数据结构与算法

数据的基本结构：list，tuple，dictionary，set

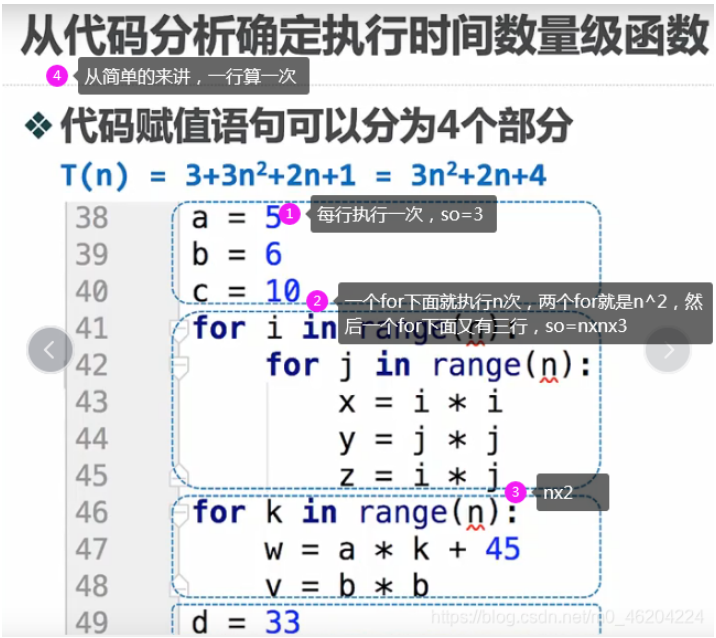
数据的基本算法：sorting，searching，recursion

时间复杂度 (import time, start or end=time.time)

不同方法间的复杂度

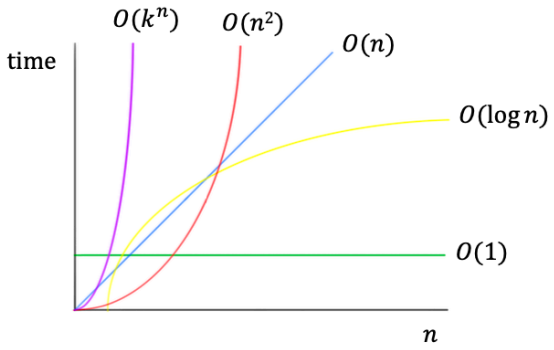
```python
import time
start= time.time()
for i in range(0,1001):
    for j in range(0,1001):
        for k in range(0,1001):
            if i+j+k==1000 and i**2+j**2==k**2:
                print(i,j,k)
end = time.time()
print("总开销：",end-start)#总开销：  126.49699997901917

start1= time.time()
for i in range(0,1001):
    for j in range(0,1001):
        k=1000-i-j
        if i**2+j**2==k**2:
            print(i,j,k)
end1= time.time()
print("总开销：",end1-start1)#总开销：  1.0120000839233398
```

1.T(n) 时间复杂度，n执行的步数



时间复杂度通过o()表示

복잡도의 표기 - Big O



1. **O(1)** : **이상적인 알고리즘**로써 n이 아무리 커져도 실제로 소요되는 시간은 사실상 증가하지 않는다.
2. **O(n)** : **괜찮은 알고리즘**로써 n이 커지면 그의 비례해서 실제로 소요되는 시간도 증가한다.
3. **O(logn)** : **바람직한 알고리즘**로써 n이 커지면 실제로 소요되는 시간도 증가하지만 그 증가폭이 완만하게 증가한다.
4. **O(n^2)** : **부담스러운 알고리즘**로써 n이 커지면 급격하게 실제로 소요되는 시간도 증가한다.
5. **O(k^n)** : **피해야 하는 알고리즘**로써 n이 조금만 커져도 사실상 실행할 수 없는 알고리즘이 된다.

O()分类法：

常见时间复杂度

1 只保留最大项

执行次数函数举例	阶	非正式术语
12	O(1)	常数阶
2n+3	O(n)	线性阶
$3n^2+2n+1$	$O(n^2)$	平方阶
$5\log_2 n+20$	O(logn)	对数阶
$2n+3n\log_2 n+19$	O(nlogn)	nlogn阶
$6n^3+2n^2+3n+4$	$O(n^3)$	立方阶
2^n	$O(2^n)$	指数阶

注意，经常将$\log_2 n$（以2为底的对数）简写成logn

2.最坏时间复杂度（알고리즘이 가장 오래 걸릴 수 있는 경우에 대한 실행 시간）

예시로 이해하기

1. 선형 검색(Linear Search)

리스트에서 특정 값을 찾는 알고리즘입니다.

python 복사 편집

```python
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
```

- 입력: `arr = [1, 2, 3, 4, 5]`, `target = 5`
- **최악의 경우**:
  - `target`이 리스트의 마지막에 있거나 없을 때.
  - 시간 복잡도는 **O(n)**.

2. 이진 검색(Binary Search)

정렬된 리스트에서 값을 찾는 알고리즘입니다.

python

```python
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

- 입력: `arr = [1, 2, 3, 4, 5]`, `target = 6`
- **최악의 경우**:
  - 탐색 범위가 계속 반으로 줄어들다가 값을 찾지 못할 때.
  - 시간 복잡도는 **O(log n)**.

몇 가지 알고리즘의 최악 시간 복잡도

알고리즘	최악 시간 복잡도	설명
선형 검색 (Linear Search)	O(n)	리스트 전체를 순회
이진 검색 (Binary Search)	O(log n)	탐색 범위를 반씩 줄임
버블 정렬 (Bubble Sort)	$O(n^2)$	정렬되지 않은 경우 반복 필요
퀵 정렬 (Quick Sort)	$O(n^2)$	피벗 선택이 매번 비효율적일 때
병합 정렬 (Merge Sort)	O(n log n)	항상 일정한 분할, 병합 수행

이진 검색 예시 最坏时间复杂度 구한 과정

왜 O(log n)인지 단계별 설명

1. 초기 탐색 범위

- 처음에는 리스트의 전체 길이 `n`을 탐색합니다.
- 중간 값을 확인하고 탐색 범위를 절반으로 줄입니다.
  즉, 탐색 범위는 `n/2`가 됩니다.

2. 두 번째 탐색

- 남은 리스트 길이는 `n/2`.
- 다시 절반으로 줄어들어 탐색 범위는 `n/4`.

3. 반복 과정

- 이 과정이 반복되면서 탐색 범위는 `n`, `n/2`, `n/4`, `n/8`...으로 줄어듭니다.
- 탐색 범위가 1이 될 때까지 반복합니다.

4. 몇 번의 단계가 필요한가?

- 탐색 범위가 `n`에서 1이 되기 위해 필요한 단계 수는 다음과 같습니다:

$$n \times \frac{1}{2^k} = 1$$

여기서 k는 탐색 단계 수입니다.

- 양쪽에서 로그를 취하면:

$$k = \log_2(n)$$

- 따라서, 이진 검색의 시간 복잡도는 **O(log n)**입니다.

list内置操作的时间复杂度 N:执行的步数

Operation	Big-O Efficiency	
indexx[]	$O(1)$	1 找索引一步实现
index assignment	$O(1)$	2 索引处赋值
append	$O(1)$	
pop()	$O(1)$	3 从尾部弹出
pop(i)	$O(n)$	4 从后往前弹出，最坏情况i=0，所以要n步
insert(i,item)	$O(n)$	1 从后往前，最坏i=0，n
del operator	$O(n)$	
iteration	$O(n)$	2 for操作
contains (in)	$O(n)$	3 看i是否在列表中，要先遍历一次
get slice [x:y]	$O(k)$	4 取切片，定位索引是一步，k=y-x
del slice	$O(n)$	5 把第一个删了，后面的会前移
set slice	$O(n+k)$	6 切片更换，先把原来的删了要n步，再加k个新的
reverse	$O(n)$	
concatenate	$O(k)$	7 两列表加，k表示第二个列表的元素，所以要执行k步
sort	$O(n \log n)$	
multiply	$O(nk)$	8 n，k两列表相乘

Table 2.2: Big-O Efficiency of Python List Operators

dict内置操作的时间复杂度

Operation	Big-O Efficiency	
copy	$O(n)$	
get item	$O(1)$	
set item	$O(1)$	
delete item	$O(1)$	
contains (in)	$O(1)$	9 使用键，就不用遍历
iteration	$O(n)$	

Time.it, list, dic 内置函数

```
from timeit import Timer
def test3():
    l = [i for i in range(1000)]
t3 = Timer("test3()", "from __main__ import test3")#1函数名，2import，因为这个Timer不一定在这里运行
print("comprehension ",t3.timeit(number=10000), "seconds")#test3()执行10000次后，10000次总的执行时间

import time
start=time.time()#从1970年到现在的计时秒数
end=time.time()-start#返回秒
print(start)
```

✓ 0.3s

```
comprehension  0.31369719999929657 seconds
1737898228.2422438
```

分析：

```
from timeit import Timer
#timeit 모듈은 Python에서 코드 실행 시간을 측정하기 위한 라이브러리입니다.
#Timer 클래스를 사용하면 특정 코드 블록의 실행 시간을 정확히 측정할 수 있습니다.
def test3():
    l = [i for i in range(1000)]

t3 = Timer("test3()", "from __main__ import test3")#1函数名，2import，因为这个Timer不一定在这里运行
#Timer는 두 가지 매개변수를 받습니다:"test3()": 실행하려는 함수나 코드(문자열로 전달).
#"from __main__ import test3": test3 함수가 현재 스크립트에 정의되어 있으므로 Timer가 이를 가져올 수 있도록 import합니다.
#주의: timeit.Timer는 입력 코드를 문자열로 받기 때문에, 외부 함수나 변수를 가져오려면 반드시 import 해야 합니다.

print("comprehension ",t3.timeit(number=10000), "seconds")#test3()执行10000次后，10000次总的执行时间
#t3.timeit(number=10000):
#test3() 함수를 10,000번 실행한 총 시간을 반환합니다.
#number=10000은 실행 횟수를 지정한 것으로, 10,000번 반복합니다.
```

```
import time
start=time.time()#从1970年到现在的计时秒数
end=time.time()-start#返回秒
#time 모듈은 현재 시간을 초 단위로 반환합니다.
#start = time.time(): 현재 시간을 기록.
#end = time.time() - start: 현재 시간에서 시작 시간을 뺀 값은 경과 시간입니다.
#이 부분은 timeit과는 다르게 직접 시간을 측정할 때 사용됩니다.
print(start)
timeit: 코드를 여러 번 반복 실행해 정확한 실행 시간을 계산.
time: 단순히 시작과 종료 시간 차이를 계산.
```

1. Unix 시간(Epoch Time)

- `time.time()` 은 **1970년 1월 1일 00:00:00 UTC**(세계 표준시)부터 현재까지의 경과 시간을 초 단위로 반환합니다.
- 이 시간을 "Unix 시간" 또는 "Epoch 시간"이라고 부릅니다.
- 반환 값은 **현재 시각까지의 총 초 단위 시간**입니다.

---

2. 왜 1970년 기준인가?

- 1970년 1월 1일은 **Unix 운영 체제의 시작 기준 날짜**입니다.
- 컴퓨터 시스템에서 시간을 표현할 때 기준점(Epoch)을 설정해 모든 시간을 상대적으로 계산하기 위해 1970년을 사용합니다.

数据结构

数据结构就是一个类的概念，数据结构有顺序表、链表、栈、队列、树

（算法复杂度只考虑的是运行的步骤，数据结构要与数据打交道。数据保存的方式不同决定了算法复杂度）

程序=数据结构+算法

算法是为了解决实际问题而设计的，而数据结构是算法需要处理的问题载体。

顺序表：

顺序表+链表=线性表：一根线串起来，两种表都是用来存储数据的。

顺序表的2个形式

1.计算机存储

计算机最小寻址单位是1字节，就是一个字节，才有一个地址，所有的地址都是统一大小0x27 4字节