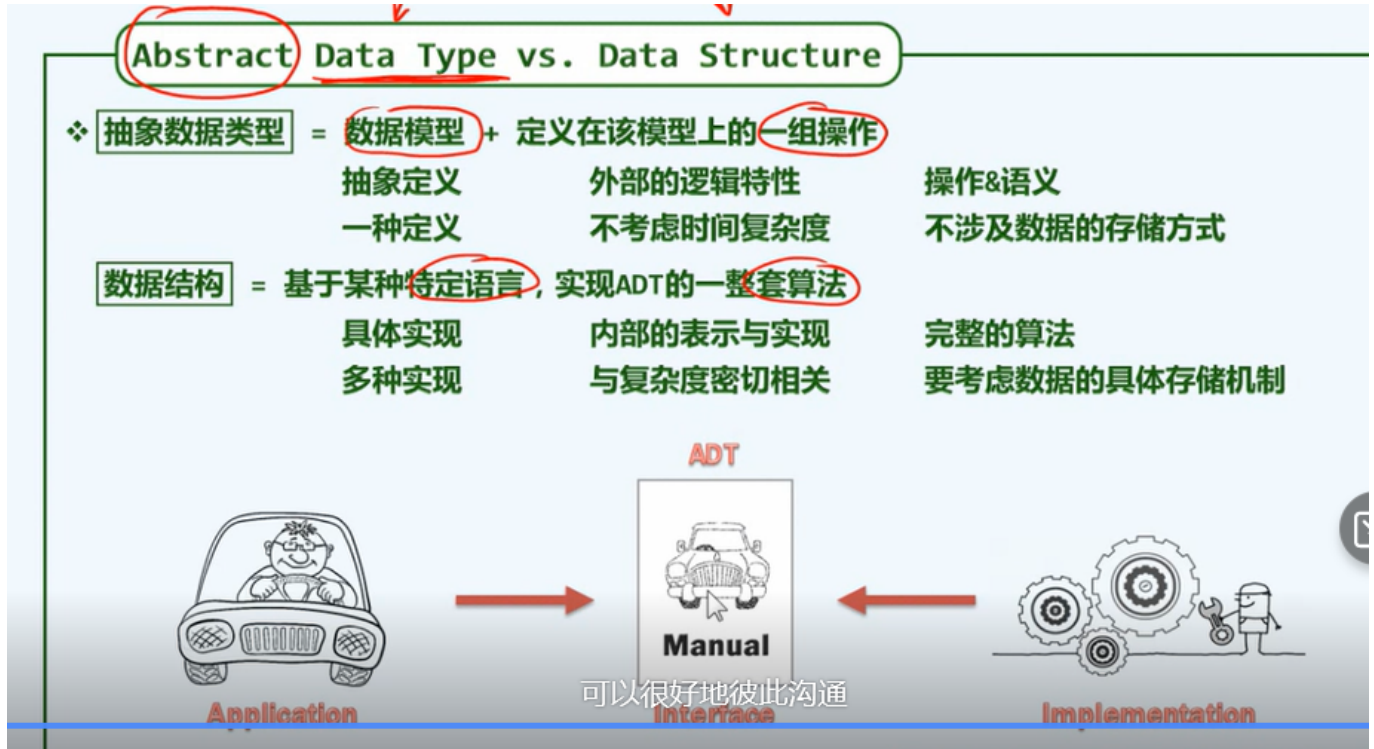


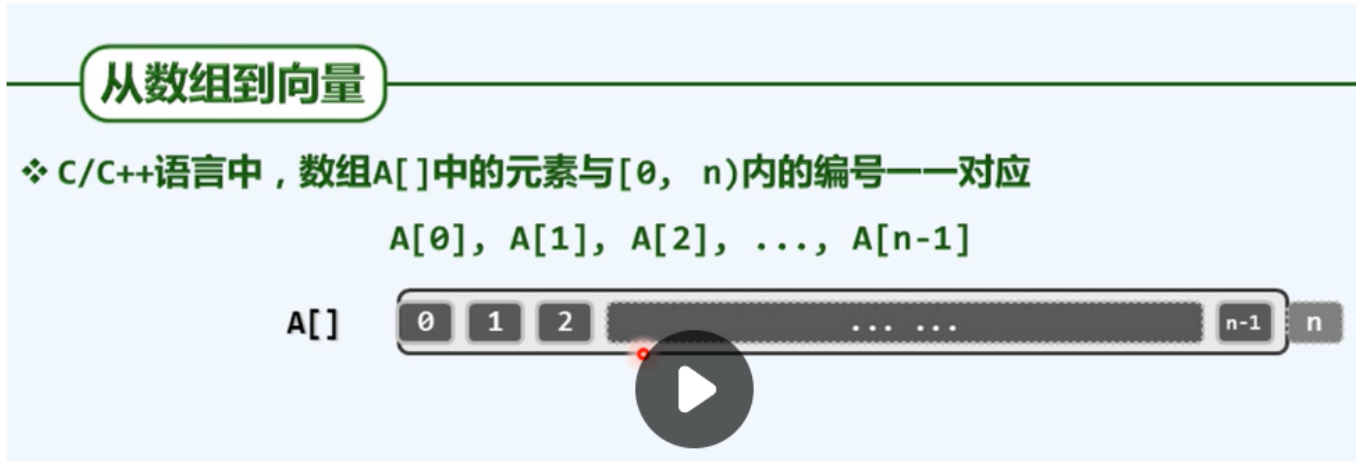
##1.抽象数据结构 <<可以理解为，类似于向量，列表那种有操作方法的，叫做抽象数据结构

##2.数据结构 <<通俗来讲的话，数据结构可以说是int，double，数据类型

看图！



##3.数组的定义 可以看这张图



数组就是以一个个条的形式，等单位的存储数据

最后一位作为“哨兵”，实际上不存在的第n位元素，以此来理解算法 ##4.复杂度 $O(1)$ 复杂度：没有转向（循环、递归、分支）的算法 但也不一定

$O(1)$

xuetangx.com 已进入全屏模式 退出全屏模式(Esc)

❖ 常数 (constant function)

 $2 = 2013 = 2013 \times 2013 = O(1)$, 甚至 $2013^{2013} = O(1)$

//含RAM各基本操作

❖ 这类算法的效率最高

//总不能奢望不劳而获吧

❖ 什么样的代码段对应于常数执行时间?

//应具体分析

一定不含循环?

for ($i = 0; i < n; i += n/2013 + 1$);for ($i = 1; i < n; i = 1 \ll i$);// $\log^* n$, 几乎常数

一定不含分支转向?

if $((n + m) * (n + m) < 4 * n * m)$ goto UNREACHABLE;

//不考虑溢出

一定不能有 (递归) 调用?

if $(2 == (n * n) \% 5)$ $O(1)(n)$; $O(\log n)$ 复杂度: 该类算法非常有效! 底数是无所谓的 $O(\log^c n)$ ❖ 对数 $O(\log n)$ $\ln n \mid \lg n \mid \log_{100} n \mid \log_{2013} n$

❖ 常底数无所谓

 $\forall \underline{a}, \underline{b} > 0, \log_a n = \cancel{\log_a b} \cdot \log_b n = \Theta(\log_b n)$ $O(nc)$ 复杂度: 多项式复杂度

❖ 多项式 (polynomial function)

$$100n^1 + 200 = O(n)$$

$$(100n - 500)(20n^2 - 300n + 2013) = O(n \times n^2) = O(n^3)$$

$$(2013n^2 - 20)/(1999n - 1) = O(n^2/n) = O(n)$$

$$\text{一般地: } a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 = O(n^k), a_k > 0$$

❖ 线性 (linear function) : 所有 $O(n)$ 类函数

❖ 从 $O(n)$ 到 $O(n^2)$: 编程习题主要覆盖的范围

$$\text{❖ 幂: } [(n^{2013} - 24n^{2009})^{1/3} + 512n^{567} - 1978n^{123}]^{1/11} = O(n^{61})$$

❖ 这类算法的效率通常认为已可令人满意, 然而...

这个标准是否太低了?

凡多项式复杂度, 即可视作可解

#凡多项式复杂度, 即可视作可解 $O(2^n)$: 指数复杂度 这类算法的计算成本增长极快, 通常认为是不可忍受的 ##
设计难度上, 指数复杂度的算法设计较多项式复杂度来说比较容

循环减半(问题规模缩小了一半)的程序复杂度为 $O(\log n)$



如何简单快速地判断算法复杂度

► 快速判断算法复杂度 (适用于绝大多数简单情况) :

► 确定问题规模 n

► 循环减半过程 $\rightarrow \log n$

► k 层关于 n 的循环 $\rightarrow n^k$

► 复杂情况: 根据算法执行过程判断

空间复杂度：评估算法内存占用大小（时间比空间重要）“空间换时间”原则，速度越快越好

UF F Y C I T Y

空间复杂度

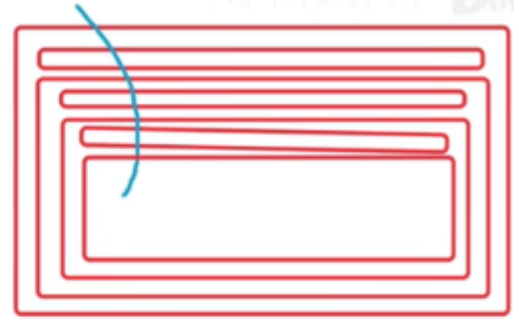
- ▶ 空间复杂度：用来评估算法内存占用大小的式子
- ▶ 空间复杂度的表示方式与时间复杂度完全一样
 - ▶ 算法使用了几个变量： $O(1)$
 - ▶ 算法使用了长度为 n 的一维列表： $O(n)$
 - ▶ 算法使用了 m 行 n 列的二维列表： $O(mn)$
- ▶ “空间换时间”

递归：递归具有两个特点

- 调用自身
- 结束条件

!!! 这似乎是个图标符号 关于递归的顶级理解

复习：递归



特点：

```
def func1(x):
    print(x)
    func1(x-1)
```

```
def func3(x):
    if x>0:
        print(x)
        func3(x-1)
```

```
def func2(x):
    if x>0:
        print(x)
        func2(x+1)
```

```
def func4(x):
    if x>0:
        func4(x-1)
        print(x)
```



上面是func3的图 下面是func4的图 函数的运行都是从上往下 func3的结果是 3 2 1 func4的结果是 1 2 3 讲的很好!

!!! 总结一下汉诺塔问题 我的代码复现：(始终着眼对n-1的处理)

```
def hanoi(n,a,b,c):
    if n > 0:
        hanoi(n-1,a,c,b)
        print('move from {} to {}'.format(a,c))#大盘子，可以抽象到第一步
        hanoi(n-1,b,a,c)
    print(hanoi(3,'A','B','C'))
```

我原来是怎么做的呢？

```
def hanoi(n, a, b, c):
    #n 圆盘个数
    #a 初始的桩
    #b 经过的桩
    #c 最后到的桩
    if n > 0 :
        #首先把n-1个圆盘从a经过c移动到b,调用自己
        hanoi(n-1, a, c, b)
```

```
print('move from {} to {}'.format(a,b))
#再把第1个圆盘从a移到c
print('move from {} to {}'.format(a,c))
#最后把n-1个经过a移动到c
hanoi(n-1, b, a, c)
print('move from {} to {}'.format(b,c))

print(hanoi(3, 'A', 'B', 'C'))
```

有点搞笑哈哈

错误出在第二步，无需对第一个盘子进行任何操作 **要知道的是，递归其实就是一种数学归纳法，不过是从n到1进行归纳 做数学归纳，其实只需关心的是“n-1”** 数学归纳一般适用于整体隔离分析的问题，把n化成n和n-1两个部分来分析

!!! 查找问题

LUFFY CITY

查找

- ▶ 查找：在一些数据元素中，通过一定的方法找出与给定关键字相同的数据元素的过程。
- ▶ 列表查找（线性表查找）：从列表中查找指定元素
 - ▶ 输入：列表、待查找元素
 - ▶ 输出：元素下标（未找到元素时一般返回None或-1）
- ▶ 内置列表查找函数：index()



顺序查找

从列表第一个元素开始，
顺序进行搜索，
直到找到元素或
搜索到列表最后一个元素

代码实现如下

```
def linear_search(li, val):
    for ind, v in enumerate(i):
        if v == val:
            return ind
```

```
else:
    return None
```

时间复杂度为 $O(n)$ (没有循环减半过程, 最多走 n 步)

二分查找 (折半查找) (只能用于有序序列)

从有序列表的初始候选区域 $li[0:n]$ 开始, 通过对待查找的值与候选区中间值的比较, 可以使候选区减少一半

代码实现

```
def binary_search(li, val):
    left, right = 0, len(li)-1
    while(left <= right):
        mid = (left + right) // 2
        if li[mid] == val:
            return mid
        elif li[mid] < val:
            left = mid + 1
        else:
            right = mid - 1
    else:
        return None
```

时间复杂度: $O(\log n)$

!!! 排序

列表排序

- ▶ 排序: 将一组“无序”的记录序列调整为“有序”的记录序列。
- ▶ 列表排序: 将无序列表变为有序列表
 - ▶ 输入: 列表
 - ▶ 输出: 有序列表
- ▶ 升序与降序
- ▶ 内置排序函数: `sort()`

常见的排序算法



常见排序算法

- ▶ 排序Low B三人组
 - ▶ 冒泡排序
 - ▶ 选择排序
 - ▶ 插入排序
- ▶ 排序NB三人组
 - ▶ 快速排序
 - ▶ 堆排序
 - ▶ 归并排序
- ▶ 其他排序
 - ▶ 希尔排序
 - ▶ 计数排序
 - ▶ 基数排序



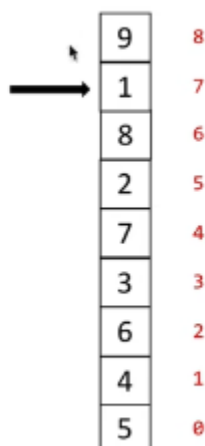
#1冒泡排序(Bubble Sort)

路飞学城
LUFFYCITY

哔哩哔哩 bilibili

冒泡排序 (Bubble Sort)

- ▶ 列表每两个相邻的数，如果前面比后面大，则交换这两个数。
- ▶ 一趟排序完成后，则无序区减少一个数，有序区增加一个数。



冒泡排序的示例图

可以发现，“冒

“泡”是此类排序方法的一个形象表示，越往上泡泡越大 然后，这里需要注意的是冒泡的次数为总数-1次，因为指针指到最后一个元素的时候不再需要冒泡了

我自己写的冒泡!!! 成功啦!!!! 哇哈哈 自我总结: 冒泡的关键在于, 趟数, 以及无序有序区间的理解。因为指针到最后一位无需再遍历, 所以只会n-1次。而无序有序的理解呢, **进入有序区的, 就不需要再遍历了, 而进入有序区的数字是多少? 就是i, 总共遍历的次数** *也可以换一个角度来理解, 即双指针。*这个仍需要思考, 并不是很完善 头指针是小, 尾指针是大, 当尾指针完成遍历的时候, 就完成了排序 而头指针负责交换元素等工作, 而且头指针只能在[列表开头, 头指针位置]这个区间内移动。头指针长度为n, 尾指针长度为i 总长度为n 所以可得总共遍历次数为n-i-1 (当头=尾时候不需要遍历了)

```
def bubblesort(lst):
    for i in range(len(lst)-1): #一共n-1趟
        for j in range(len(lst)-1-i):
            if lst[j] > lst[j+1]:
                lst[j], lst[j+1] = lst[j+1], lst[j]
    return lst

alist = [4,5,6,1,2,7,9]
new = bubblesort(alist)
print(new)
```

答案基本差不多 时间复杂度是 $O(n^2)$ **冒泡排序还有个可以改进的地方**

```
def bubblesort(lst):
    for i in range(len(lst)-1): #一共n-1趟
        exchange = False #此处进行标记, 如果没有交换的话就直接return列表
        for j in range(len(lst)-1-i):
            if lst[j] > lst[j+1]:
                lst[j], lst[j+1] = lst[j+1], lst[j]
        #print(lst)
        if not exchange: #注意这个用法, 就是布尔表达式, 意思是如果exchange不是True
            return lst
    return lst

alist = [4,5,6,1,2,7,9]
new = bubblesort(alist)
print(new)
```

#2选择排序

复习一下pop和remove的用法区别: pop()需要索引作为参数; remove()需要具体值作为参数

插入排序的简单思路: 遍历一遍列表, 找出第一小的值, 遍历二遍列表, 找出第二小的值... 如此往复 那么可以得到思路简单、操作麻烦一点的做法如下: *上面是我写的, 下面是教程写的*

```

#这里是较为复杂一点的选择排序
def select1(alist):
    #创建新列表用于储存值
    newone = []
    for i in range(len(alist)):
        #思路：弹出最小值的索引-添加入新列表-返回
        newone.append(alist.pop(alist.index(min(alist))))
    return newone
#注意，写算法的时候最好不用内置函数哦(index等)
def select2(alist):
    newone = []
    for i in range(len(alist)):
        for j in alist:
            newval = min(alist)
            newone.append(newval)
            alist.remove(newval)    #使用remove索引会发生变化，不好
    return newone

```

select2算法有诸多缺陷，比如 1.相比于冒泡排序（原地排序，未占用多余的空间），这个写法占用了一段重复的空间 2.复杂度问题。 $\min()$ 的复杂度为 $O(n)$, $\text{remove}()$ 的复杂度又为 $O(n)$ ，所以导致整体算法的复杂度为 $O(n^2)$ 注意，是 $n * (n + n)$ **优化过后的选择排序代码**

```

def select_sort3(alist):
    for i in range(len(alist) - 1): #同冒泡排序的方法，最后一位不需要遍历
        min_loc = i
        for j in range(i + 1, len(alist) - 1):
            if alist[j] < alist[min_loc]:
                min_loc = j
        alist[i], alist[min_loc] = alist[min_loc], alist[i] #往有序区添加元素，宏观
        #来看可以视作是往列表最前面添加元素
    return alist

```

选择排序 (Select Sort)

- ▶ 一趟排序记录最小的数，放到第一个位置
- ▶ 再一趟排序记录记录列表无序区最小的数，放到第二个位置
- ▶
- ▶ 算法关键点：有序区和无序区、无序区最小数的位置

另一个

理解角度：递归

我用递归的方式来解释一下
f(x)代表第1~x小的数字
是最小的x的数字 f(0)为原排列（或者理解为空）
f(x)=f(x-1)加上第x小的数字
比如5 4 3 2 1
f(0)=. 5 4 3 2 1
f(1)=1 5 4 3 2
f(2)=1 2 5 4 3
以此类推

附上这次写的几个函数代码汇总&思考

```
import random

#这里是较为复杂一点的插入排序
def select_sort1(alist):
    #创建新列表用于储存值
    newone = []
    for i in range(len(alist)):
        #思路：弹出最小值的索引-添加入新列表-返回
        newone.append(alist.pop(alist.index(min(alist))))
    return newone
#注意，写算法的时候最好不用内置函数哦(index等)
def select_sort2(alist):
    newone = []
    for i in range(len(alist)):
        for j in alist:
            newval = min(alist)
            newone.append(newval)
            alist.remove(newval)    #使用remove索引会发生变化，不好
```

```
    return newone

def select_sort3(alist):
    for i in range(len(alist) - 1): #同冒泡排序的方法, 最后一位不需要遍历
        min_loc = i
        for j in range(i + 1, len(alist) - 1):
            if alist[j] < alist[min_loc]:
                min_loc = j
        if min_loc != j: #思考, 为什么这一步要这么做?
            alist[i], alist[min_loc] = alist[min_loc], alist[i]
    return alist

def select_sort4(alist):
    for i in range(len(alist) - 1): #同冒泡排序的方法, 最后一位不需要遍历
        min_loc = i
        for j in range(i + 1, len(alist) - 1):
            if alist[j] < alist[min_loc]:
                min_loc = j
        alist[i], alist[min_loc] = alist[min_loc], alist[i]
    return alist

# def selectdigui_sort5(alist):
#     wait to write

alist = [random.randint(1,100) for i in range(10)]
print(select_sort3(alist))
```