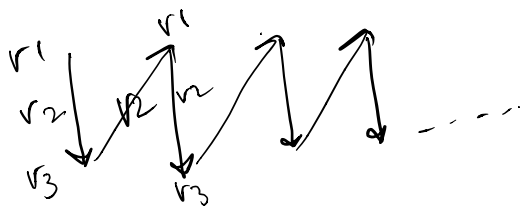
ZIGZAG CONVERSION

String "PAYPALISHIRING" is written in ZigZag pattern on a given number of rows like :

P A H N
A P L S I I G
Y I R

14/3 = 4

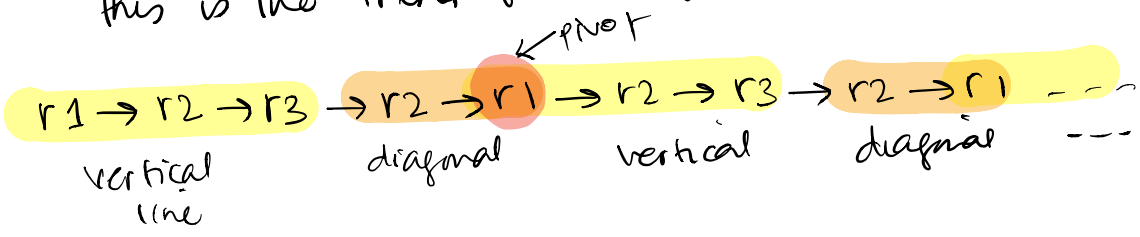
we can see here that the pattern looks something like this :



fill rows in the same fashion

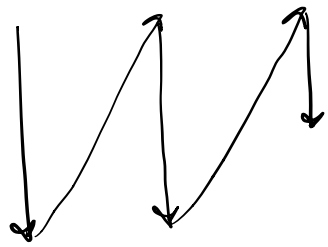
P A H N
A P L S I I G
Y I R

we can see from the above example that this is the trend followed for 3 rowed pattern.



Similarly for a 4 rowed pattern, given

r1 P I N
r2 A L S I G
r3 Y A H R
r4 P I



vertical diagonal pivot vertical

r1 → r2 → r3 → r4 → r3 → r2 → r1 → r2 → r3 → r4

INFERENCES

So from the above two patterns we can see that the Rig zag pattern always pivot around r1. (this is where a diagonal flows in and vertical flows out).

0 1 2 3 4 5 6 7 8 9 10 11 12 13
S = PAYPALISHIRING

total_rows = 4^{-1}, i = 0

So row_1 will always have elements from

S[0] , S , S[12]

S[i * (total_row + total_rows - 1)]

i = 0, S[0]
i = 1, S[1 x 6] = S[6]
i = 2, S[2 x 6] = S[12]
⋮

row_calc = i * (total_row + total_row - 1)

Similarly row_2 will always have elements from: i=1

s[1], s[5], s[7], s[11], s[13].

s[i * (total_row + total_row - 2)]

row_3 s[2], s[4], s[8], s[10].

row_4 s[3], s[9], s[15]

row_1; k=0

i=0, s[row_calc(0) - k], s[row_calc(0) + k]
= s[0], s[6]

i=1 s[row_calc(1) - k], s[row_calc(0) + k]
= s[6]

row_2; k=

s[row_calc(0) - k], s[row_calc(0) + k].
s[-1] , s[1]

s[row_calc(1) - k], s[row_calc(1) + k].
s[5] , s[7]

s[row_calc(2) - k], s[row_calc(2) + k]
s[11], s[13]

From the above inference we can see a pattern to calculate correct indexes for each rows. So we define a ROW_CALC(i) function that we can use to create a memoized list for base row indexes

```
ROW_CALC(i, n): // where n → total rows
    return i * (n + n-1)
```

```
// Create mem list for base row (i.e. row 1)
// And we can use this base index
// list to calculate indexes for other rows.

MEM_LIST = [ ]          N = total_no. of rows.

For i ← 0 to N:
    MEM_LIST.append (ROW_CALC(i), N-1)
```

This will give something like;

MEM_LIST = [0, 6, 12] // First row will never have elements ≥ N

```
// Now use this memList to fill
// indexes of elements for each row.
// And store them in an output variable

OUTPUT = [set() For i in 0 -> N]

// Output for first row
// will be same as memList.
// So fill first row with memList

for i in mem_list:
    OUTPUT[0].add(i)

// Now fill the rest.

for row in 1 -> N
    // Fill elements at row
    k = 1
    for mem in MEM_LIST:
        // Fill left
        if mem - k > 0:
            OUTPUT[row].add(mem-k)
        // Fill right
        if mem + k <= len(s):
            OUTPUT[row].add(mem+k)
```

[[], [], [], []]

// NOW OUTPUT has all the elements in order. So we can start printing from row 1 --- row Nth.

```
batch = [ ]
For out in OUTPUT :
    batch.append(" ".join(out))
return "".join(batch)
```

COMPLEXITY ANALYSIS

$O(r \times k)$

- r ↓ no of rows
- k ↓ size of mem list

But we know, the greater the no. of rows the smaller is the mem list and vice versa.

Hence at any point $O(r \times k) \approx O(n)$

APPROACH - 2

"PAYPALISHIRING"

		0 P	1 A	2 Y	3 P	4 A	5 L	6 I	7 S	8 H	9 I	10 R	11 I	12 N	13 G
0	R1	P						I						N	
1	R2		A				L		S				I		G
2	R3			Y		A				H		R			
3	R4				P						I				

i	j
0	0
1	1
2	2
3	3
2	4
1	5
0	6
1	7
2	8
⋮	⋮

R1 P I N

R2 A L S I G

R3 Y A H R

R4 P I

```
i = 0 → total rows
j = j+1 ≤ len(s)
result = [[] ... []] * total_rows
i ← -1
j ← -1
goDown ← True
while j ≤ len(s):
    IF i == total_rows -1 :
        goDown = False
    IF i == 0:
        goDown = True
    result[i] = s[j]
```

Complexity:

Time: O(n)

space: O(n)

```
IF goDown is True:
    i = i+1
ELSE:
    i = i-1

j = j+1
```

```
batch = []

O(n)  for r in result
          batch.append("".join(r))

O(n)  return "".join(batch)
```