AND → & → ampersand
OR → | → pipe
XOR → ^ → caret
NOT → ~ → tilde

TC
↳ O(1)
SC
↳ O(1)

pragy@scaler.com
7351769231

Data Types → Numeric

byte, int, long, float, double

store an integer in memory
↳ allocate/reserve some space in RAM
how much space?

k bits → 2^k different values.

Number falls in some range.

```
byte x = 200;
short y = 60,000;
```
} C++/Java

Range	# of bits Needed.	
[0 ... 255]	8 bits ⇒ byte	$2^8 = 256$
[0 ... 65,535]	16 bits ⇒ short 2 bytes	$2^{16} = 65,536$
[0 .. $2^{32}-1$]	32 bits ⇒ int 4 bytes	$2^{32} \approx 4$ billion
[0 .. $2^{64}-1$]	64 bits ⇒ long 8 bytes	$2^{64} \approx 16$ billion billion.

Unsigned types (non-negative values)

In python Integers are unbounded

in C++ → 2^{1000} → overflow

$2^{1000} >> 2^{64}-1$

in Python → $2^{**}1000$ → Power

Signed Types

$2^8 = 256$ values
0 – 256

Data Type	Size	Range
byte	8 bit	$[-128, 127]$ $[-2^{8-1}, 2^{8-1}-1]$
short	16 bit	$[-2^{16-1}, 2^{16-1}-1]$
int	32 bit	$[-2^{32-1}, 2^{32-1}-1]$
long	64 bit	$[-2^{64-1}, 2^{64-1}-1]$
T	k bit	$[-2^{k-1}, 2^{k-1}-1]$

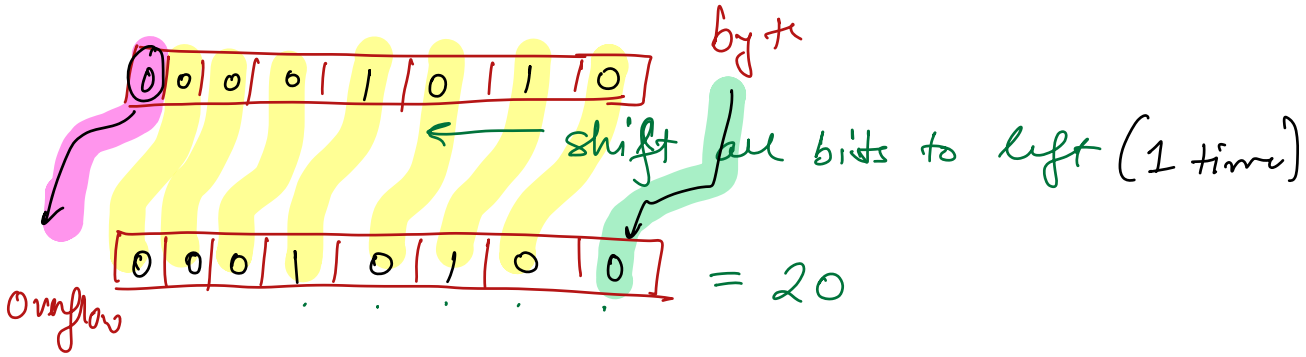
Unsigned
0 – 255
-128 — 127
-128 → -1 [128]
0 — 127 [128]
256

---

fractions
- float
- double

+ve numbers → ignoring the # of bits

A = 10 binary → 1010



Left shift operator → unsigned

A = 10

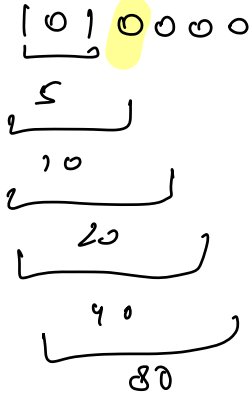
A << 1

print(A) = 20

A = 5

A << 4

print(A) = 80



N << k } left shift N by k times
drop off the topmost bits

& introduce 0's at the end.

8 bits

byte a = 10;

a = a << 100;

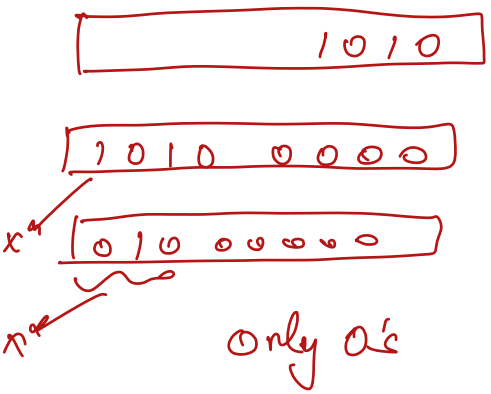
print (a) # 0

C++

$N << k$ no overflow

$N \cdot 2^k$

$10 \cdot 2^{100}$



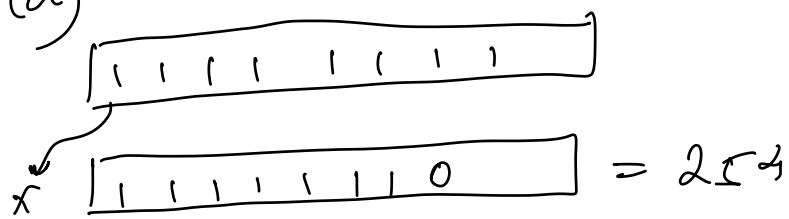
in python → $10 \cdot 2^{100}$

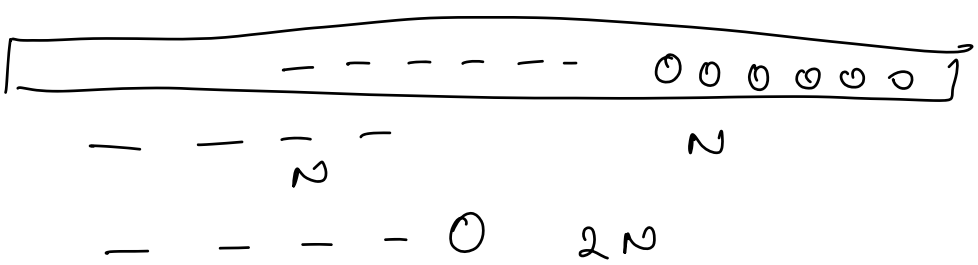
---

byte a = 255;

a = a << 1;

print(a) → 254





a	b	c	d

$a \cdot 2^3 + b \cdot 2^2 + c \cdot 2^1 + d \cdot 2^0$ (1)

$= N$

a	b	c	d	0

$a \cdot 2^4 + b \cdot 2^3 + c \cdot 2^2 + d \cdot 2^1 + 0 \cdot 2^0$ (1)

$2(a \cdot 2^3 + b \cdot 2^2 + c \cdot 2^1 + d \cdot 2^0) + 0$

$2 \cdot N + 0 = 2N.$

$a \cdot 2^{3+k} + b \cdot 2^{2+k} + c \cdot 2^{1+k} + d \cdot 2^k + 0 \cdot 2^{k+} + 0.2^{k-2} - 0.2^0$

$2^k(N) + 0 = N \cdot 2^k$

---

a

$a << n = a \cdot 2^n$ assuming no overflow

if there is overflow, the above will not hold true.

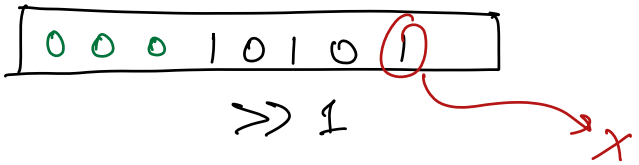
$15 << 2 \quad = \quad 15 \times 2^2 = 15 \times 4 = 60$

Right Shift operator.

$a << k$

$a >> k$

$a = (10101)_2 = (21)_{10}$



$a >> k \equiv \left\lfloor \frac{a}{2^k} \right\rfloor$

(integer division)

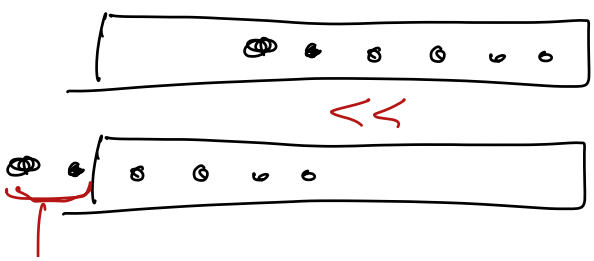
$15 >> 1 \Rightarrow 7$

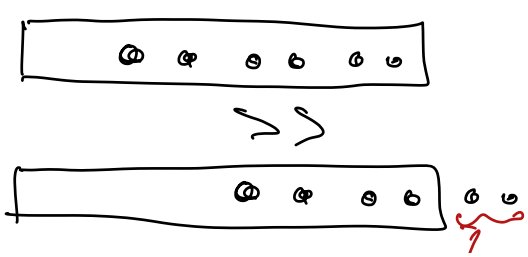
$16 >> 1 \Rightarrow 8$

$50 >> 2 \rightarrow \left\lfloor \frac{50}{4} \right\rfloor = \lfloor 12.5 \rfloor = 12$

$10 >> 4 \rightarrow \left\lfloor \frac{10}{2^4} \right\rfloor = \left\lfloor \frac{10}{16} \right\rfloor = \lfloor 0.\text{---} \rfloor$

$= 0$





overflow

NOT overflow

$a >> \text{ceil}(\log(a,2)) = 0$

$\frac{a}{2^{\lceil \log_2 a \rceil}} = \frac{a}{>a} = 0$

`a = 10;` // $a = (1010)_2$

7,8

a = 789

→ High level language

→ decimal to binary

→ machine code

In python `<<` has no overflow

`>>` will drop the LSB it

---

given a Number N (non-negative integer)

check if the k^{th} bit of N is set or not?

↳ is it 1

N = 50

k = 3

unset

⇒

6	5	4	3	2	1
1	1	0	0	1	0
32	16	8	4	2	1

50 = 32 + 16 + 2

50 − 32 = 18

18 − 16 = 2

1 → set

0 → unset

N = 50

k = 3

1 1 0 [0] 1 0

RAM

---

① divide to extract each bit & check the 3rd bit

```
n = //
k = //
while n>0 and k >0:
    n = n//2
    k = k-1
if n%2 == 0:
    return (unset)
else
    return (set)
```

TC
$= O(\min(k, \lg_2(N)))$

SC
$= O(1)$

$\lceil a \rceil \rightarrow$ ceil

$\lceil 2.6 \rceil = 3$

$\lceil 2.1 \rceil = 3$

$\lceil 2 \rceil = 2$

if a No is N

it needs $\lceil \lg_2(N+1) \rceil$ bits to be stored.

---

k bits $\rightarrow$ 2^k different values $\rightarrow$ $0 -- 2^k - 1$

unsigned

N

to represent N, what is the value of k ?

k must be the smallest no such that

$2^k - 1 \geq N$

if N = 1024

k = 3 ? $\times$ $2^3 - 1 = 7$

$2^k - 1 \geq 1024$

$$2^k \geq N+1$$

$$\lg_2(2^k) \geq \lg_2(N+1)$$

$$k \cdot \lg_2 2 \geq \lg_2(N+1)$$

$$k \geq \lg_2(N+1)$$

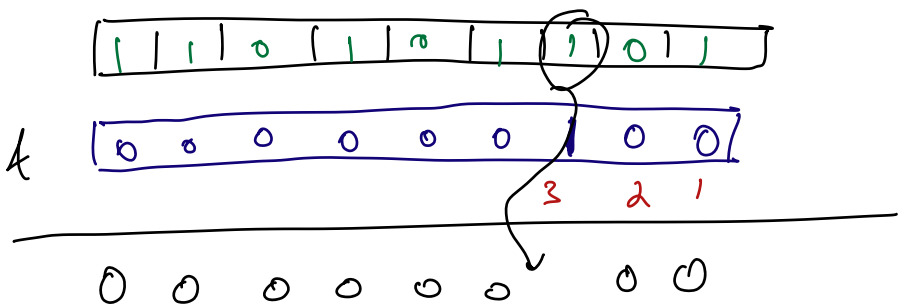
k is the smallest no $\geq \lg_2(N+1)$

$$k = \lceil \lg_2(N+1) \rceil$$

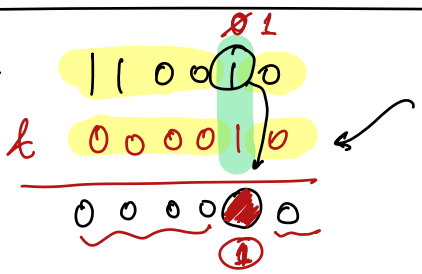
---

② check the k^{th} bit

$k = 3$



$N = 50 = 110010$



$nk1 = n$

$1 << (k-1)$

$\boxed{1}0$
2 1

$N \;\&\; (1 << (k-1))$

if (N & (1<<(k-1))) == 0 → check k^{th} bit

return 'unset'

else

return 'set'

$2^{**}(k-1)$ ✓

bitwise operators are very fast → Take only 1 clock cycle.

+ → 5

* → 20

& | → 1

---

a = 203 →  binary

---

[N & (1<<(k-1))] → truthy → true

→ falsey (0) → false

≡

[(N >> (k-1)) % 2] ✓

```
def checkBit(N, k):
    return (N >> (k-1)) % 2
```

checkBit(14, 3) → true

14 = 8 + 4 + 2

1 1 1 0

↓

set

$$\left(14 >> (3-1)\right) \% 2$$

2

14 >> 2 = 3

3 % 2 = 1

checkBit(24, 2)

[N & (1 << (k-1))]

24 & (1 << (2-1))

1

24 & (2)

24 & 2

0

24 = 16 + 8

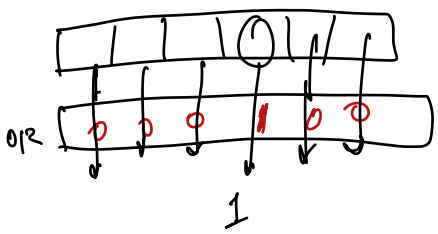
1 1 0 0 0

↓

0

checkBit

Set Bit	unset Bit	Toggle Bit
N, k return (N | [1<<(k-1)])	return N & (~(1<<(k-1)))	N ^ (1 << (k-1))



^ → XOR operator

** → power (python)

---

checkbit(24, 2) → false $(11000)_2$

setbit(24, 2) = 26 $(11010)_2$

N + (1 << (k-1))

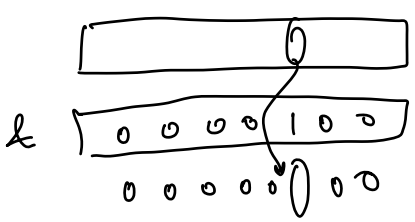
≠ N | (1 << (k-1))

N = 24

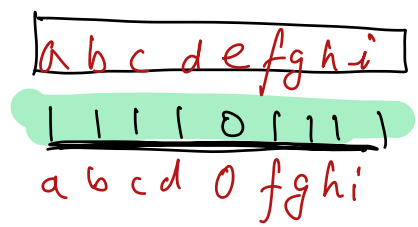
00010

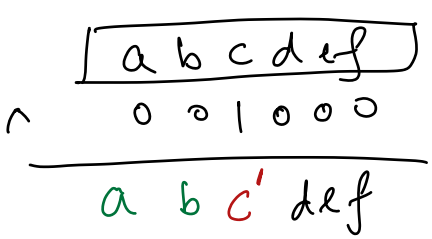
11010 + 10 = 11100

11010 + 10 = 11010



XOR





toggle

unset

x ^ 0 = x

x ^ 1 = ux

0 0 0 1 0 0 0

N & (~(1<<(k-1)))

bit ^ 0 = bit

bit ^ 1 = bit NOT operation.

toggle

a b c d e f g h i j

^ 0 0 0 0 1 0 0 0 0

a b c d e' f g h i j

~2

00000010

1

11111101

= -3

253

255 − 2 = 253

−127 −128 → byte

>

char

int

long

a b c d e f g h i j

k 1 1 1 1 0 1 1 1 1

a b c d 0 f g h i j

k k-1

~(0 0 0 0 1 0 0 0 0)

N & (~(1<<(k-1)))

~N

N ^ (~0)

16 bits

8 bits

16 bits

^

11111

~N

given N

Count how many bits are set?

N = 50 → 110010

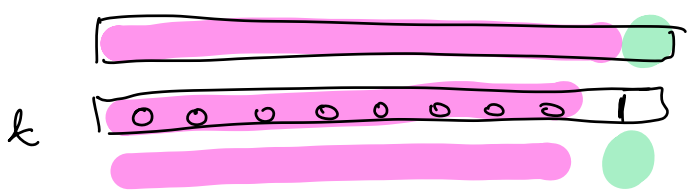
```
c = 0
while (n > 0) {
    c += (n % 2)
    n //= 2
}
return c;
```

TC $O(\lg_2 N)$

SC $O(1)$

```
c = 0
while (n > 0) {
    c += (n & 1)      → extract last bit  ≡ (n % 2)
    n = n >> 1        ≡ n = n // 2
}
return c;
```

$n \,\&\, 1 \equiv (n \,\%\, 2)$



Convert no to binary

```
bin = []
while n > 0:
    bit = n % 2
    n //= 2
    bin.append(bit)
return reverse(bin)
```

$O(\lg_2 N)$ TC

$N \rightarrow [\quad]$

$\lceil \lg_2 (N+1) \rceil$ bits

$O(\lg_2 N)$

N

$N/2$ $\quad \frac{N}{2^1}$ → 1st step

$N/4$ $\quad N/2^2$ → 2nd step

$N/8$ $\quad N/2^3$

→ i^{th} step

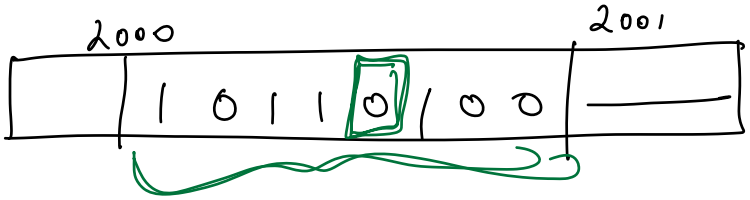
$\frac{N}{2^i} = 0$?

$2^i > N$

$i > \lg_2 N$

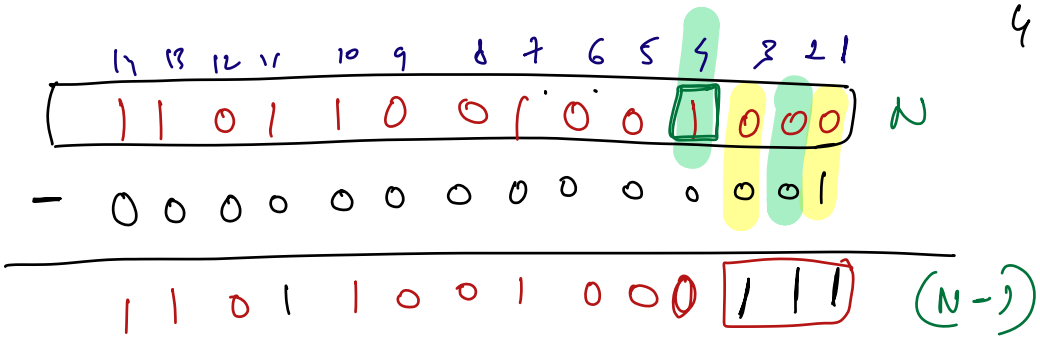
RAM is byte addressable

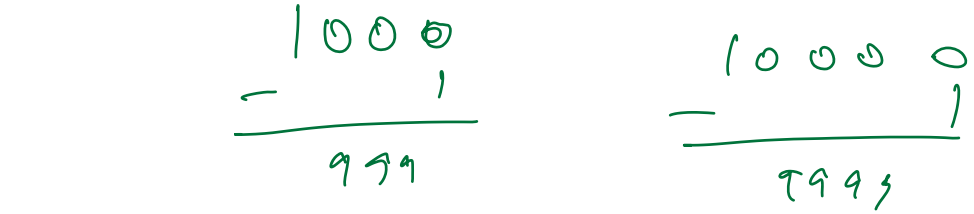
↳ work with units of 8 bits

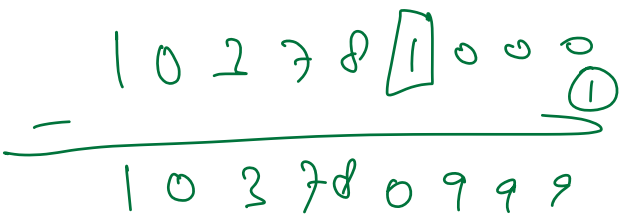


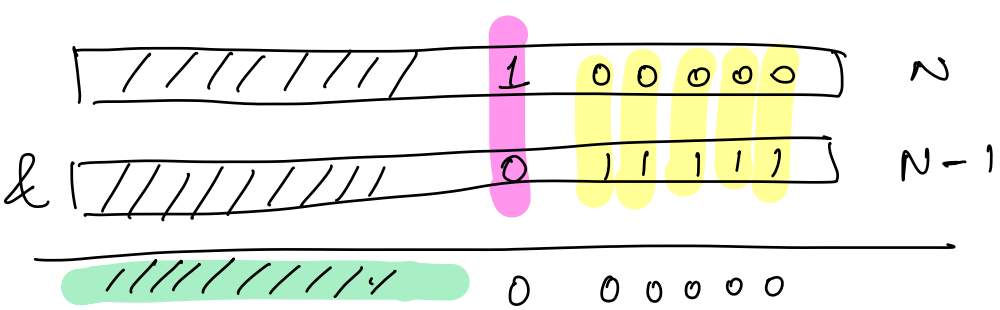
$N \;\&\; (N-1)$

first set bit?









$[N \;\&\; (N-1)]$ → unset the first set bit

1 0 1 0 1 1 0 0 = 172

128 64 32 16 8 4 2 1

128
+ 32
+ 8
+ 4
172

	1	0	1	0	1	1	0	0	= N	172
&	1	0	1	0	1	0	1	1		171
	1	0	1	0	1	0	0	0		168

N-1

(N & (N-1)) unset the first set bit from right

High frequency Trading
Gaming / Cuda

```
c = 0
while (n > 0) {
    n = n & (n-1);   → unsetting the last set bit
    c++;
}
```

TC
O(k)
↳ # of set bits

vs → O($\log_2(N)$)
↳ usual.
256? → 1

---

usually → worst case
avg
best } merit

```
  1 0 0 0 0 0 0 0 0 0     256
& 0 1 1 1 1 1 1 1 1 1     256-1
-------------------
  0 0 0 - - - - - - -
```

---

don't show off

code → not for writing it
↳ being read

Shift bits / Data types & ranges / Count/ Set/ Unset/ toggle

---

n

```
while (n > 0) {
    ≡
    n = n/2
    ≡
}
```

$O(\lg_2 N)$

step	value	
0	n	1024
1	$n/2$	512
2	$n/4$	256
3	$n/8$	128
⋮		
i	$n/2^i$	
⋮		
k	$(n/2^k) = 0$	

k =)

$$\frac{n}{2^k} = 0$$

$$n \leq 2^k$$

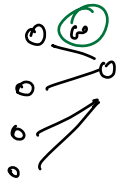
$$\log_2(n) < \log_2(2^k) \quad (k)$$

$$\log_2(n) < k$$

k atleast $\log_2(n) + 1$

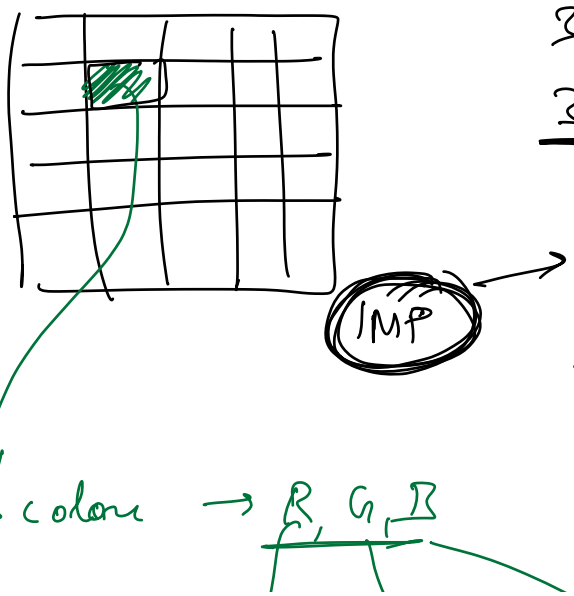
$$k = \lceil \log_2(N) \rceil$$

NN



An → all no appear twice except for 1

2 → XOR to entire array

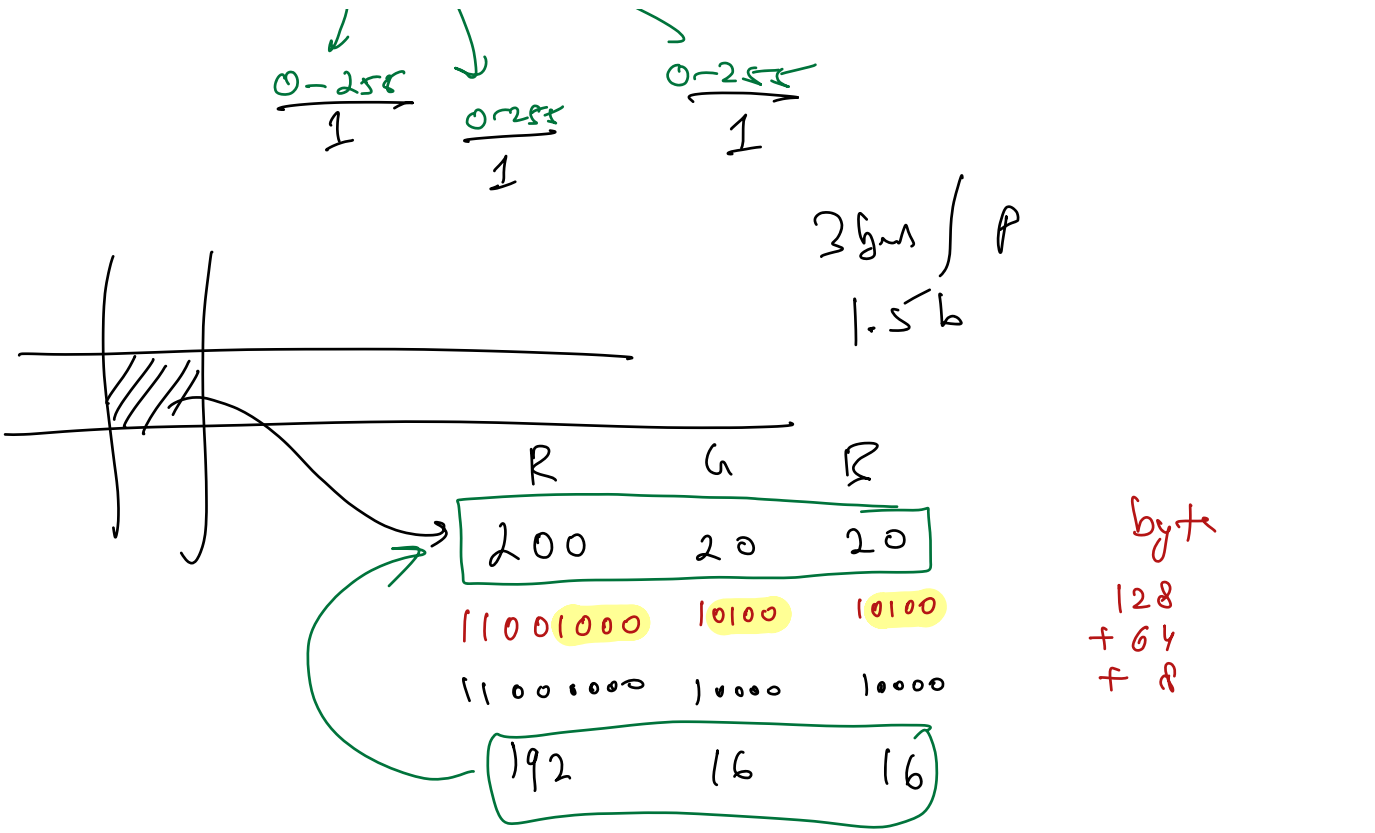


3 byte / pixel

3 MB of space

10^6 pixels

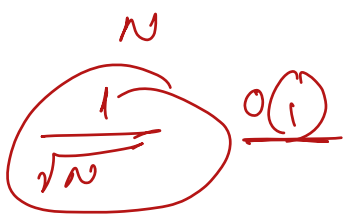
3 colors → R, G, B



Quake 3 fast inverse square root

↳ Bit Manip

↳ Newton Raphson



```
n = int(input())
index = 0
nums = []
while n > 0:
    for i in input().split():
        nums.append([index, int(i)])
        index = index + 1
    n = n - 1
```

li → bad name

print (nums)

return 0

Input

5

10	11	6	7	3	2
1	2	3	4	5	6

nums = [ ]

n = ~~5~~ 4

i = 0

index = 6

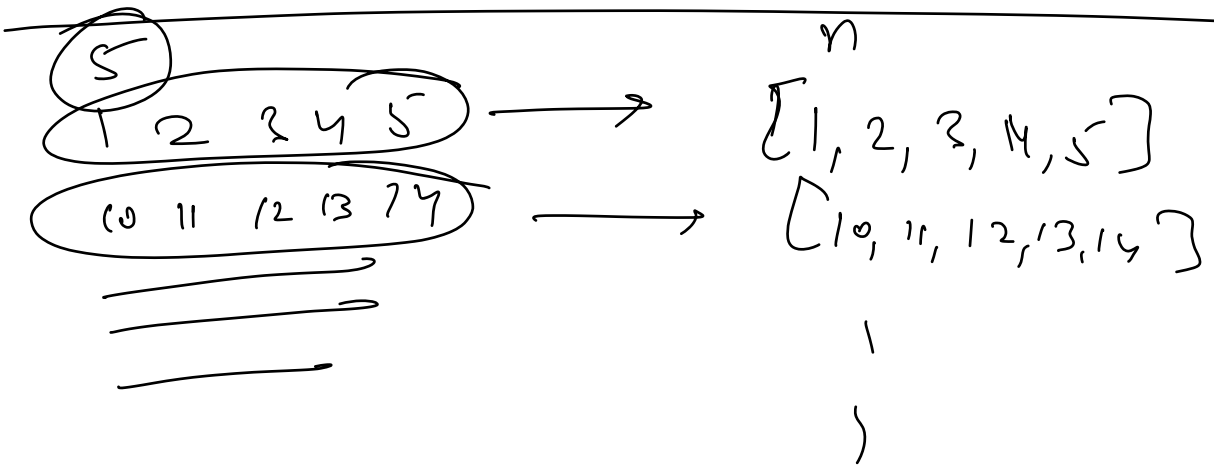
nums = [[0, 10] [1, 11] [~~2~~, 6] [3, 7] [4, 3] [5, 2]]

```
n = int(input())
index = 0
nums = []
while n > 0:
    row = []
    for i in input().split():
        row.append(i)
    nums.append(row)
print(nums)
```

list comprehension

```
[[int(y) for y in input().split()]
```

return 0



N → print it in base 8

↳ pri in base 2

N → base 13

↳ ' ____ ' → expect a no in base 13

↳ (____) → reason in base 7?

'faf0' → base 16

```
def change_base(original, b, b2):
    for c in reversed(original):
        value = hex_to_int(c)
```

assume b2 < b

0 – 0
9 – 9
a – 10
f – 15

atto → 10^{-18}

femto — 10^{-15}
pico — 10^{-12}
nano → 10^{-9}
micro → 10^{-6}
milli → 10^{-3}
centi → 10^{-2}
deci → 1/10
unit → 1
deca → 10
hecta → 100
Kilo → 1000
Mega → 10^{6}
Giga → 10^{9}
Tera → 10^{12}
Peta → 10^{15}
Exa → 10^{18}

mac

Kilo Byte ⇒ 8 bits

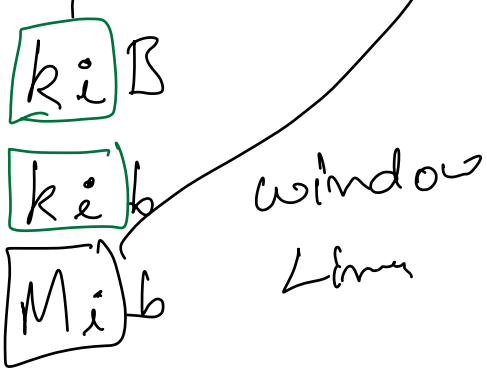
kB

kilo bit

kb

1 Gbps → 10^{9} bits per sec

1,024 = 2^{10}
2^{20}
2^{30}
2^{40}
2^{50}
2^{60}



window
Linux

1 ↳ 125 MB/s

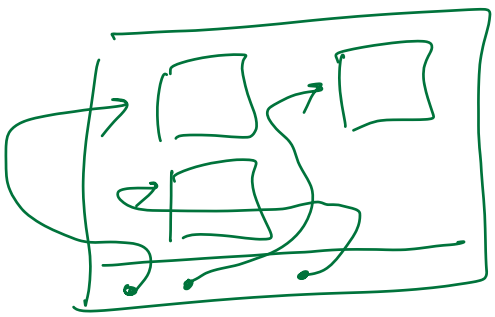
---

class Foo:
    # an object of this class takes 1000 bytes

non-primitive objects

objs = [Foo(), Foo(), Foo()]
0 1 2

size of objs? [64 bytes + 3]
↳ objs = 24 bytes



---

2 ** 1,000,000]
↳ 1 Mbits of memory

print(2 ** 1000000) → work → a long time to print

a = 2 ** 1000000 ↵ within a fraction of a sec

GPT3 → 180 B params

float → 32 bits
4 bytes

180 x 4 GB

600 GB

mobile phone → 1 GB

Python → int ← byte/short/int/long

→ float
→ complex
→ bool
→ str

32 bit int

4 GB

16 GB

Big int → unbounded

given an array of integers

↳ every no appears exactly twice except for 1 no that appears only once.

A = [1 2 2 8 3 6 7 8 2 6 1]

identify the no that appears only once

Brute force → for each no

count how many times it appears

↳ 1 → return

```
for n in A:
    c = 0
    for x in A:
        if n == x:
            c += 1
    if c == 1:
        return n
```

$c = \phi \vee 2$

$O(n^2)$ time

$O(1)$ space

Hashmap | dictionary in Python → $O(n)$ time

$O(n)$ space

$x \wedge x = 0$

$y \wedge 0 = y$

```
xor = 0
for i in range(len(A)):
    xor ^= A[i]
```

$O(n)$ TC

$O(1)$ SC

Finding xor

Unset k^{th} bit

bit & 1 = bit

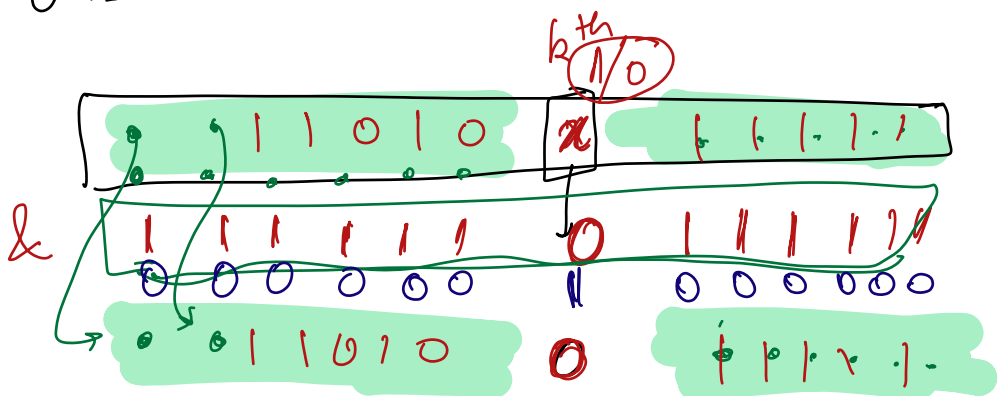
0 & 1 = 0

1 & 1 = 1

bit & 0 = 0

0 & 0 = 0

1 & 0 = 0



$N \;\&\; (\sim [1 << (k-1)])$

Variables are always stored in stack
in compiled languages

objects → heap

---

in python → heap
except for the call stack

```
class Foo:
    def __init__(self, n):
        self.n = n

a = Foo(10)
a.n += 20
```

check if a.__getitem__ exists

object.__getitem__(a, 'n')

if class a has any descriptor
if a exists in Obj.__dict__ ?

(0) int

int f = 20

--add--(int)

a = a+1 → 100s of things

---

on avg python is 100 times slower than C++

↳ don't program python

↳ program the python interpreter

12 noon – 4 pm 6 pm +

C++ → 1s TLE

R → 3-5 TLE

Balanced Binary Search Tree

(N >> (k-1)) % 2

2+1

N = N >> (k-1)

---

3 blue 1 brown → investing math