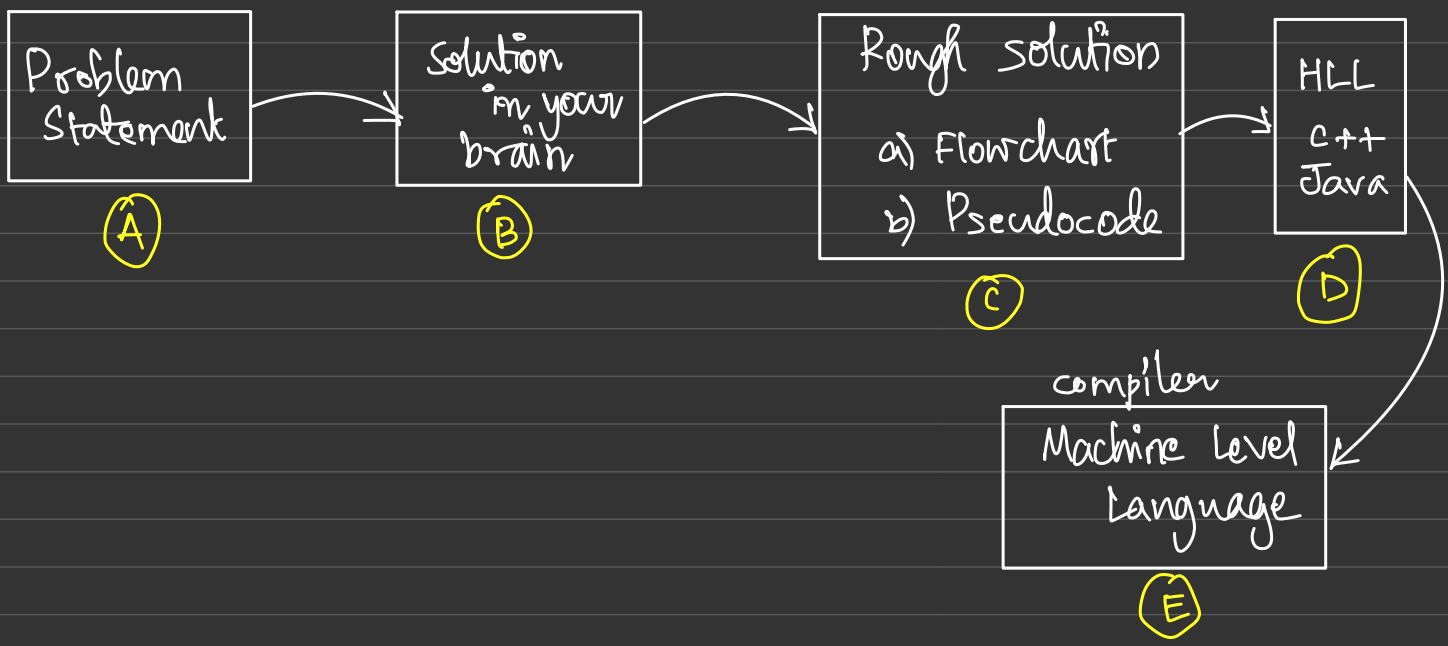
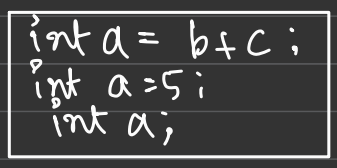


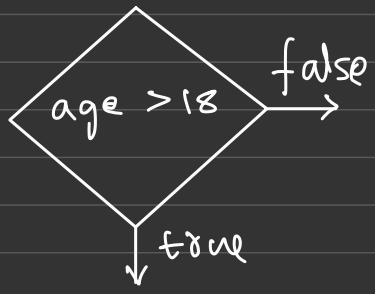
Flowchart Components

→ terminator block

→ input / output block
(read) (print)



→ process block
calculation / initialization / declaration

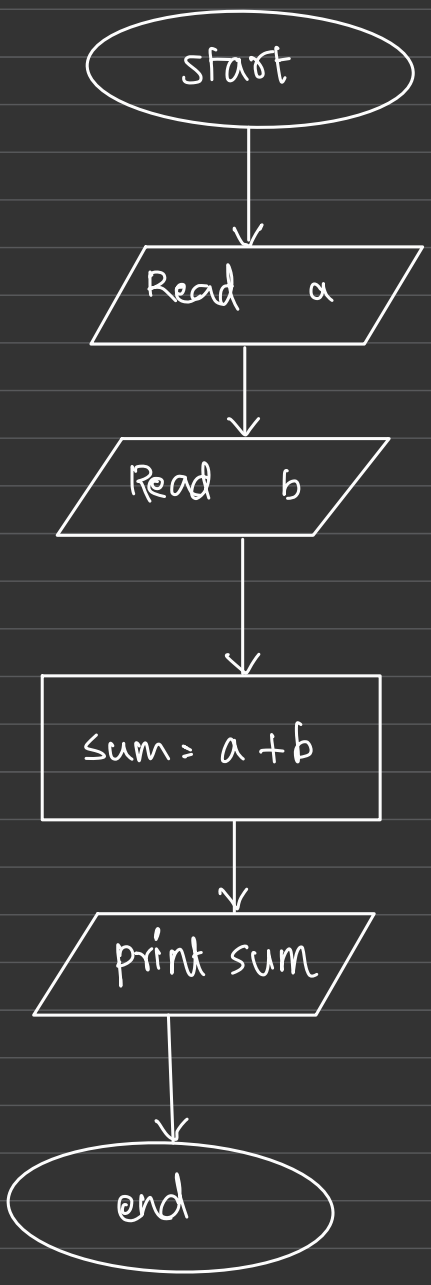


→ decision making block

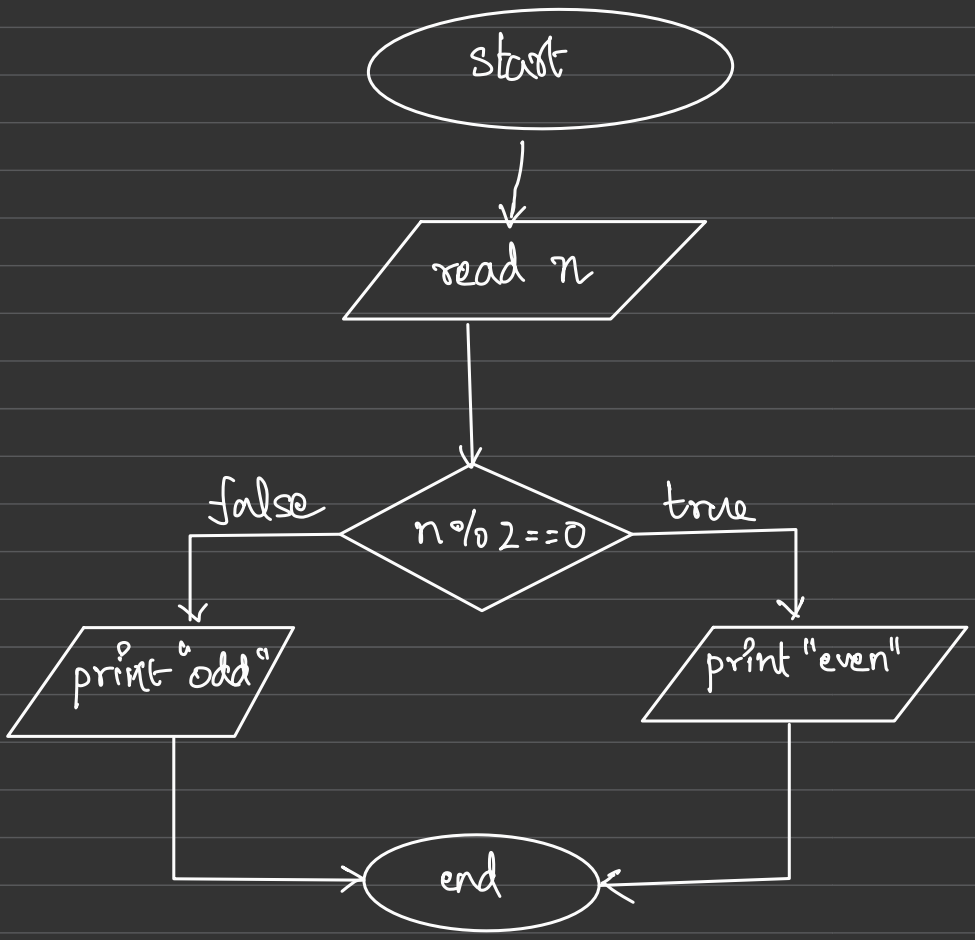
– Connectors

– flow of execution (arrows)

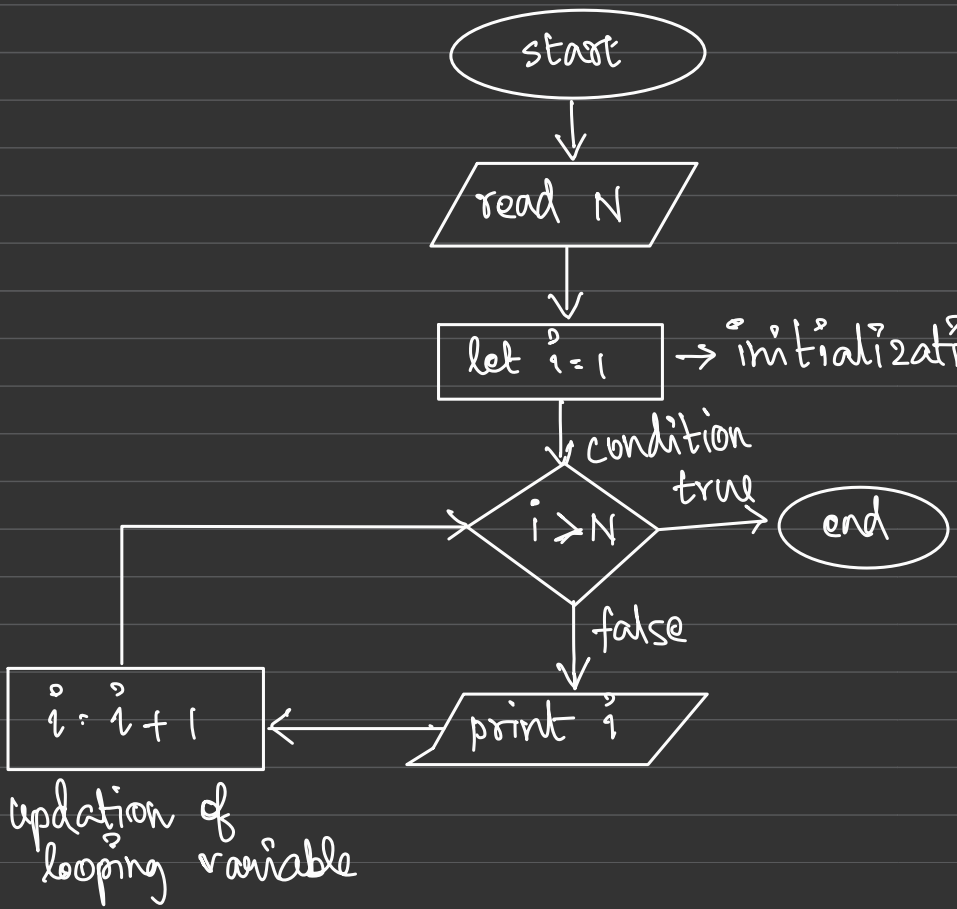
Print sum of a and b



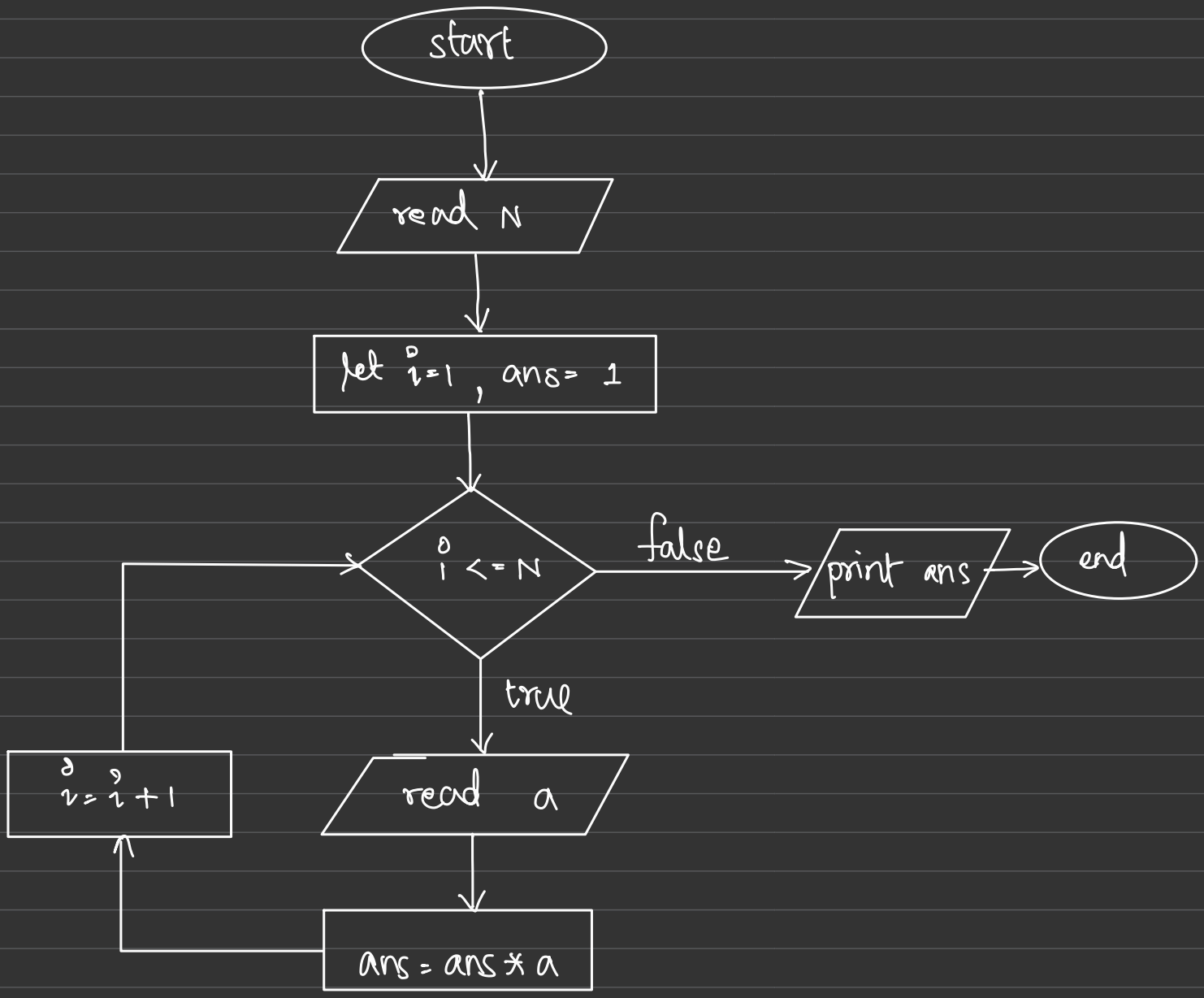
Check num is even or odd



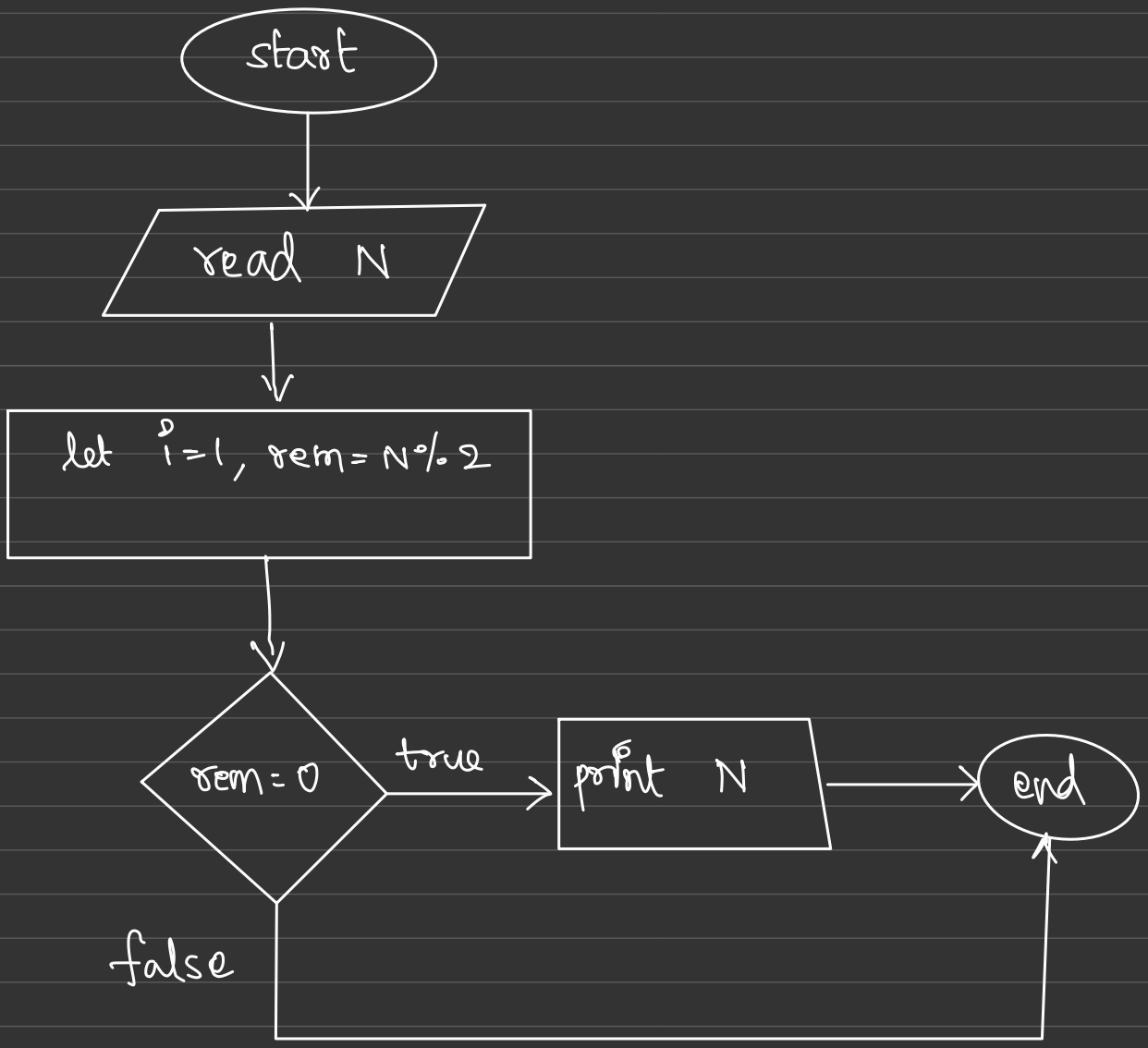
Print 1 to N



Multiply N numbers from user



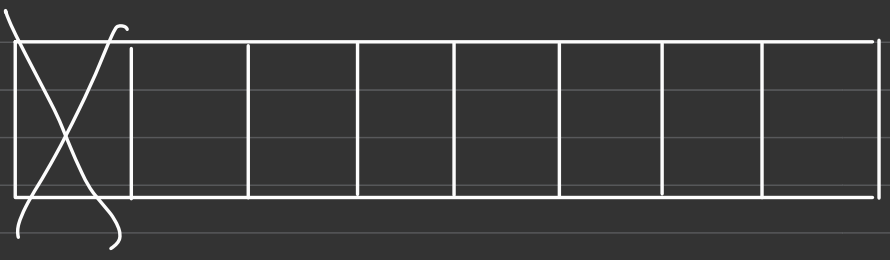
Print 1 to N, but only even numbers



Signed & unsigned data

For character data types there can be 2^8 total combinations. In case of unsigned data, the most significant bit will be used to indicate sign either positive (0) or negative (1).

char → 1 byte → 8 bits



$$\text{total combinations} = \frac{2^8}{2} = 2^7$$

For negative combinations, you can reach upto -2^7
For positive combinations, you can reach upto $2^7 - 1$

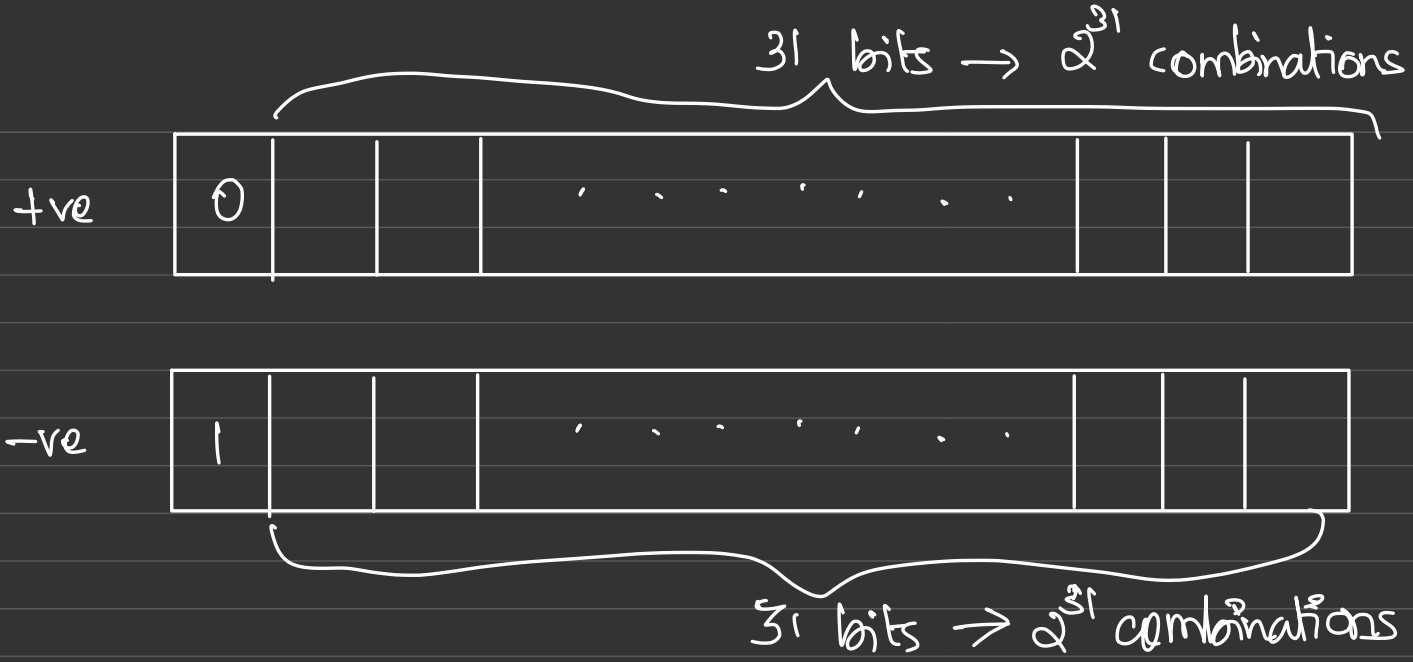
For n bits in a signed data the range would be

$$-2^{n-1} \rightarrow 2^{n-1} - 1$$

+ve → 0 → 128 − 1 = 0 → 127

-ve → -1 → -128

So, the range would be $-128 \rightarrow 127$



In integers, the range for a positive number is

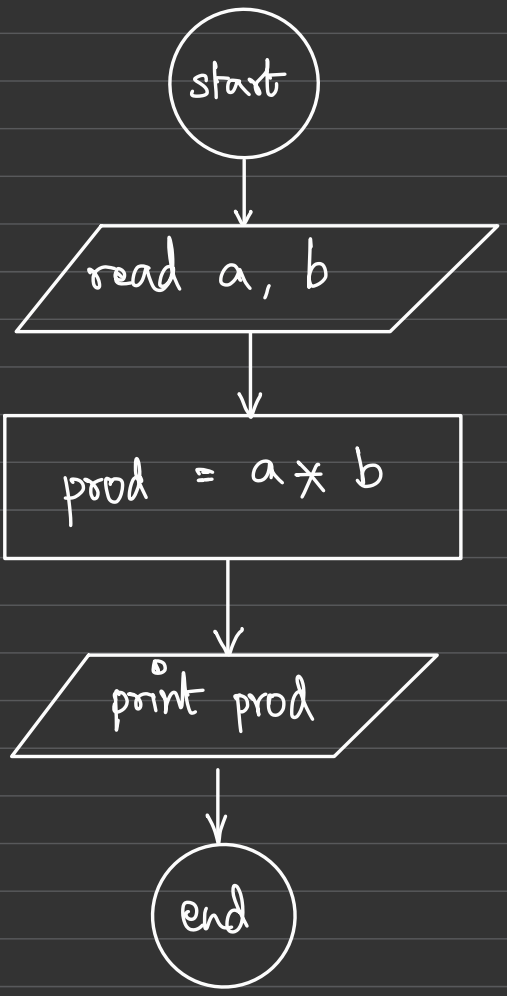
$0 \rightarrow 2^{31}-1$

and negative numbers would range from $-1 \rightarrow -2^{31}$

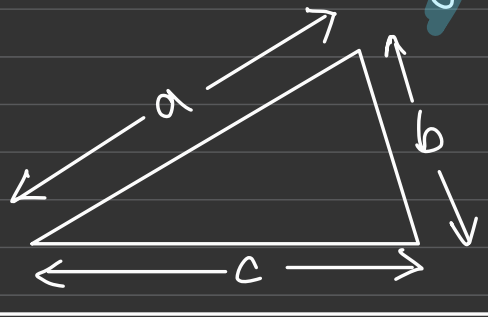
So, the total range would be $-2^{31} \rightarrow 2^{31}-1$

Assignments - Flowcharts

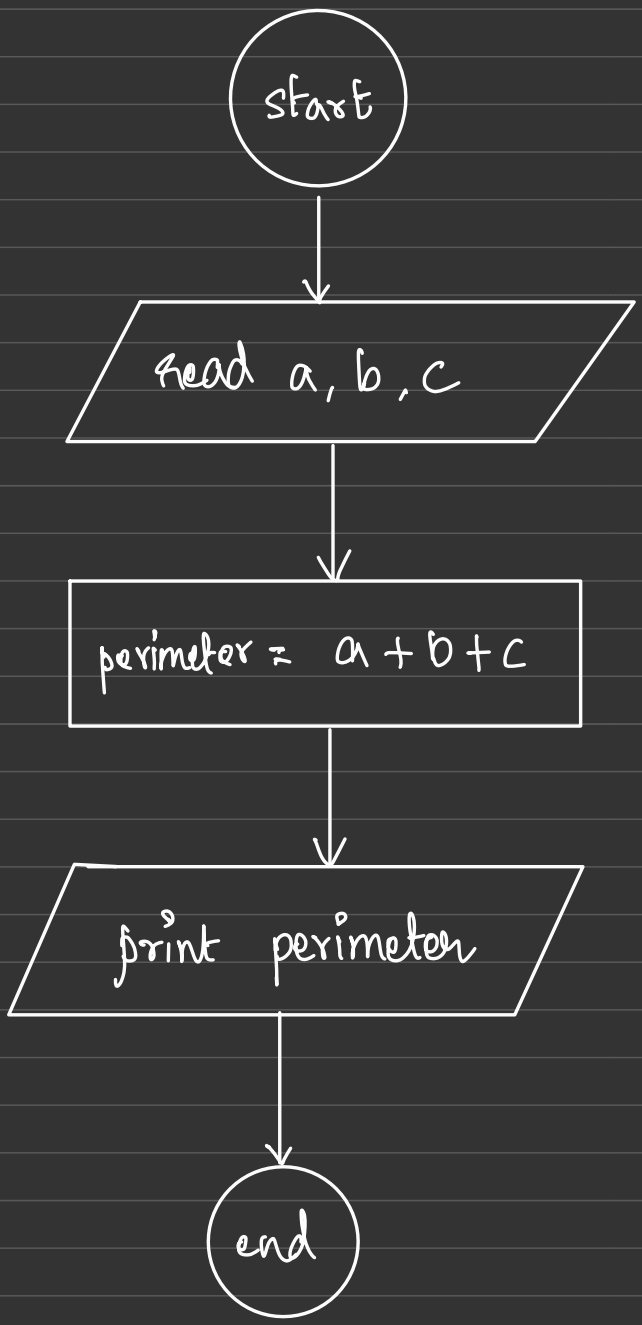
1. Multiply 2 numbers



2. Find perimeter of a triangle

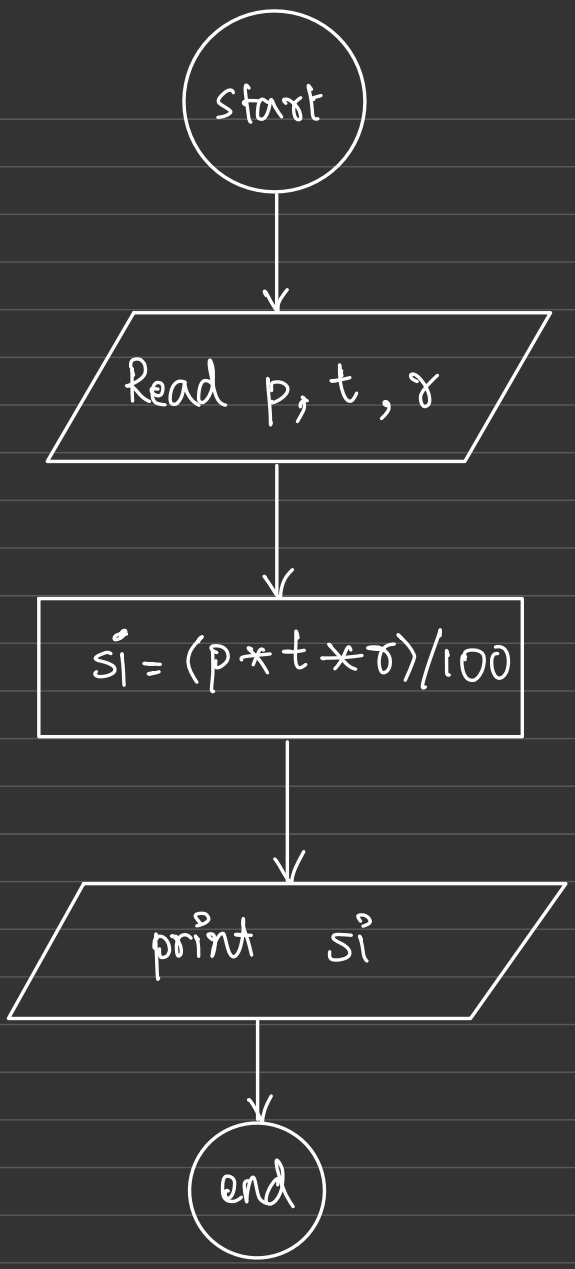


Δ^{les} perimeter = Sum of all sides of a triangle



3. Find simple interest

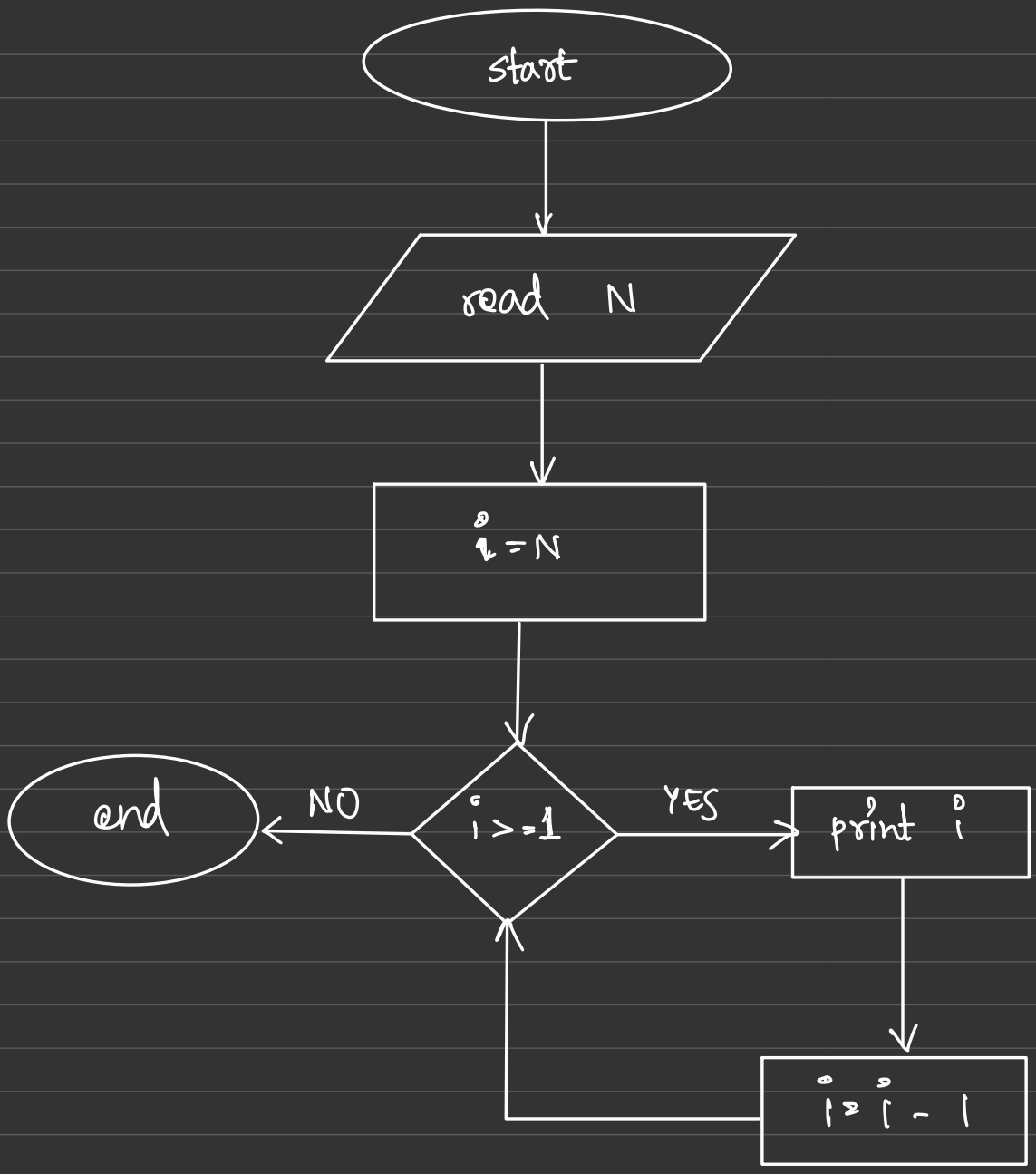
$$S.I = \frac{\text{Principle Amount} * \text{Rate of interest} * \text{Time}}{100}$$



4. Print counting from N to 1

Ex:-

```
N = 5
for(int i= N; i>0; i--){
    System.out.println(i);
}
```



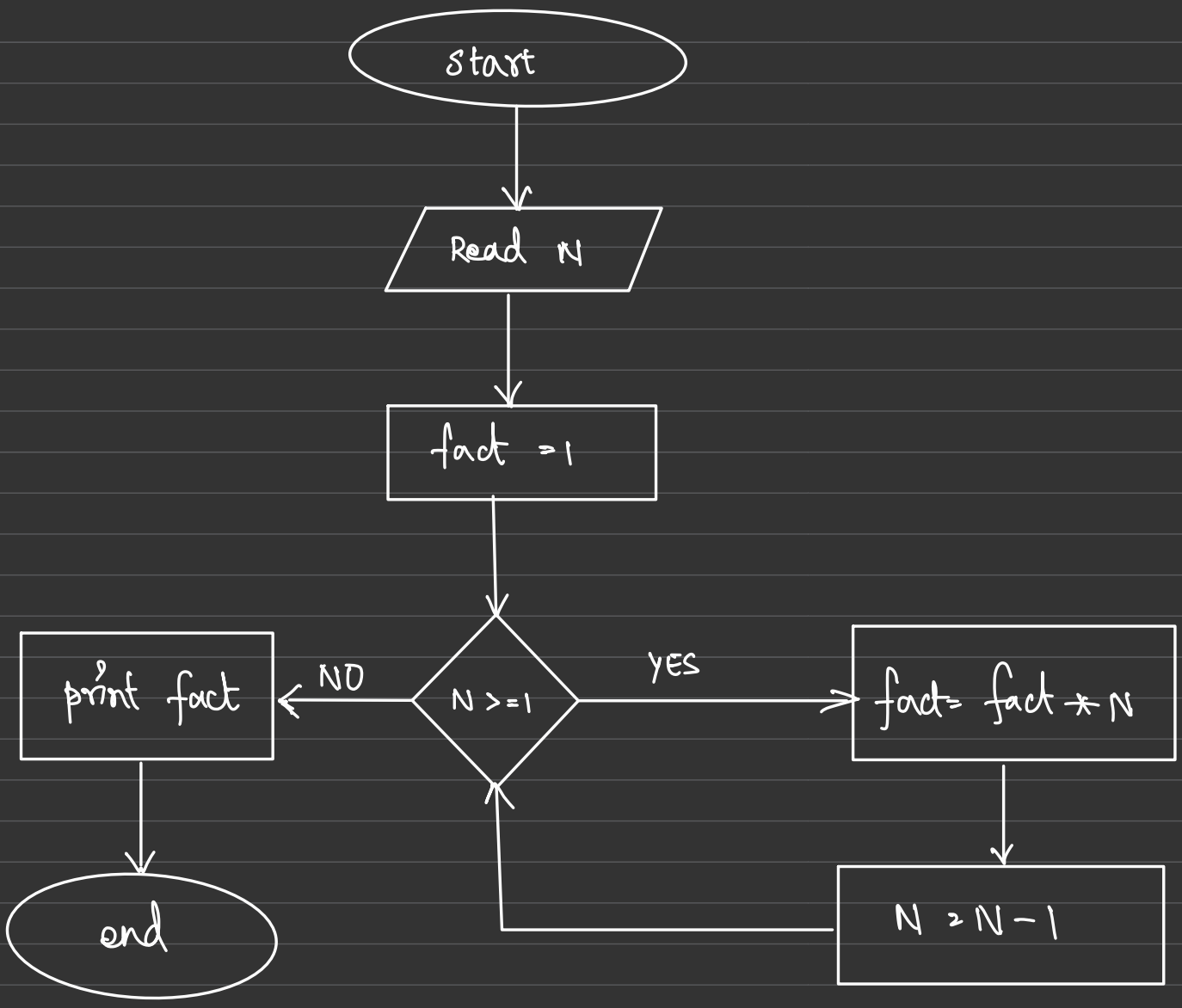
5. Find factorial of a number:

$$N! = N * (N-1) * (N-2) * (N-3) * \ldots . * 1$$

Eg: N = 5

$5! = 5 * 4 * 3 * 2 * 1 = 120$

$3! = 3 * 2 * 1 = 6$



6. Check if a number is prime or not

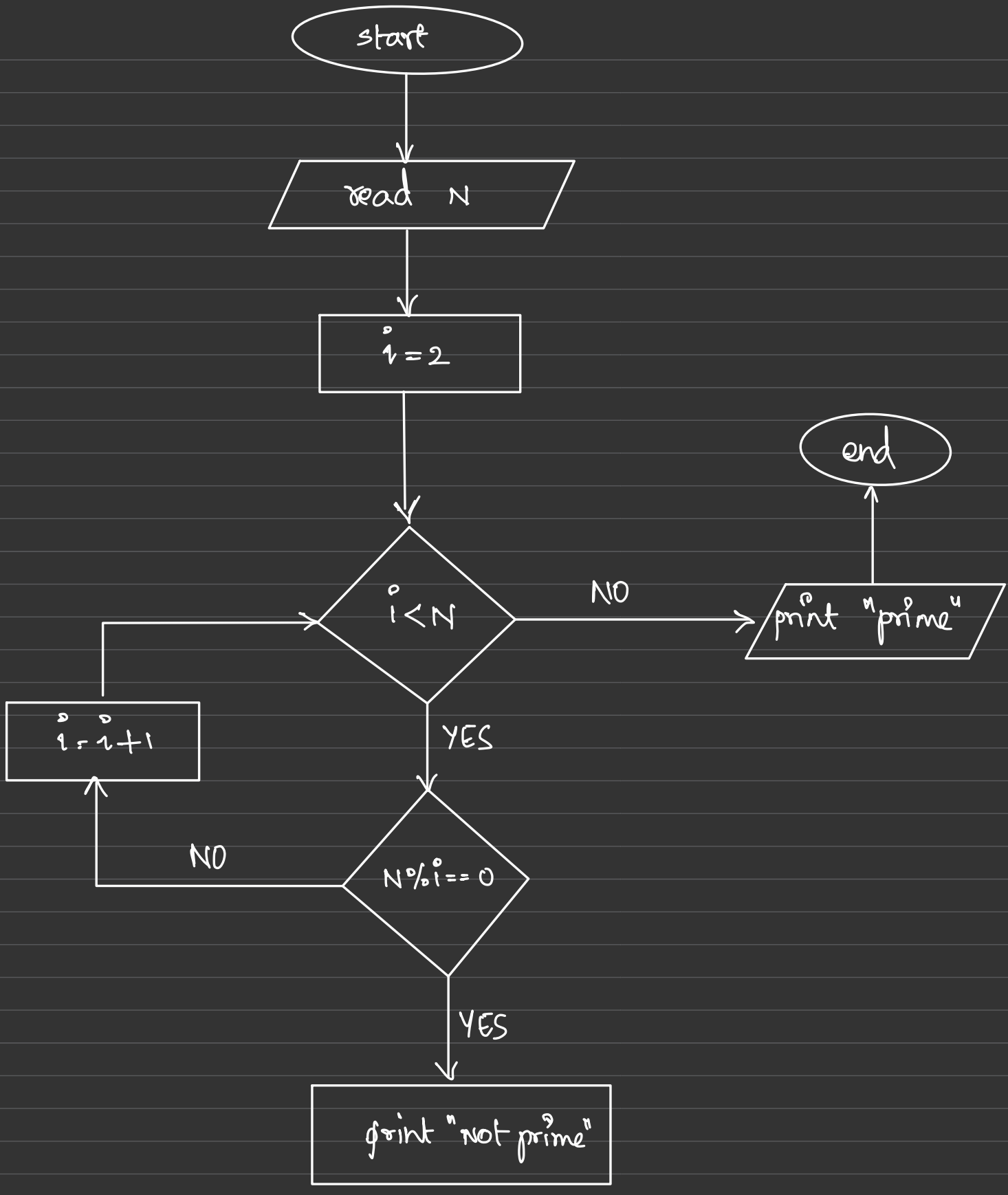
if N is a prime number, we must have only 2 factors - a) 1 b) N itself

Eg: 5 → 1 and 5 → 5/1, 5/2, 5/3, 5/4, 5/5

7 → 1 and 7 → 7/1, 7/2, 7/3, 7/4, 7/5, 7/6, 7/7

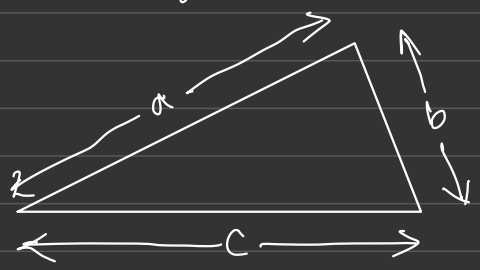
11 → 1 and 11 → 11/1, 11/2, 11/3, 11/4, 11/5 ... 11/11

5, 7 and 11 are prime numbers.


start
read N
i = 2
i<N
NO
print "prime"
end
i = i+1
YES
N%i == 0
NO
YES
print "not prime"

7. Check if a given triangle is valid or not

To define a triangle we need to know its sides



Mathematically, for a triangle to be valid we need to satisfy a condition

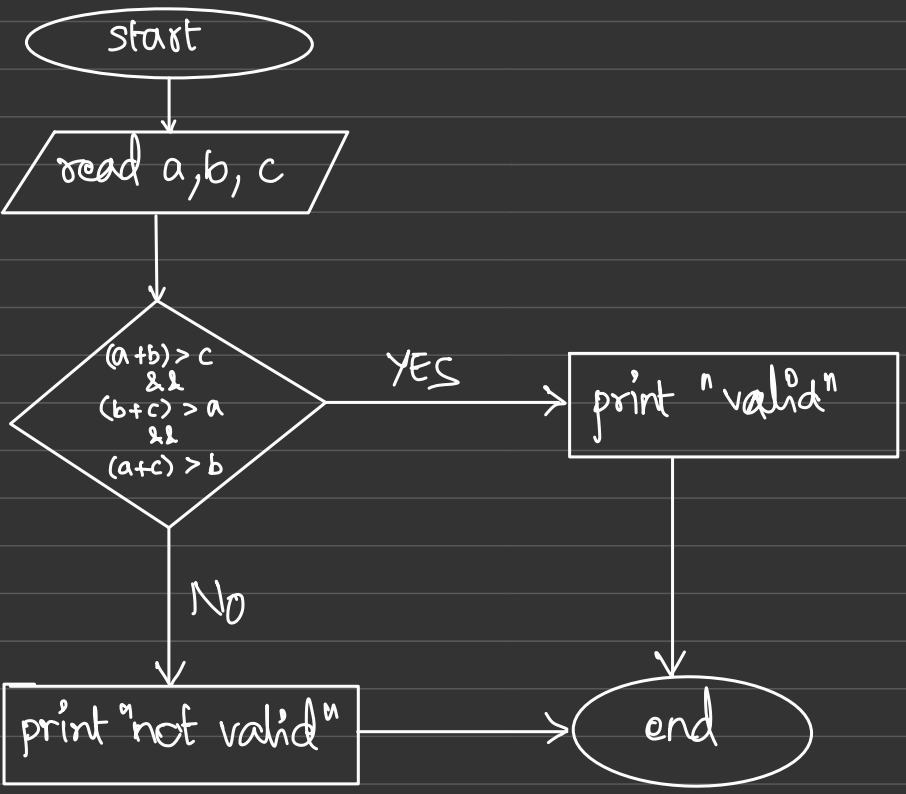
Σ two sides > other side

i.e.,

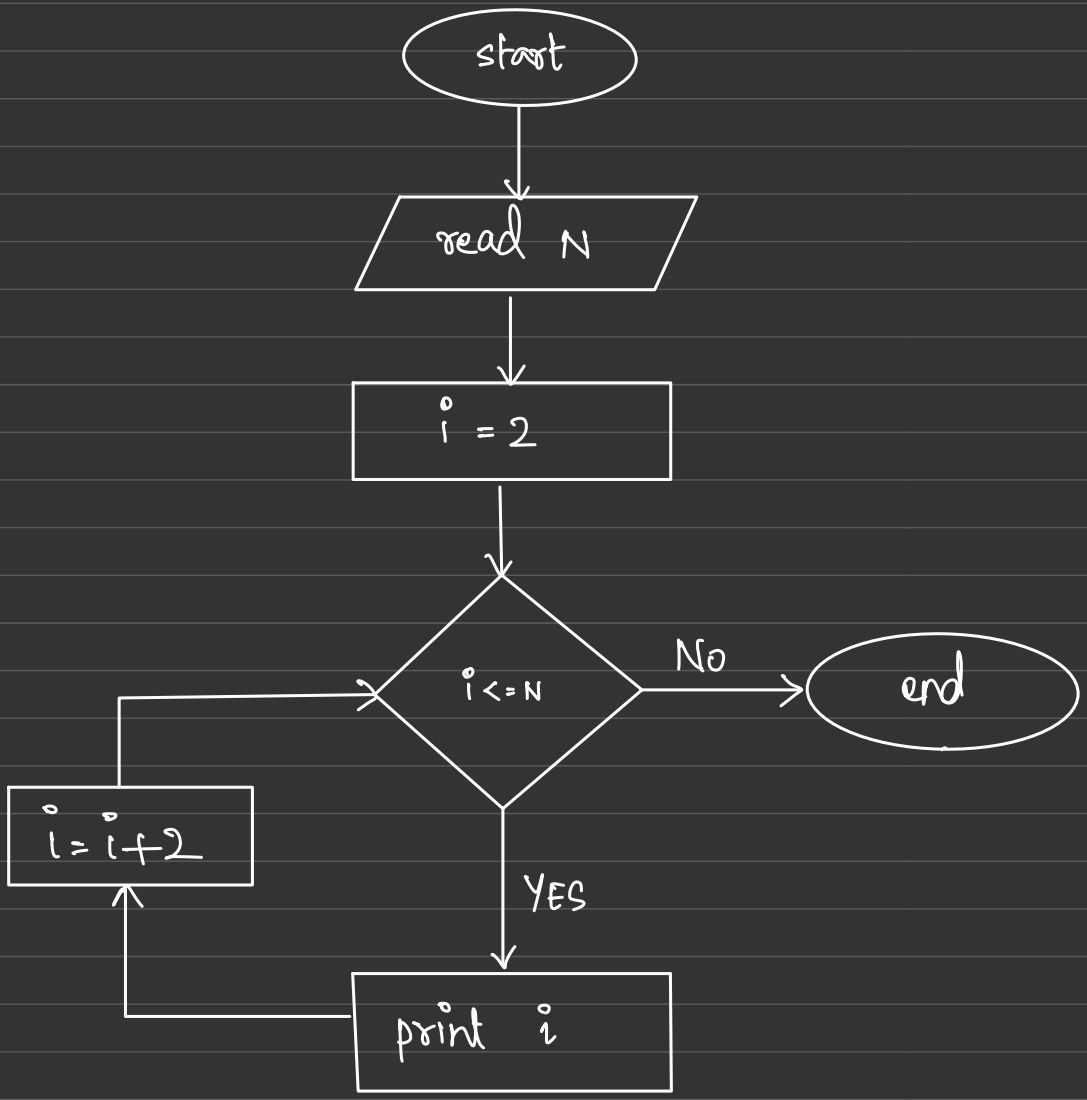
$(a+b) > c$

$(b+c) > a$

$(a+c) > b$



8. Print only even numbers from 1 to N

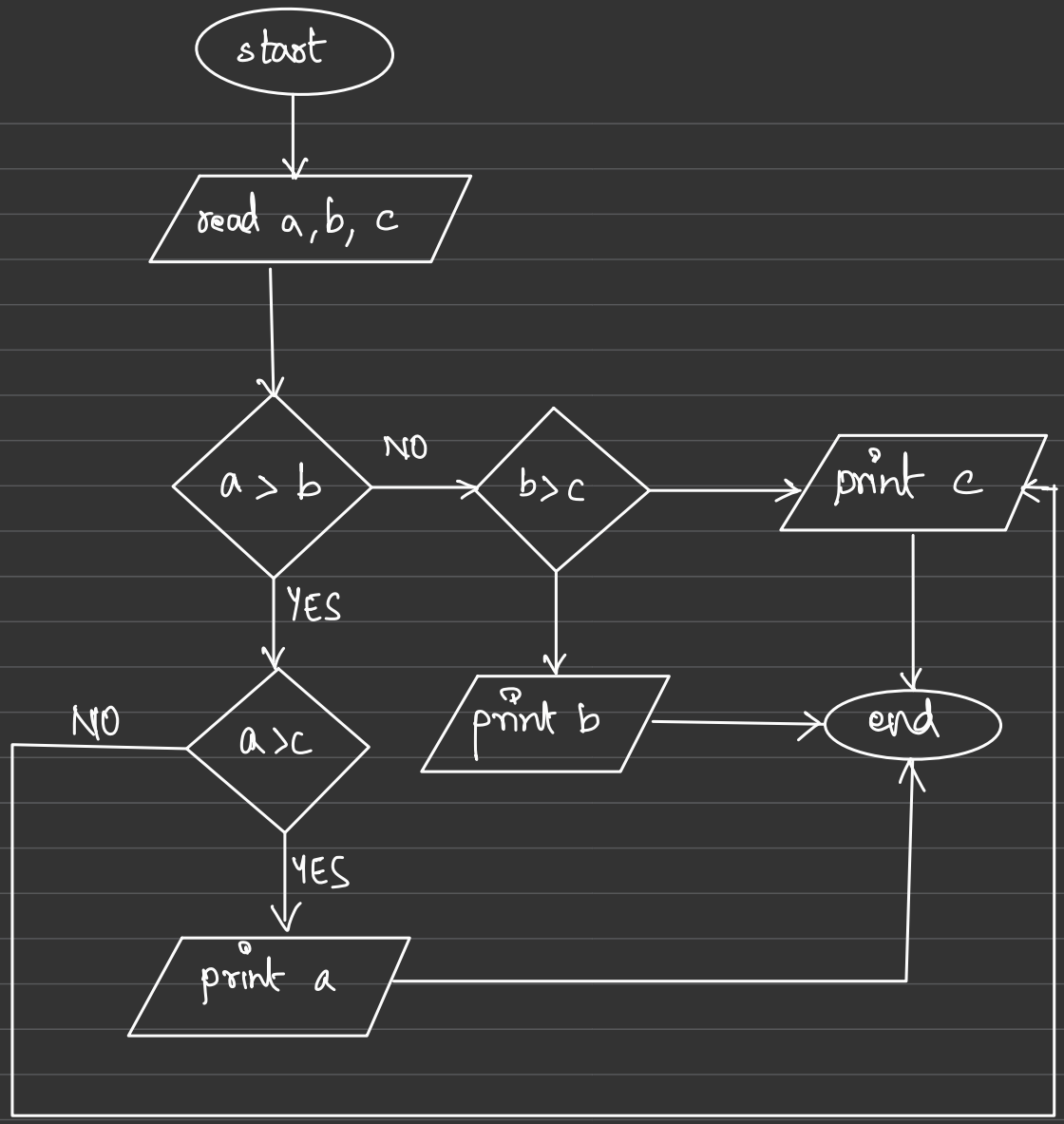


9. Print max of 2 numbers

Eg: a = 2 a > b? → a > c? → max is a

b = 3

c = 1


start
read a, b, c
a > b
NO
b > c
print c
YES
print b
end
NO
a > c
YES
print a

Patterns : 1. Hollow pyramid pattern

N = 5 = ROWS

= COLS = N * 2 - 1 = 5 * 2 - 1 = 9

= so, 5 * 9 matrix

= N * (2N - 1) matrix → no. of columns

for 5 rows, we have 9 columns.

if N = rows = 5, then cols would be (2N - 1) = 9

when i = 0 → 4 spaces - 1 * - 4 spaces

i = 1 → 3 spaces - 3 * - 3 spaces

i = 2 → 2 spaces - 5 * - 2 spaces

i = 3 → 1 space - 7 * - 1 space

* To print spaces, we arrive at a condition - (N - i - 1)
* To print stars, we arrive at a condition - (2i + 1)
* print a space

Dry runs

for $j < (N-i-1)$ and $N = 5$

1) if $i = 0$

$j < (4-0) \Rightarrow j < 4$

2) if $i = 1$

$j < (4-1) \Rightarrow j < 3$

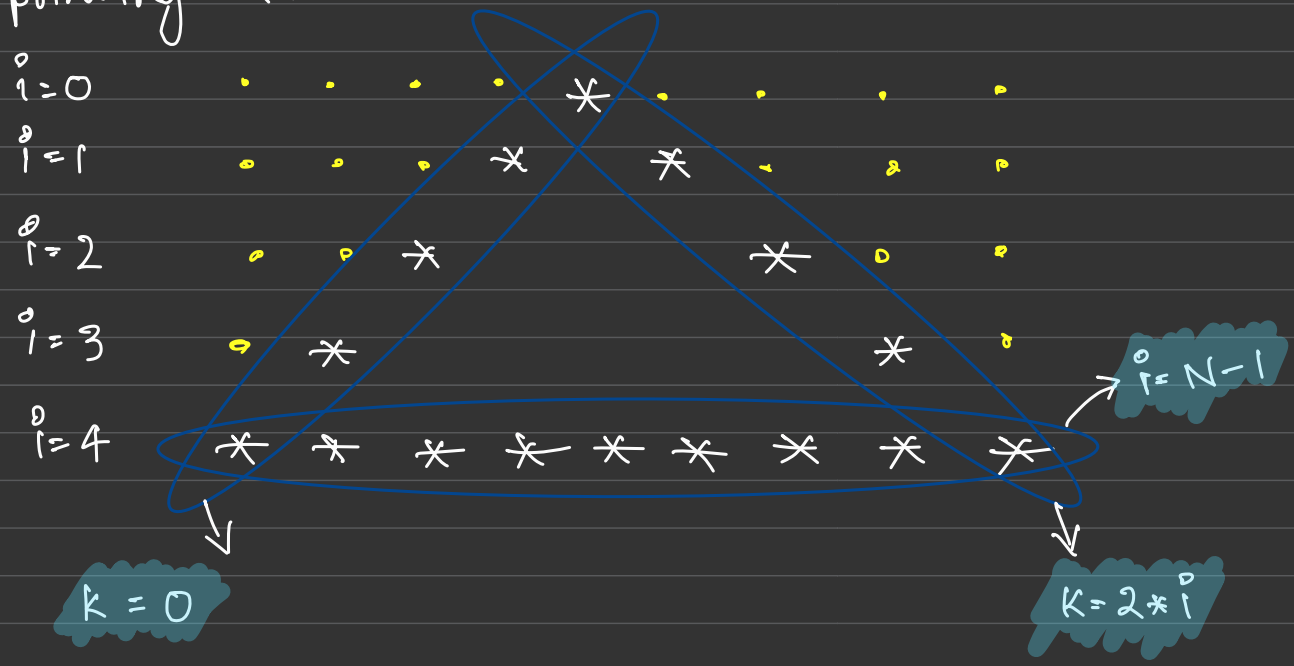
3) if $i = 2$

$j < (4-2) \Rightarrow j < 2$

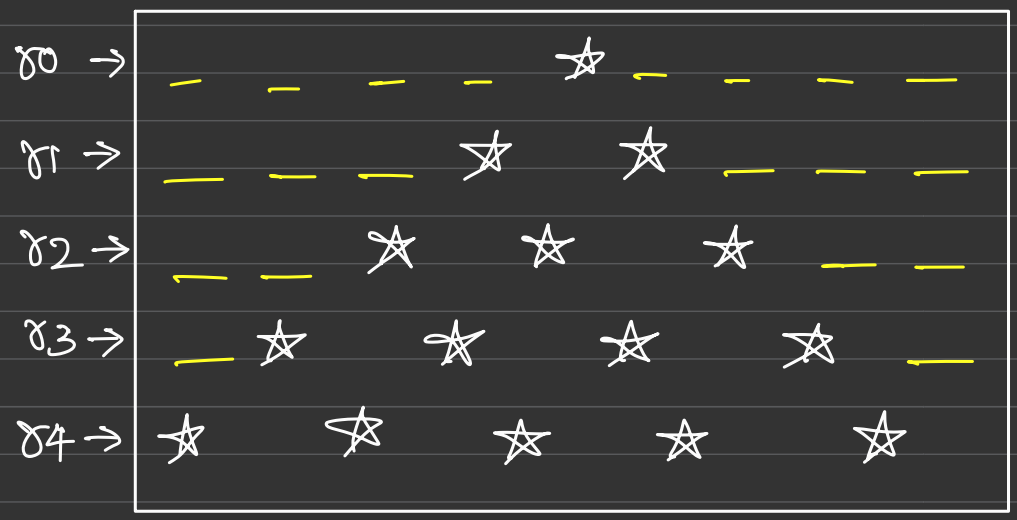
(4) if $i = 3$

$j < (4-3) \Rightarrow j < 1$

To make the triangle hollow, we have to make some changes to $(2i+1)$ since only this condition is responsible for printing *.



2. Full pyramid pattern



rows = N = 5

r0 → 4 sp, 1 ★

r1 → 3 sp, 2 ★

r2 → 2 sp, 3 ★

r3 → 1 sp, 4 ★

r4 → 0 sp, 5 ★

n=5, r0 → 4 sp, 1 ★

row+1

N - row - 1 = 5 - 0 - 1 = 4

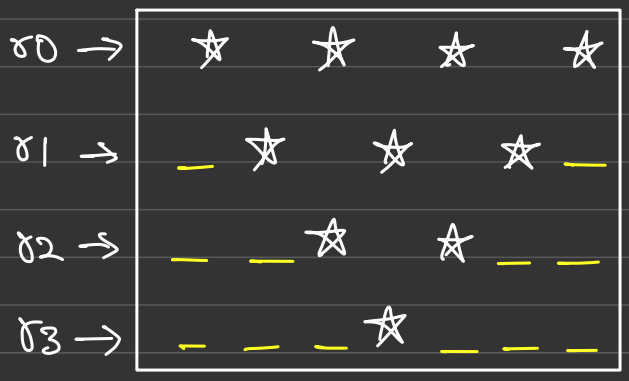
n=5, r1 → 3 sp, 2 ★

N - row - 1 = 5 - 1 - 1 = 3

n=5, r2 → 2 sp, 3 ★

N - row - 1 = 5 - 2 - 1 = 2

3. Inverted full pyramid pattern



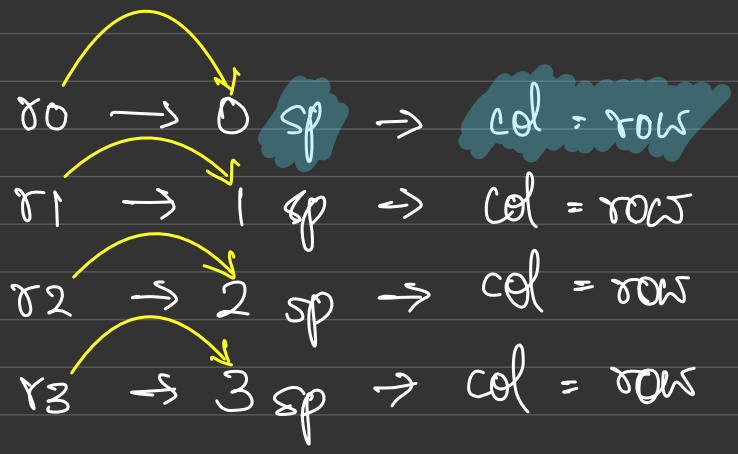
rows = N = 4

r0 → 4 ★, 0 sp

r1 → 3 ★, 2 sp

r2 → 2 ★, 4 sp

r3 → 1 ★, 6 sp



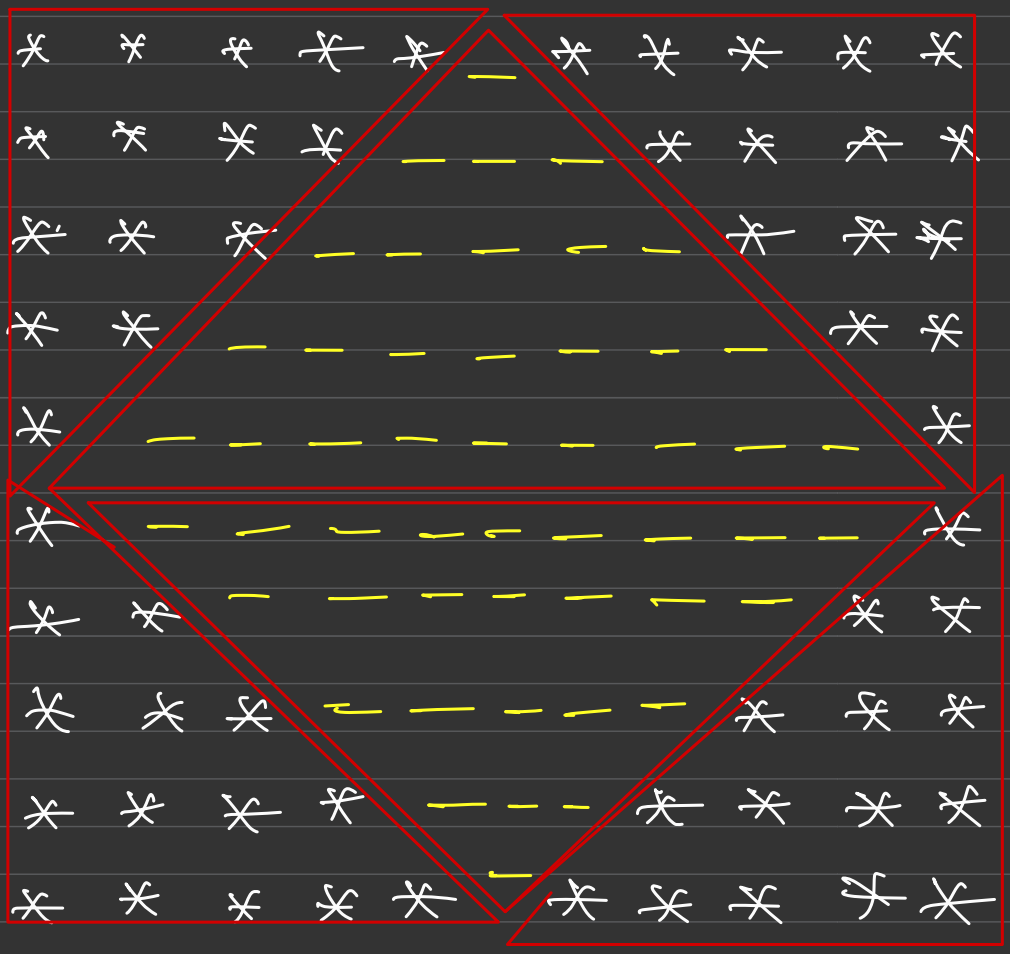
r0 → 4 ★ → N - row = 4 - 0 = 4

r1 → 3 ★ → N - row = 4 - 1 = 3

r2 → 2 ★ → N - row = 4 - 2 = 2

r3 → 1 ★ → N - row = 4 - 3 = 1

Flipped Solid Diamond pattern



N = 5

r0 * * * * *
r1 * * * *
r2 * * *
r3 * *
r4 *

0th row - 5 *
1st row - 4 *
2nd row - 3 *
3rd row - 2 *

N-row = 5-0 = 5
N-row = 5-1 = 4
N-row = 5-2 = 3
N-row = 5-3 = 2
⋮

for spaces -

	r - *
r0 - 1 sp	0 - 1
r1 - 3 sp	1 - 3
r2 - 5 sp	2 - 5
r3 - 7 sp	3 - 7
r4 - 9 sp	4 - 9

2 * row + 1

2 * row + 1	2 * 1 + 1 = 3
2 * row + 1	2 * 2 + 1 = 5
2 * row + 1	2 * 3 + 1 = 7
2 * row + 1	2 * 4 + 1 = 9

N = 5

r0 * — 0 - 1 * — row+1 — 0+1 = 1

r1 * * — 1 - 2 * — row+1 — 1+1 = 2

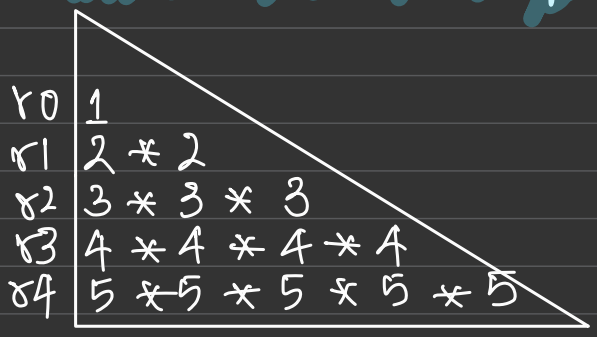
r2 * * * — 2 - 3 * — row+1 — 2+1 = 3

r3 * * * * — 3 - 4 * — row+1 — 3+1 = 4

For spaces,

2 * N - 2 * row - 1

Numbers and stars pattern - 1



row = 4 = N

r0 → 1

r1 → 2

r2 → 3

r3 → 4

row+1

r0 → 1 char

r1 → 3 char

r2 → 5 char

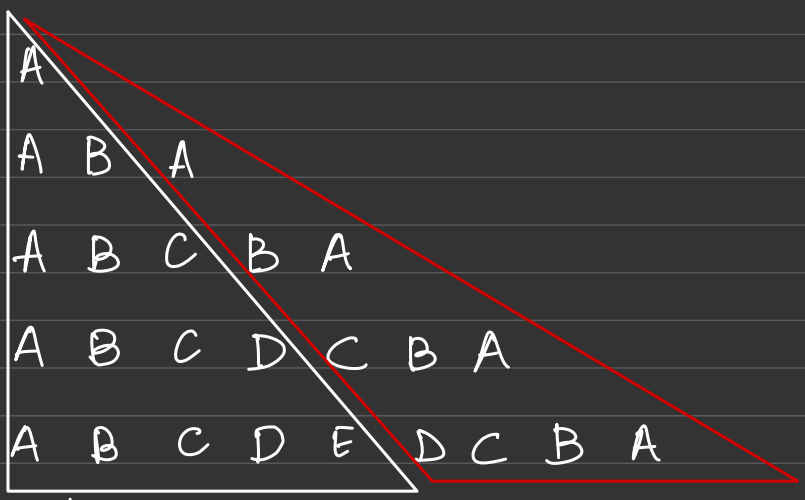
r3 → 7 char

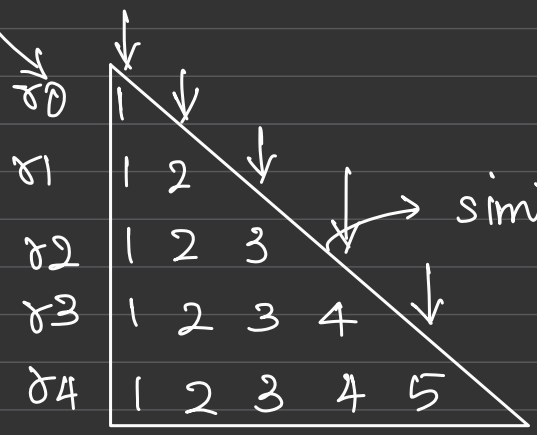
2 * row + 1

even columns have numbers

odd columns have stars

Alphabetical triangle pattern





similar to this pattern

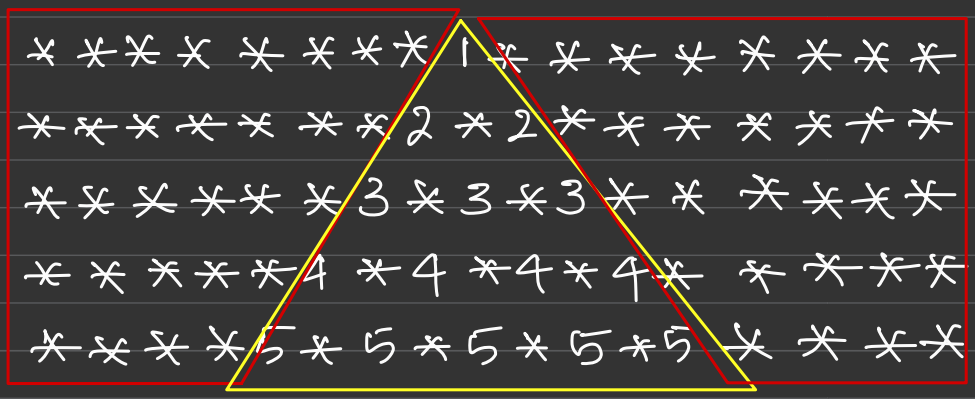
formula → row

we need to map the numbers to alphabets.

One simple way would be add a character to the number and decrement by 1

`col + 1 + 'A' - 1` this will print the character

Numbers and stars pattern - 2



Problem can be broken into 3 parts as shown above

N = 5

r0 * * * * * * * * *
r1 * * * * * * *
r2 * * * * *
r3 * * * *
r4 * * *

2N - row - 2 - formula

1
2 * 2
3 * 3 * 3
4 * 4 * 4 * 4
5 * 5 * 5 * 5 * 5

2 * row + 1 - formula

print * on even columns

print numbers on odd columns

Numeric Hollow Inverted Half Pyramid Pattern

col 1 ↓

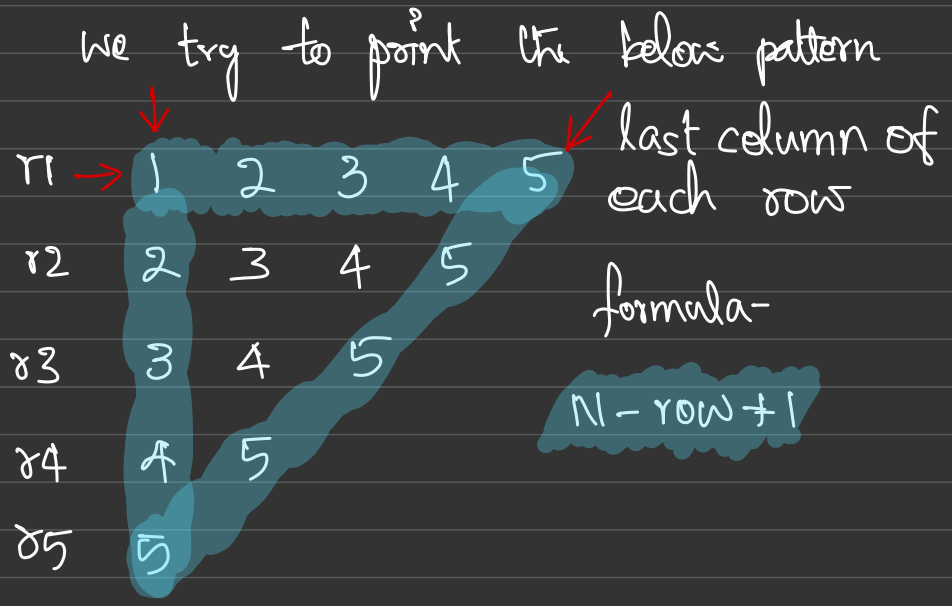
r1 - 1 2 3 4 5 ← row 1

r2 - 2 5

r3 - 3 5

r4 - 4 5

r5 - 5

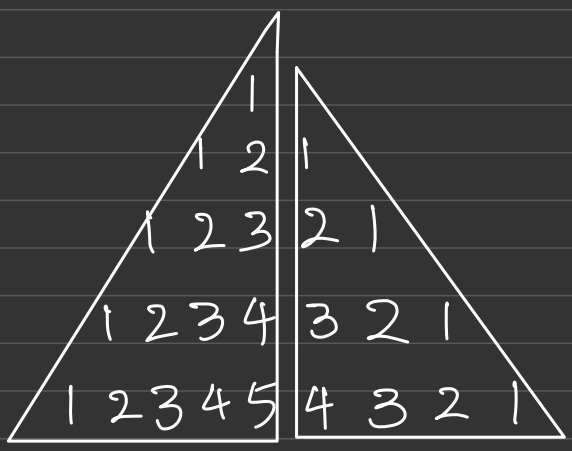


Now, we put conditions to remove the digits which are not needed.

print digits on col 1 and col N-row+1 and print spaces for others.

However, we also need to print numbers for row 1

Numeric Palindromic Equilateral Pyramid Pattern



we can break the problem into 2 parts

N = 5

```
    1 <- r1
   1 2 <- r2
  1 2 3 <- r3
 1 2 3 4 <- r4
1 2 3 4 5 <- r5
```

r1 - 4 spaces - 1 number

r2 - 3 spaces - 2 numbers

r3 - 2 spaces - 3 numbers

for spaces formula is N - row

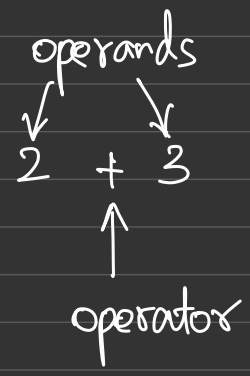
for numbers formula is simply row

```
1          <- r1
2 1        <- r2
3 2 1      <- r3
4 3 2 1    <- r4
```

formula is row

Bit wise operators :- works on bit level

- AND → &
- OR → |
- NOT → ~
- XOR → ^



Truth table

& (AND)

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

(OR)	a	b	a | b
	0	0	0
	0	1	1
	1	0	1
	1	1	1

^ (XOR)

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

same bits - 0
diff bits - 1

~ (NOT)

a	~a
0	1
1	0

XOR - If you send same bits, it will give you 0.

Ex:- ① 5 ^ 5 - 0 → (XOR of same numbers is always a 0)

So if you want to cancel out any two numbers (duplicates) you can perform a XOR operation.

② If input has some duplicates, to find the duplicates you XOR the input.

input - 2 ^ 3 ^ 4 ^ 5 ^ 2 ^ 4 ^ 3 → output - 5

2 & 3

2 - 0 1 0

3 - 0 1 1

(&) 0 1 0 - 2

5 & 10

5 - 0 0 0 0 0 1 0 1

10 - 0 0 0 0 1 0 1 0

(&) 0 0 0 0 0 0 0 0 - 0

5 | 10

0 0 0 0 0 1 0 1

0 0 0 0 1 0 1 0

(|) 0 0 0 0 1 1 1 1 - 15

~1

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

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

↳ represents a negative sign (left most bit)

To access the negative number, we do a 2's complement

2's complement = 1's complement + 1

00000000 00000000 00000000 00000001

+1

00000000 00000000 00000000 00000010 = 2

So, the output of v1 is -2

5 ^ 10

00000000 00000000 00000000 00000101

(^) 00000000 00000000 00000000 00001010

00000000 00000000 00000000 00001111

15

5 ^ -5

5: 00000000 00000000 00000000 00000101

-5 = 11111111 11111111 11111111 11111011

-5 is stored as 2's complement of 5

(^) 11111111 11111111 11111111 11111110

-ve number so we do a 2's complement

00000000 00000000 00000000 00000001

1

00000000 00000000 00000000 00000010 → 2

---

5 ^ -5 = -2

Left and Right Shift Operators

Left shift operator - <<

Right Shift Operator - >>

"<<" - when we left shift any number by 1, then we are multiplying the number by 2^1.
If we shift by 2, then we are multiplying by 2^2 and so on.

```
int a = 2;
```

a<<1 → output : 4

```
int a = 5;
```

a<<1 → output : 10

```
int a = 7;
```

a<<1 → output : 14

If we left shift "a" by "n" times that is exactly equal to $a * 2^n$.

">>" - when we right shift any number by 1, we are technically dividing the number by 2^1.
right shift by 2 means dividing the number by 2^2
This is not true for all the cases. as it is compiler dependent.

int a = 4

a >> 1 o/p: 2

int a = 100

a >> 1 o/p: 50

a >> 2 o/p: 25

a >> 3 o/p: 12

int a = -5 = (stored as 2's complement)

→ MSB is -ve

(1) 1111 1111 1111 1111 1111 1111 1111 1011

if we do, a >> 1

(0) 1111 1111 1111 1111 1111 1111 1111 1101

→ MSB becomes a large +ve number. However, compiler will handle a signed negative integer. So -100 >> 1 would give us -50

but in case of an unsigned negative integer we would get a large positive number as output.

Shifting a number by a negative number would give us a garbage value.

```
int a = 10;
a<<-1    // prints a garbage value
```

→ never throws an error but this is wrong.

If we right shift a by n times, that means the number is getting divided by 2^n i.e., $a/2^n$

The above statement is compiler dependent and it is not always true.

Pre / post increment / decrement operator

pre-increment → ++a

post-increment → a++

pre-decrement → --a

post-decrement → a--

++a - first increment and then use the value.

a++ - first use the value and then increment.

--a - first decrement and then use the value.

a-- - first use the value and then decrement.

Examples

① public static void main() {
    int a=5;
    (++a);
    s.o.pln(a);
}

a → ~~5~~ 6

output: 6

② public static void main(){
    int a=5;
    s.o.pln(++a);
}

Here, we are
(a) incrementing the value
(b) also accessing the variable from print statement.

So, here again we increment and then print 6

③ public static void main( ) {

int a=5; a → ~~5~~ 6

print (a++); output : 5

}

→ first a will be used by print function, then the value will be incremented in the memory.

④ public static void main() {

int a = 21; a → ~~21~~ ~~22~~ 23

★ (A) → incr
22 (B) → use print (++a); output : 22

22 (A) → use
★ (B) → incr print (a++); output : 22

23 (A) → print print (a); output : 23

}

⑤ public static void main( ) {

int a = 10; a → ~~10~~ ~~11~~ 12

print (++a * a++) ;

11 * 11 = 121

int a=10;

print (a++ * ++a); a → ~~10~~ ~~11~~ 12

10 * 12 = 120

Break & Continue keywords

Break

```
for (int i=0; i<5; i++) {
    if (i==2)
        break;
}
```

i=0
0<5 → T
i=1
1<5 → T
i=2
2<5 → T
break

control goes out of for loop as break keyword will break the continuous flow of for-loop.

Here, iterations 3 and 4 will not work.

Continue - skips the current iteration in a running loop.

```
for (int i=0; i<=5; i++) {
    if (i==1)
        continue;
    print(i);
}
```

iteration 1 would be skipped

0, 2, 3, 4, 5

Any statement present after the continue statement would not be executed.

Operator Precedence Table

Use brackets to avoid confusion to solve any expression

Instead of 2 * 3 + 5/10 - 2 prefer writing it as ((2 * 3) + (5/10)) - 2