# Ruchika PSA Class 2

Ruchika Shashidhara

NU 002245068

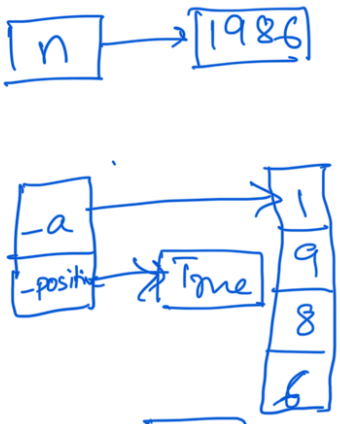
# PSA Class 2

Recap

DS

time: $\Theta(\log_{10} n)$ space: $\Theta(\log_{10} n)$

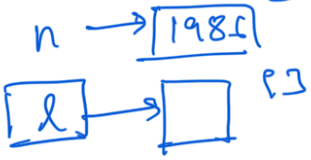
```
def __init__(self, n: "Python int" = 0):
    self.__positive = True
    if n < 0:
        self.positive = False
    self.__a = self.build(n)
```



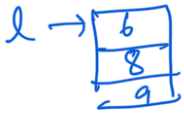

time: $\Theta(\log_{10} n)$ space: $\Theta(\log_{10} n)$

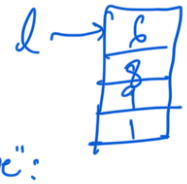
Algo

```
def build(self, n: "Python int")
    -> "list of int":
    if n < 0:
        n = -n
    l = []
    if n < 0:
        l.append(n % 10)
        n = n // 10
    self._reverse(l)
    return l
```






k steps:
- $n = n/10^0$
- $n/10 = n/10^1$
- $n/100 = n/10^2$
- ...
- $1 = n/10^k$

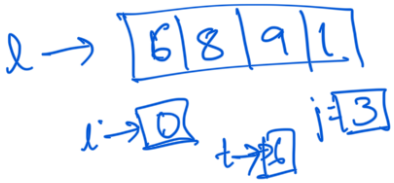
$1 = n/10^k$

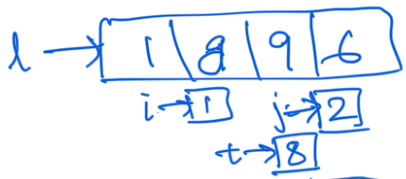
$k = \log_{10} n$

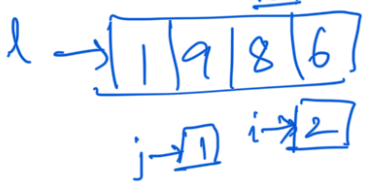
time: $\Theta(\log_{10} n)$ space: $\Theta(1)$

Algo

```
def _reverse(self, l: "list of int") -> "None":
    i = 0
    j = len(l) - 1
    while i < j
        t = l[i]
        l[i] = l[j]
        l[j] = t
        i = i + 1
        j = j - 1
```



Algorithm is an effective method for solving a problem expressed as a finite sequence of instructions → Mohammed al-Khowarizmi

Properties of Algorithm:

1. Precise
2. Unambiguous
3. Mechanical
4. Efficient
5. Correct

1. Is it correct?
2. How much time does it take, in terms of n? (Time complexity)
3. Can we do better? (Optimization)

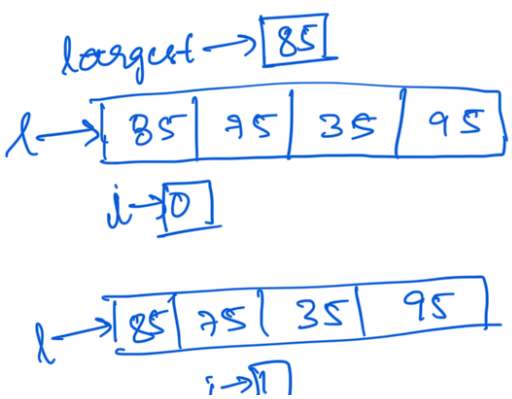
Complexity of Algorithms

Complexity – how much work we are doing?

Finding topper in unsorted list -- Theta(n)

Largest Number

```
Largest = l[0]
for( i : 1 to N-1) {
    if (l[i] > Largest)
        Largest = l[i]
```



largest = l[i]

}

return largest

largest → 85

l → 85 | 75 | 35 | 95

i → 2

largest → 85

l → 85 | 75 | 35 | 95

i → 3

largest → 95

Time complexity = $\Theta(n)$

n: size of list l

(i.e. exactly n operations)

Space complexity = $\Theta(1)$

if we sort:

asc.

sort(l)

return l[n-1]

desc.

sort(l, reverse = 0)

return l[0]

but time complexity of sort (best): $O(n \log n)$

so prev. algo was more efficient

<= Big O

<= Big O At the worst

Ex: If we have a book sold for $20, $50, $100

At the worst we can buy it for $100

>= Big Omega

>= Big $\Omega$ At the best (luck)

Ex: At the best we can buy it for $20

== Big Theta

== Big $\Theta$

If all books are priced at $50

we have to buy it at exactly $50

i.e. Best case == Worst case

Ex: Largest Number algo

$O(n) = n$

$\Omega(n) = n$

$\therefore \Theta(n) = n$

Finding a suitcase in an unsorted airport baggage carousel

$O(n) = n$ — unlucky — At the worst your bag is the last

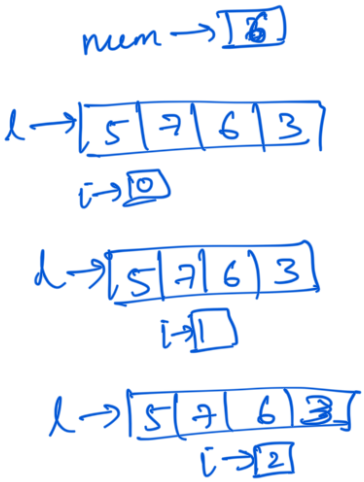
$\Omega(n) = 1$ — lucky — At the best your bag is the first

Here there is no $\Theta(n)$

$\therefore O(n) \neq \Omega(n)$

Finding a number in unsorted list -- O(n)

```
find ( l, num):
    for ( i = 0 → i = len(l) - 1):
        if ( l[i] == num):
            return i
    return -1
```




$O(n) = n = len(l)$

work being performed <= n steps

$\Omega(n) = 1$

Finding a fake coin (lesser weight) out of n coins using a balance that can measure coins at a time

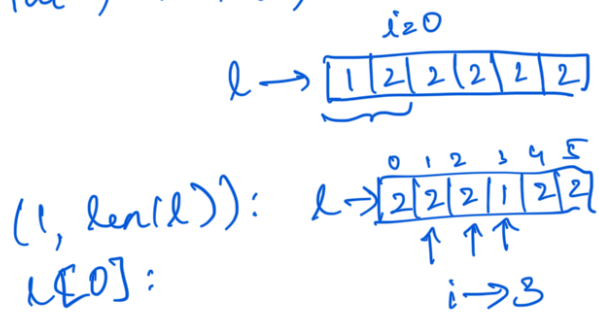
Every time we use the balance to measure any no. of coins, there is a cost, say $1

So we must try to reduce amount of work done / measures to reduce the cost of finding the fake coin

weight

Iterating through all the coins / list (weight of each coin)

```
find_fake_coin(l: "list of int") -> "int":
    if l[0] < l[i]:
        return 0
    for i in range(1, len(l)):
        if l[i] < l[0]:
            return i
    return -1
```

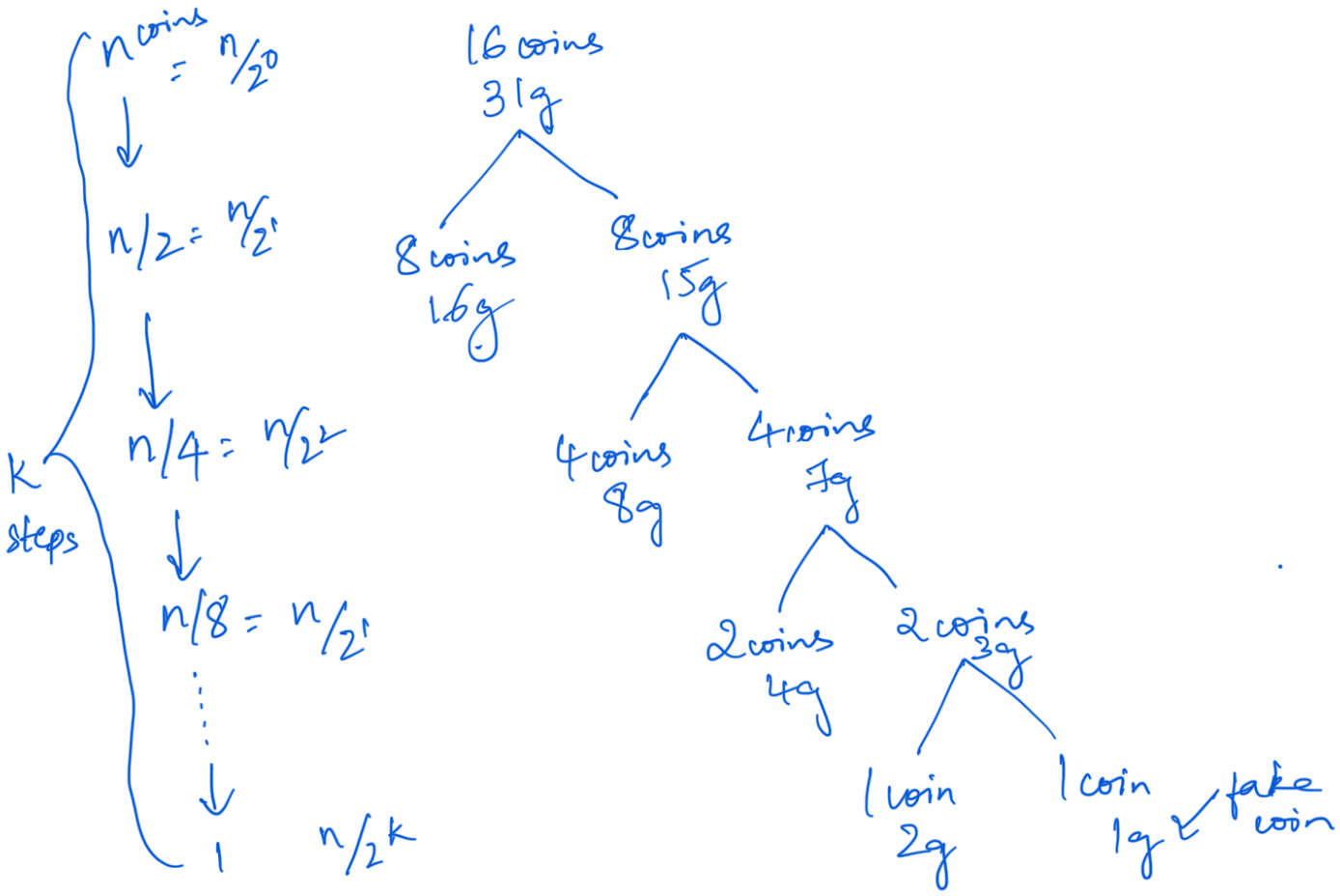



Time complexity: $O(n) = n$, $\Omega(n) = 1$

There exists no $\Theta(n)$

Divide and conquer

Optimizing it: Use Divide & Conquer




$n/2^k = 1$

Finding a fake coin from a list of n coins -- O(log n)

$2^k = n$

$k = \log_2 n$

$k \approx O(\log_2 n)$

i.e. how many times I need to divide n by 2 to get 1

Here, time complexity $= \theta(\log_2 n)$

```
find_fake_coin(l -> "list of int") -> "int":
    if len(l) == 1: return 0
    mid = len(l) // 2
    group1 = l[:mid]
    group2 = l[mid:]
    if sum(group1) == sum(group2):  # odd case
        return mid + find_fake_coin(group2)
    elif sum(group1) < sum(group2):
        return find_fake_coin(group1)
    else:
        return mid + find_fake_coin(group2)
```

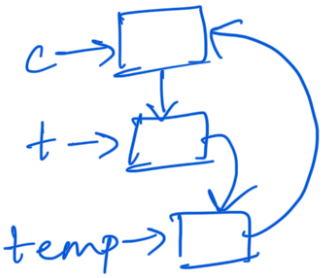
Time complexity $= \theta(\log n)$

Space complexity $= \theta(1)$

Independent of any steps

Ex: Swapping

```
temp = c
c = t
t = temp
```




What ever the size of c & t are,

total no. of steps = 3

∴ Time complexity $= \theta(1)$

Space complexity $= \theta(1)$

} Independent of n

Sum of n numbers

```
def sum(n: "int") → "int":
    s = 0
    for i in range n:
        s = s + i
    return s
```

n = 4

i → 1
s → 1
i → 2
s → 3
i → 3
s → 6
i → 4
ans: s → 10

Time complexity : $\Theta(n)$

Space complexity : $\Theta(1)$

Optimize it by Gauss's Algo:

$s = 1 + 2 + 3 + \ldots\ldots + n$

$s = n + n-1 + n-2 + \ldots\ldots 1$

$2s = n(n+1)$ $\quad s = \frac{n(n+1)}{2}$

```
def sum(n → "int") → "int":
    return n * (n+1) // 2
```

n → 4
s → 10

Time complexity = $\Theta(1)$
Space complexity = $\Theta(1)$
} Remains constant for any given n

Finding a number in a sorted list

```
def findnum(l, n):
    if l[0] == n:
        return 0
```

n → 4
l → 2 | 3 | 4 | 5
i=0 i=1 i=2

return

Time complexity
= O(n)
or Ω(1)

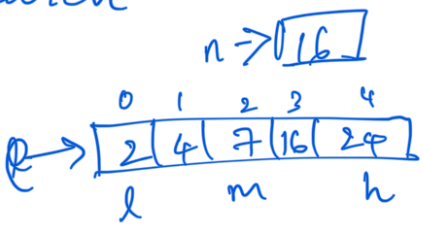
Space Complexity
= Θ(1)

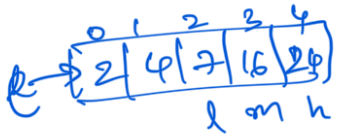
```
for i in range(len(L)):
    if L[i] == n:
        return i
return -1
```

Finding a number in sorted list -- O(log n)

We can optimize it by using divide & conquer: binary search

```
def findnum(L, n):
    l = 0
    h = len(L) - 1
    while l <= h:
        m = (l + h) // 2
        if L[m] == n:
            return m
        elif L[m] < n:
            l = m + 1
        else:
            h = m - 1
    return -1
```



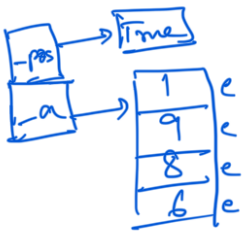



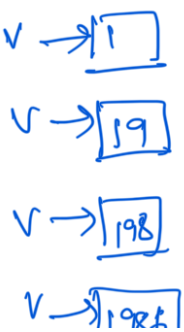

Time complexity
= O(log₂n)

Space complexity
= Θ(1)

```
def __int__(self) -> "Python int":
    v = 0
    for e in self._a:
        v = 10 * v + e
        if v == 0 or self._positive:
            return v
    return -v
```

Time complexity = Θ(log₁₀n)



Space complexity = $\Theta(1)$

useless adder : n m

```
def __add__(self, other):
    va = self.int()
    vb = other.int()
    vc = va + vb
    return Int(vc)
```

→ `__sub__` : `vc = va - vb`

→ `__mul__` : `vc = va * vb`

Time complexity = $\Theta(\log_{10} n)$

Space complexity = $\Theta(\log_{10} n)$

Ripple carry adder algorithm -- Theta(n)

Ripple Carry Adder

```
  ( 1  1  0     carry  → starts with 0
  2  5  6  9     self
+    9  9  9     other
-----------
  3  5  6  8     sum
-----------
```

Time complexity = $\Theta(\log_{10} n)$

Space Complexity = $\Theta(\log_{10} n)$

Ripple barrow subtraction algorithm -- Theta(n)

Ripple Carry Borrower

2959 - 12314

always subtract bigger digits from smaller digits

```
  ( 1  0  1     borrow
  1  2  3  14    self
-    2  9  5  9  other
-----------
  9  3  5  5
-----------
```

Time Complexity = $\Theta(\log_{10} n)$

Space Complexity = $\Theta(\log_{10} n)$

Relational operators < > == <= >=

# Relational operators

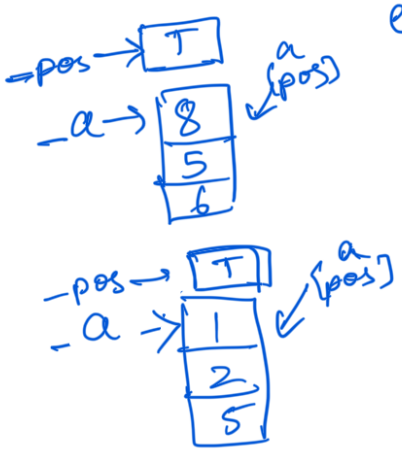
< <= > >= == !=

a = 856 b = 125

F F T T F T

< algorithm -- O(n)

```
def __lt__(self, other->"Int") -> "bool":
    if self._positive and not other.get_positive():
        return False
    elif not self._positive and other.get_positive():
        return True
    else:
        for pos in range a:
            if a[pos] < other[pos]
                return self._positive
            elif a[pos] > other[pos]
                return other._positive
    return False #equal
```




Time complexity = O(n)

Space Complexity = O(1)

Why we need to implement only <

We can derive other relational operators just from <

```
def __gt__(self, other):
    return other < self
```

a > b ≡ b < a

```
def __eq__(self, other):
    return not(self < other) and
```

a == b ≡ !(a<b)&&

```
                    not (other < self)
def __ne__ (self, other):
    return not (self == other)
def __le__ (self, other):
    return not (other < self)
def __ge__ (self, other):
    return not (self < other)
```

!(b<a)

a != b
≡
!(!(a<b) && !(b<a))

a <= b
≡ !(b<a)

a >= b
≡ !(a<b)

(a == b) ≡ !(a<b) && !(b<a)

| a | b | a == b | a < b | !(a<b) | a > b | !(a>b) |
|---|---|---|---|---|---|---|
| 0 | 0 | T | F | T | F | T |
| 0 | 1 | F | T | F | F | T |
| 1 | 0 | F | F | T | T | F |
| 1 | 1 | T | F | T | F | T |

```
def scum(self, a: int, b: int) -> int:
    ⋮
Exam(ma, mb, madd, mrelation)
    ⋮
```

fill with ops
< <= >= == !=

ma, mb: should be unchanged

madd, mrelation: should be updated in Exam class