个内存里的分布是这样的:

a[0][:][:][:] a[1][:][:][:] a[2][:][:][:] ...

其中的每个大块，如 a[1][:][:][:] 里边又是这样的:

a[1][0][:][:] a[1][1][:][:] a[1][2][:][:] ...

因此在访问 a[2][3][5][7] 的时候，我们就是先跳过前 2 个最大的块， 然后跳过前 3 个次大的块，再跳过前 5 个次次大的块， 然后找跳过前 7 个元素，看第 8 个位置就可以了。

每个最大的块有多大? 一个最大的块里应该放了后三个维度所有的东西， 所以应该有 $13 * 17 * 19$ 这么多元素。 次大的块自然就有 $17 * 19$ 这么多。 所以 目标元素应该是从最开始往后多少个元素? 应该是

$$n = 2 * (13 * 17 * 19) + 3 * (17 * 19) + 5 * (19) + 7$$

这么多个。 而每个元素占用了 4 个字节，所以从首地址要往后多少字节?

$$l = 4 * n = 37876 = \texttt{0x93F4}.$$

所以我直接读取 0x300093f4 开始，往后 4 个字节的数据就可以了。

特殊数组的存储

有一些矩阵有特殊的性质。存的时候可能可以特殊对待。 但是如果特殊对待了，它们的存储结构就不再是上面说的这种方方正正在内存里了。 所以我觉得这东西不应该在数组讲。广义表的时候可以提一下。

简单说一个 insight 吧。

0	0	0	0	8
0	0	3	0	0
0	0	0	0	0
1	0	0	0	0
0	0	0	6	0

Figure 1: 逻辑结构: 稀疏矩阵

Row	Col	Value
0	4	8
1	2	3
3	0	1
4	3	6

Table 1: 存储结构: 三元组表

广义表

广义表是线性表的推广，它不再要求每个位置的数据类型相同，也不再追求齐次性。

为什么要单独抽象出这个结构？

- 支持嵌套逻辑：线性表只能存原子数据，而广义表可以存“列表的列表”，这在处理如 JSON、Lisp 表达式或目录树等递归结构时非常自然。
- 表达能力更强：它是递归定义的终极形态。如果说数组是“规整的方阵”，广义表就是“自由的树状结构”。

广义表 Definition 28

广义表 L 是 n 个元素 (a1,a2,⋯,an) 的有限序列，其中每个 ai 或者是原子类型，或者是一个广义表。 数学环境？

补点人话

ADT: 广义表

数据元素：
原子类型或广义表类型

数据关系：
仅在最外层存在线性的前驱和后继关系，但元素内部可进一步嵌套

基本操作：

GetHead(L) -> ElemType　O(1)　Operation 29

获取广义表的第一项。

GetTail(L) -> List　O(1)　Operation 30

获取广义表除第一项外的其余部分（返回的一定是表）。

Depth(L) -> Int　O(L) ?　Operation 31

递归计算表的嵌套深度。

Length(L) -> Int　O(1)　Operation 32

获取最外层序列的长度。