1. Flip
2. Nth magic number
3. subarray OR
4. Tower of Hanoi
5. strange Equality

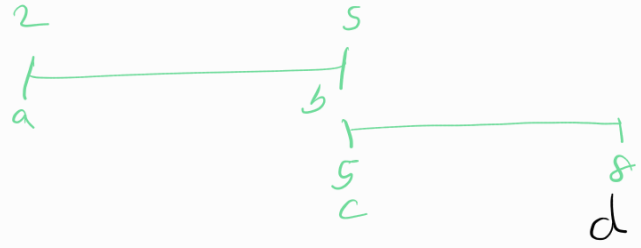
Flip:

You are given a binary string A(i.e., with characters 0 and 1) consisting of characters A1, A2, ..., AN. In a single operation, you can choose two indices, L and R, such that $1 \le L \le R \le N$ and flip the characters AL, AL+1, ..., AR. By flipping, we mean changing character 0 to 1 and vice-versa.

Your aim is to perform ATMOST one operation such that in the final string number of 1s is maximized.

If you don't want to perform the operation, return an empty array. Else, return an array consisting of two elements denoting L and R. If there are multiple solutions, return the lexicographically smallest pair of L and R.

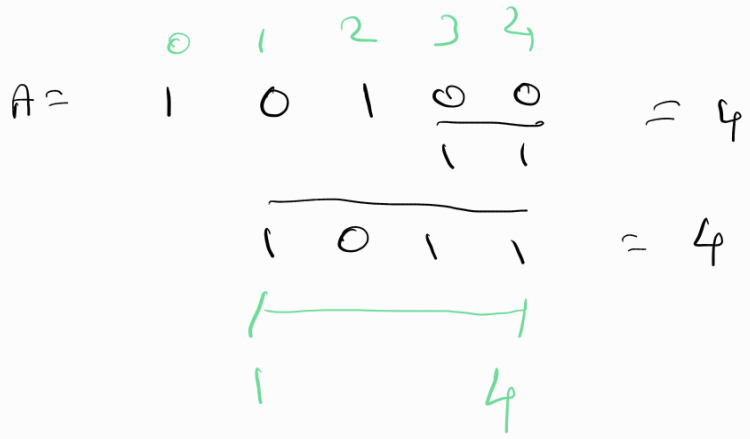
NOTE: Pair (a, b) is lexicographically smaller than pair (c, d) if a < c or, if a == c and b < d.



(a, b) ✓

if a < c

elif a == c and b < d

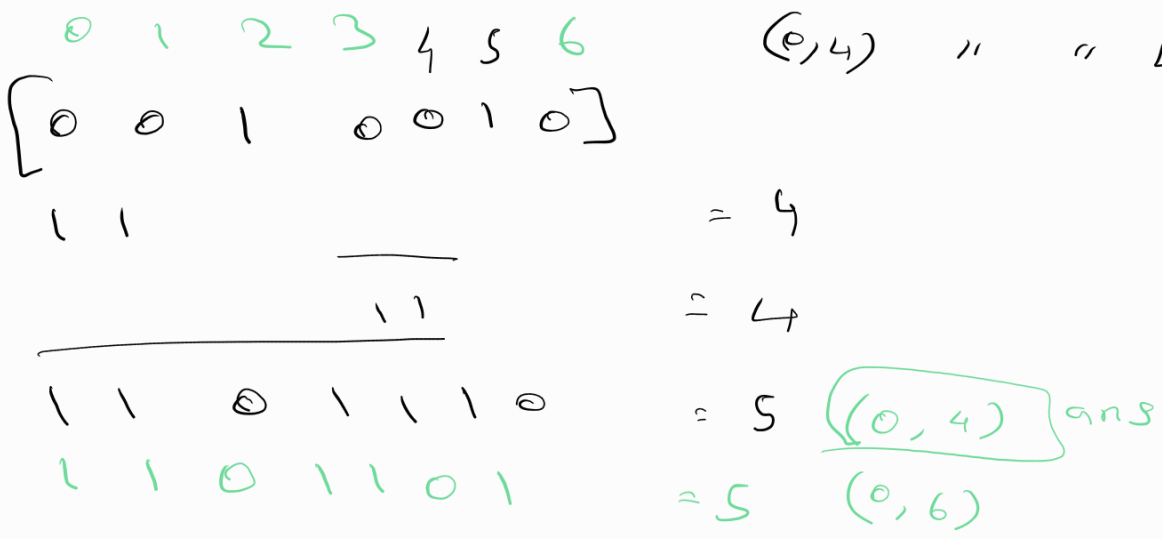


(1 4) → Lexicographically smaller

(3 4)

(0,6) subarray has 5 0's

(0,4) " " 4 0's



BF: TC: $O(N^3)$ SC: $O(1)$

→ Create all subarray and calculate the max ones possible.

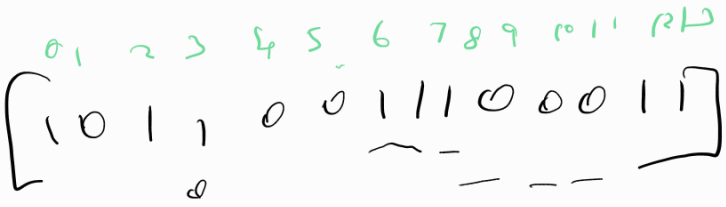
→ return the indexes.

A = [1 1 1 1 1]

(1) No need to switch

(2) return empty array ans = [ ]

Tip: ~~Flip subarray has max count of 0s~~



net gains of ones

0s 1s

6 < 8

(1, 11) 9 1's

(4, 11) 11 1's

⇒ Flip subarray where we get net gain of 1's maximum

↓

means we need subarray with maximum sum

↓

To solve this

Kadane's algorithm.

↳ if the sum is positive keep the sum carry forwarded.

arr [1 0 1 0 0]

↓ ↓ ↓ ↓ ↓

-1 +1 -1 +1 +1

sum = ~~0~~ ~~-1~~ ~~0~~ ~~-1~~ 0

↓

→ we can not carry forward negative value

keep it 0

Code:

```
def flip(A):
    N = len(A)
    csum, l, r, max_sum, nsi = 0, 0, 0, 0, 0   → integers
    for i in range(N):
        if A[i] == '0':
            csum += 1
        else:
            csum -= 1
        # update max_sum and l, r
        if csum > max_sum:
            max_sum = csum
            l = nsi
            r = i
        # reset csum if < 0
        if csum < 0:
            nsi = i+1
            csum = 0
    if max_sum == 0:
        # If we do not have possible subarrays
        return []
    else: return [l, r]
```

TC = O(N)

SC: O(1)

Nth magic Number:

Nth Magic Number →

Given an integer A, find and return the Ath magic number.

A magic number is defined as a number that can be expressed as a power of 5 or a sum of unique powers of 5.

First few magic numbers are 5, 25, 30(5 + 25), 125, 130(125 + 5), ...

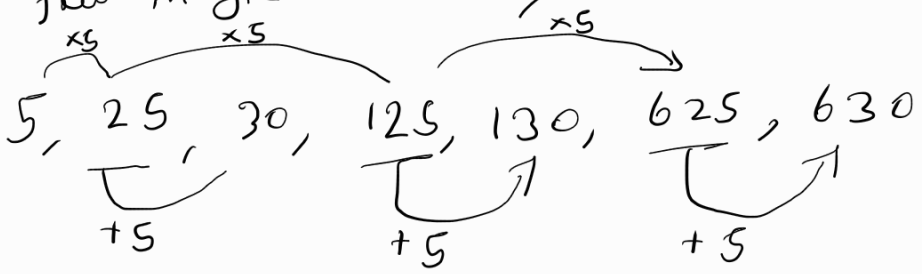
→ int A

→ Find Ath magic number.

→ magic number is, number that can be expressed as a power of 5 (or)

↳ Sum of unique powers of 5

→ First few magic numbers



1	5	5^1	→ 1×5^1	= 1
2	25	5^2	→ $1\times5^2+0\times5^1$	= 10
3	30	5^2+5^1	→ $1\times5^2+1\times5^1$	= 11
4	125	5^3	→ $1\times5^3+0\times5^2+0\times5^1$	= 100
5	130	5^3+5^1	→ $1\times5^3+0\times5^2+1\times5^1$	= 101
6	150	5^4	→ $1\times5^3+1\times5^2+0\times5^1$	= 110
7	155	5^4+5^1	→ $1\times5^3+1\times5^2+1\times5^1$	= 111
8	625		→ $1\times5^4+0\times5^3+0\times5^2+0\times5^1$	= 1000
9	630			

10 = 650

⇒ binary number of 1, 2, 3 ...

⇒ instead of multiplying with powers of 2
multiply with powers of 5

9 → 1 0 0 1

4 3 2 1 → 1 based indexing.

5^4 5^3 5^2 5^1

625 0 0 5 ⇒ 650

BF?

Find A^{th} magical number!

```
ans = 0 , power = 5, i = 1
while (A > 0):
    # find last bit is set or not
    if (A & 1) > 0:
        # bit is set
        ans += (5 ** i)
    A = (A >> 1) = A/2
    i += 1
return ans
```

32 → A bits

TC: (32)
every time we devid a by 2... ($\log_2 A$)

SC: O(1)

1 0 0 1

0 1 0 0

0 1 0 0

4 3 2 1

1 ϕ X ϕ

A = 10

i = 4

ans = 25 650

0 1 0

0 0 0 1

0 0 0

25 × 5

125 × 5

625

25

A = 1

4 3 2 1
0 0 0 1

5

5' = 5

Subarray OR

Subarray OR .

You are given an array of integers A of size N.

The value of a subarray is defined as BITWISE OR of all elements in it.

Return the sum of value of all subarrays of A % 109 + 7.

→ given int[ ] A
→ find value for each subarray
→ and return sum of all values.

value(0,1) = A[0] | A[1]

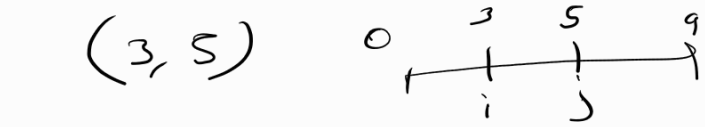
BF :

1. going through all subarray of A. → N^2
2. Finde vale of each subaray → N
3. maintaining ans, which is sum of all values.

TC : $O(N^3)$
SC : Constant

A = [1, 2, 3]

(3, 5)




```
def ORvalue(A, i, j):
    ans = A[k]
    for k in range(i+1, (j-i+1)+1):
        ans = ans | A[k]
    return ans
```

```
n = len(A), ans = 0
for i in range(n):
    for j in range(i, n):
        # find value for subarray(i, j)
        value = ORvalue(A, i, j)
        # adding it to sum
        ans += value
return ans
```

TC: $O(N^3)$

SC: $O(1)$

Optimized using carry forward $O(N^2)$

1 →

1 2

1 2 3

2 →

2 3

3 →

0 0 0 1

0 0 1 0

0 0 1 1

0 0 1 1

0 0 1 1

0 1 0 0

0 1 1 1

1	0 0 0 1	OR
2	0 0 1 0	1 0
3	0 0 1 1	1 1
4	0 1 0 0	1 1
5	0 1 0 1	

$$\frac{N(N+1)}{2} = \frac{3(3+1)}{2} = 6$$

optimized

OR's Property

if any bit is 1 output is 1

```
  0
  0
  1
  0
  0
-----
 1
```

output = 1

0th bit → Count the subarray that has 0 in 0th bit

A[0] → 0 → 1

A[1] → 1

A[2] → 0

A[3] → 0

A[4] → 0

A[5] → 1

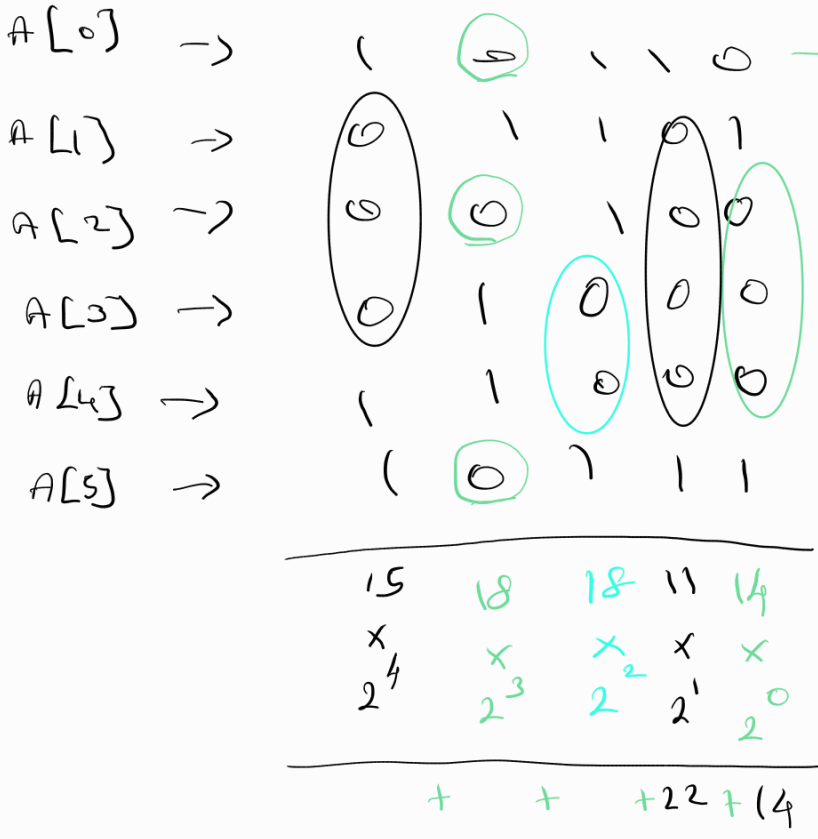
$\frac{3 \times (3+1)}{2} = 6$ → 7 subarray has 0th bit unset

Num of subarray that has 1 in 0th bit = num of total sub-array − num of subarray that has 0 in ith bit.

$= 21 - 7$

0th bit $= 14$

$= 14 \times 2^0$



$\frac{21-7}{14}$

$\frac{4 \times (4+1)}{2} = 10$

$21 - 10$

11

Code:

```
N, ans, modulo = len(A), 0, (10**9+7)

for i in range(32):
    count=0, subarrwithzero=0
    for j in range(N)
        # for every element find num of subarrs that has
          ith bit zero.
        if (A[j] & (1<<i)) ==0:
            # ith bit un-set
            count +=1
        else:
            # update subarrs with zeros
            subarrwithzero = (count * (count +1))//2
            # reset zeros count
            count =0
        # update subarrs with zeros after the loop complete
        subarrwithzero = (count * (count +1))//2
        # num of subarray - count
        value = subArrCount - subarrwithzero
        ans += (value * (1<<i)) % modulo
    return ans % modulo
```

TC: (32 * N)
SC: (1)

strange Equality:

Strange Equality →

Given an integer A.
Two numbers, X and Y, are defined as follows:

X is the greatest number smaller than A such that the XOR sum of X and A is the same as the sum of X and A.
Y is the smallest number greater than A, such that the XOR sum of Y and A is the same as the sum of Y and A.
Find and return the XOR of X and Y.

NOTE 1: XOR of X and Y is defined as X ^ Y where '^' is the BITWISE XOR operator.

NOTE 2: Your code will be run against a maximum of 100000 Test Cases.

→ given int A

→ x greast < A, $x \wedge A = x + A$

→ y smallest number > A, such that

$y \wedge A = y + A$

→ Find and return XOR of x and y

XOR (^)

a	b	output
1	1	0
1	0	1
0	1	1
0	0	0

→ x greast < A, $\boxed{x \wedge A = x + A}$

$$A = 10110 = 2^5 + 2^3 + 2^2 = 44$$

$$x = 01001 = 2^4 + 2^1 + 2^0 = 19$$

$$11111 = 2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 = 63$$

1. x can not be same as A,
2. and it has to be smaller than A.

Code:

```
x = 0, i = 0
while(A > 0):
    if (A & 1) == 0: # bit is un-set
        x = x + (1 << i)   (x | (1 << i))
    i += 1
    A = A >> 1   => A/2
```

i currently holding net bit index of A

y = (1 << i)

TC: $\log_2 A$

SC: O(1)

01001 = 10

10000 = 16

11001 = 26

→ Y smallest number $\boxed{> A}$, such that

$$Y \wedge A = Y + A$$

$$A = 1\,0\,1\,1\,0\,0 = 19$$

$$Y = 1\,0\,0\,0\,0\,0\,0 = 2^6 = 64$$

$$1\,1\,0\,1\,1\,0\,0 = 2^6+2^5+2^3+2^2 = 108$$

$$2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$$

$$64\ \ 32\ \ 16\ \ 8\ \ 4\ \ 2\ \ 1$$

40

104

108

Tower of Hanoi (classical example of Recursion)

Tower of Hanoi

In the classic problem of the Towers of Hanoi, you have 3 towers numbered from 1 to 3 (left to right) and A disks numbered from 1 to A (top to bottom) of different sizes which can slide onto any tower. The puzzle starts with disks sorted in ascending order of size from top to bottom (i.e., each disk sits on top of an even larger one).
You have the following constraints:

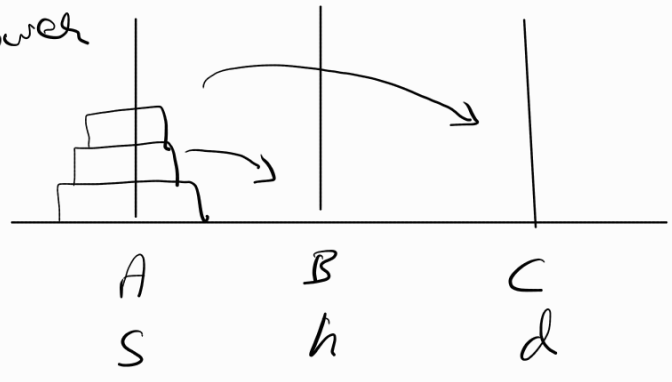
Only one disk can be moved at a time.
A disk is slid off the top of one tower onto another tower.
A disk cannot be placed on top of a smaller disk.

You have to find the solution to the Tower of Hanoi problem.
You have to return a 2D array of dimensions M x 3, where M is the minimum number of moves needed to solve the problem.
In each row, there should be 3 integers (disk, start, end), where:

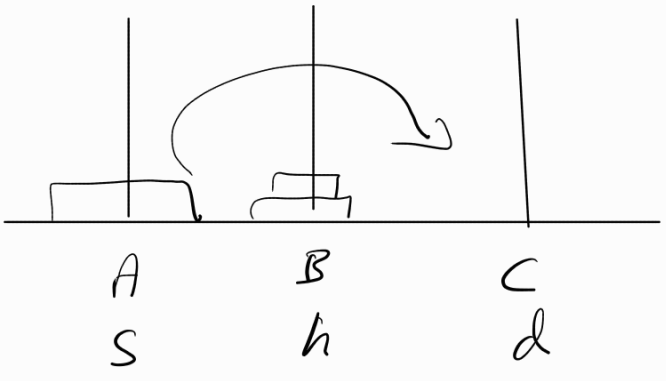
disk - number of the disk being moved
start - number of the tower from which the disk is being moved
end - number of the tower to which the disk is being moved

1. N num of disc placed on a tower

return [disk, from, to]



ex: N = 3



1. move $N-1$ disk to helper disk
2. move Nth disk to distination tower
3. move $N-2$ disk from helper disk to source tower
4. move $(N-1)$ th disk to distination.

:

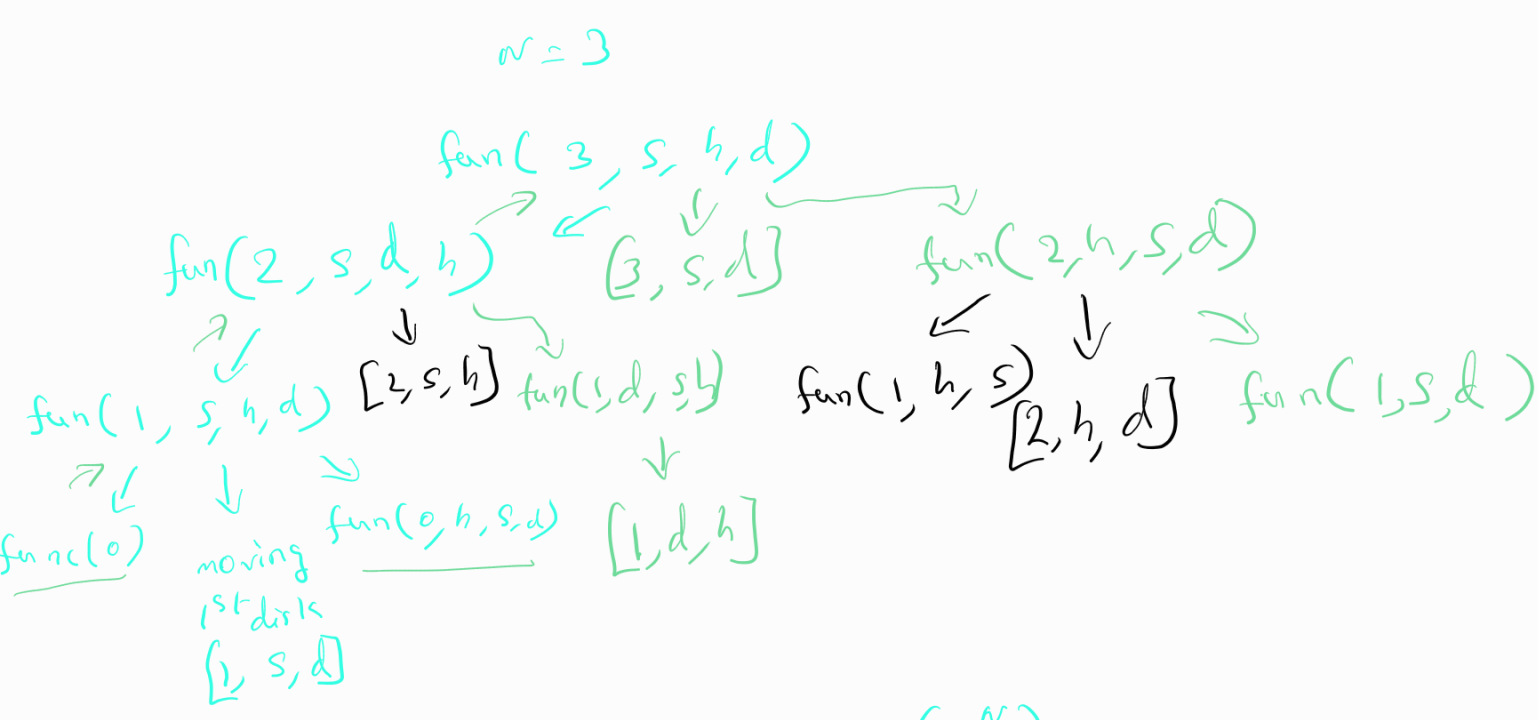
move (N) disks.

① move $(N-1)$ disks to helper
② move Nth disk to distination.
③ move $(N-1)$ disk to distination.

```
def moveDisks(N, S, h, d, ans)
    if N = 0: return ans  # base case

    moveDisks(N-1, S, d, h)
    temp = []
    temp.append(N)
    temp.append(S)
    temp.append(d)
    ans.append(temp)
    moveDisks(N-1, h, S, d)
    return ans
```

moving Nth disk.



[1, s, d
2, s, h
1, d, h
3, s, d
1, h, s
2, h, d
1, s, d]

TC: (2^N)

func calls $\Rightarrow 2^0, 2^1, 2^2, 2^3 \ldots 2^N$

$\Rightarrow 2^N$

Time per function call $\Rightarrow$ Constant

SC: $O(N)$

Dept of func calls = N

1, 2, 3, ... N

space per func call = 3