? operator:

Exp1 ? Exp2 : Exp3

↓ check (Exp 1 true) (Exp 1 false)

```
y = x > 9 ? 100 : 200 ;
```

Same as

```
if (x > 9)  y = 100;
else        y = 200;
```

& and * operators

& → memory address

pointer variable → specifically declared to hold a pointer to an object of the specified type.

→ Allow functions to modify contents
→ Support linked lists, binary trees
→ Dynamic data structures.

& → unary operators (returns memory of operand)

```
m = & Count;
```

* → Complement of &

* → unary returns the value of the object located at the address that follows it.

```
q = *m;
```
→ meaning count!

&, * → higher precedence than arithmetic operators except unary operator (-) → shares equal precedence.

```
int *a;
```
→ denote that a will point to a address. (hold an address)

```
char *ch;
```
→ ch is not a character but a pointer to a character.

base type → object of the base type.

mix pointer and non pointer declaration

```
int a, *b, Count;
```

sizeof → Compile time unary operator!

* gives same result as defined by typedef library
* returns type same as the type of size_t (loosely: unsigned int).

Comma operator

```
x = (y=3, y=4);
```

→ parentheses

x = 4
y = 4

① x set 3

② x set to 4! (Final) Answer.

```
x=1,2,3,4;
```

x = 1 → not 4

```
x=(1,2,3,4);
```

→ parentheses

Now x=4

Dot (.), arrow (→) operator

* Access individual elements of structures & unions (Compound DS under single name).
* Dot → used with structure/union
* Arrow → used with pointers to a structure.

```
struct employee {
    char name[80];
    int age;
    float wage;
} emp;

struct employee *p = &emp;
```

tag!

```
emp.wage = 123.23;

p→wage = 123.23;
```

(meaning dereference p to get emp.wage)

```
(*p).wage
```

[], () operators

() → increase precedence of operations inside it!

[] → array indexing!

Highest `()` `[]` `→` `.`

`!` `~` `++` `--` `-` (type) `*` `&` sizeof

`*` `/` `%`

`+` `-`

`<<` `>>`

`<` `<=` `>` `>=`

`==` `!=`

`&`

`^`

`|`

`&&`

`||`

`?` `:`

`=` `+=` `-=` `*=` `/=` etc..

Lowest `,`

`x = f1() + f2();`

"can't tell which one first"

Type Conversion

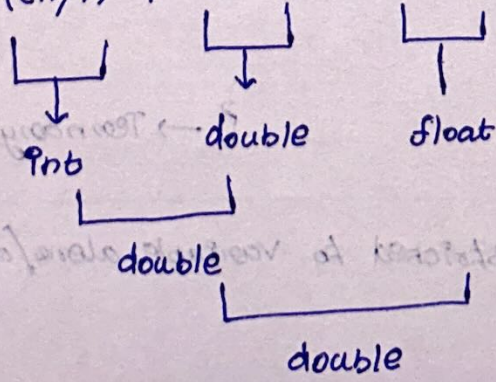
longdouble , other is also converted to long double

float , " float

exception: unsigned int ← can't be represented by a long.

(Both are converted to unsigned long).

```
char ch;
int i;
float f;
double d;
result = (ch/i) + (f*d) - (f+i);
```



"Longer type" - suitable ?

Cast

(float) x/2 ; → explicit.

Spacing & parenthesis – 'Easy to read'

```
x = 10/y ~(127/x);
x = 10 / y  ~((127)/x);
```
] Same!

* Additional space / parenthesis → don't slow down.
* Gives more clarification.

Statements

- Selection (if, switch)
- Iteration (while, for) ...
- Jump (break, Cont.. goto)
- Label (case, default)
- Block ({})
- Expression (valid expression)

true / false → Boolean expressions

If (Scalar - Int / Char / pointer / float / Bool)

rand() → <stdlib.h>

```
if ( )
if ( )
if ( )
else
```
) These two are associated!

C99 → 127 (atleast) – must be supported by Compilers.

? → Alternative

1 > 2 ? printf ("Hello") : printf ("Hi");

o/p : Hi

? → Ternary operator!

'It is not restricted to variable alone/assignment'

we can do anything!

Conditional expression!

* Any valid expression can control if (or) ? operator! (not just restricted to relational & logical operators as in PASCAL/BASIC)

true → non zero
false → zero

Division:

```
switch (expression) {
case constant 1 :
        :
        break ;
case constant 2 :
        :
        break ;
        :
default
        statement
}
```

Int/ Char Constants

Switch

* float not allowed!
* case % executed (matched) → until break / → until end of switch
* No match → default (optional)

C99: Atleast 1 or 2 statements. (case)

'Low : high efficiency' (no. of cases)

* switch → only test for equality
* No two case – identical expression constant!
* char – Automatically converted to int.

```
switch (n) {
case 1
    printf("H");
case 2
    printf("A");
    :
case 5
    printf("PRAKASH");
    break;
default
    printf("Hello");
```

(once true: execute until break or end of switch)

2
ARIPRAKASH
1
HARIPRAKASH
3
IPRAKASH

→ No break!

1

HARIPRAKASH Hello

Nested Switch

```
switch (n) {
case 1
    switch (n)
    {
      :
    }
```

outer correct doesn't mean inner also correct.

Inner – evaluate & follow the same!

Iteration

```
for (initialize; check; increment/decrement)
{
}
```

Check: Condition

variants:

```
for (x=0, y=0; x+y < 10; ++x)
{
}
```

× * * * * * *

(79) Page

29

14 — 14

```
for ( : x < 10; ) {
```
↓ no initi ↓ no incr/decr

for (;;)

for (; ;) ⟶ ∞ loop;

* A statement may be empty!
* Body of for loop – also may be empty!

"Show" ≠ " Show"
↳ 'Remove space'

```
for ( ;*str == ' '; str++);
```

Advance leading spaces!

Note: arr[20] = "Hello";

* we get ordinary pointer! that points somewhere
* Incrementing that pointer leads to lvalue error

∴ it is a pointer to an array!

array: non modifiable left values (lvalues) ⟶ No operators modify the array itself.

Problem: Create explicit pointers

eg: char *pt = arr;

for (; *str == ' '; str++); str[10] = "Hello";

↳ lvalue error!

```
char arr[15] = "  Hello";
char *a = arr;
for (; *a == ' '; a++);
printf ("%s", a);
```

Ans: Hello

'No whitespace-leading'

```
char *arr = "  Hello";
for (; *arr == ' '; arr++);
printf ("%s", arr);
```

Time delay

* for (k=0; k < Some_value; k++) → catup time

Embedded programming!

declare within for loop

for (int i=0; i<10; i++)

while loop

Padding → fill with spaces, finally \0

```
while (i < lenstring)
    space
\0
```

do while

```
do {
   :
} while(...)
```

do atleast once, then check!

Continue ← Jump statements → Return, goto, break, exit()

Return:

* Return from a function (Jump back to where took off!)
* Return – may or may not have associated value
* Return – only when non void function!
* In void → Return without value is used.

Return function → must return a value!

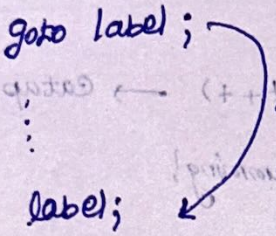
* Function stops at 1st return!
* void function → No return value.

return expression;

---

goto

* occasionally has its uses – narrow situations – jumping out of set of nested loops.
* Can't jump b/w functions!
* label → identifier!



Break

* terminate a case in switch statement
* Terminate a loop. (only inside loop – where break is)!

kbhit() → returns 0 → if key is not pressed!
else nonzero value.

'conio.h'

'Break causes exit from innerloop alone'!

exit() → not a jump statement

* Function in std library
* Termination of the entire program return to operating system.
* general form:

void exit (int return_Code);

* Usually Zero is used as return code!

* Also: errno.h : EXIT_SUCCESS , EXIT_FAILURE
(macros for return code.)

<stdlib.h> → exit()
(use for debugging!)

exit() → quit the program!

Continue!

* Doesn't terminate → Just skips to next iteration.
* Cause increment; But after continue statement won't be executed in that particular iteration alone!

---

eg: If space: don't print - Just skip!

Expression statements: Valid expression followed by semicolon

eg:
```
fun();   // function call
a=a+b;   // assignment
b+5();   // valid - no use (as never stored/used)
;        // empty statement!
```

Block Statement

```
int i=5;
{
  i=120;
  printf("%d", i);   → 120
}
printf("%d", i);     → 120
```

→ Free standing block statement!

not like func - creating local values!

'Patterns'

```
* * * * *   i=0
*       *
*       *   j=end.
*       *
*       *
* * * * *   i=end
```

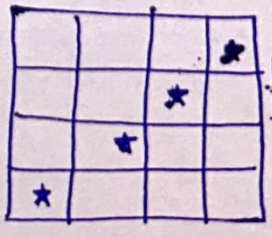
j=0

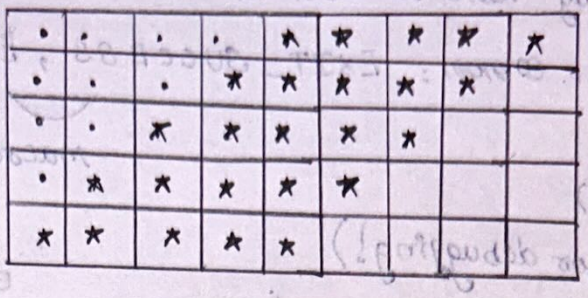
hollow square with diagonal

i==j → left diagonal

```
*
    *
        *
            *
```

(0,4) = 5
(1,3) = 5
(2,2) = 5
(3,1) = 5.





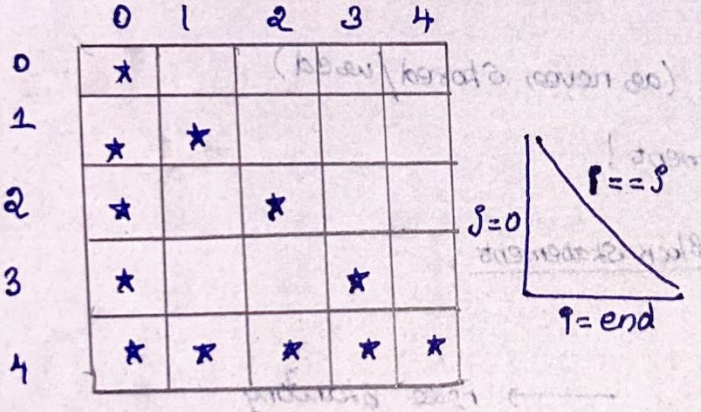
For the loop 9+S as usual!

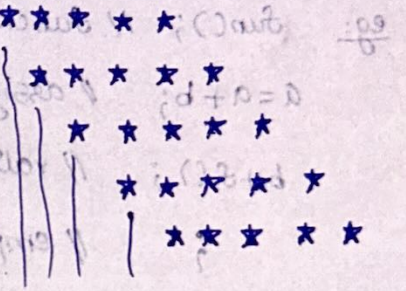
```
for(9=0; 9<n; 9++)
{
    for (K=0; K<n-1-9; K++)
    printf (" ");
    for (S=0; S<n; S++)
        printf ("* ");
    printf ("\n");
}
```

→ For 9, S 9+S as usual!

Square!

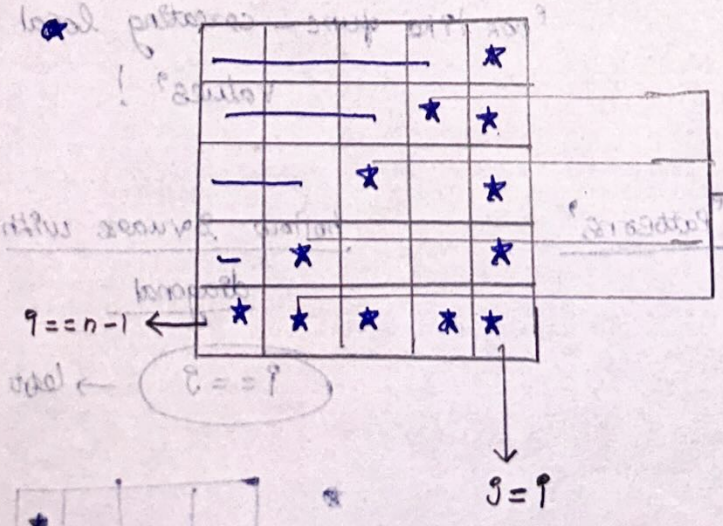
mirrored – Rhombus –

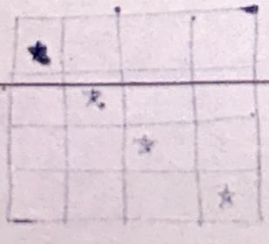


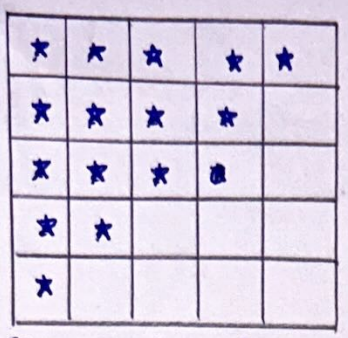


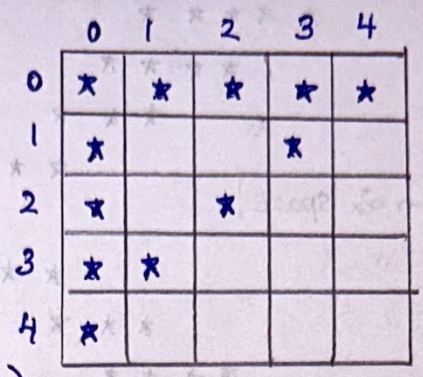
(K=0; K<9; K++)

(9==n-1) || (S==0) || (9==S)







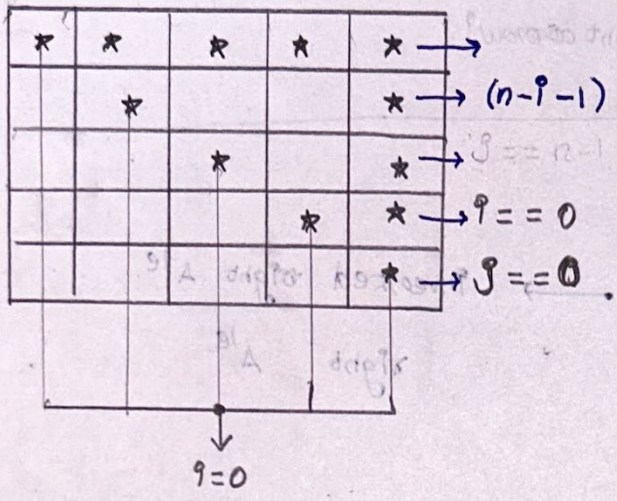


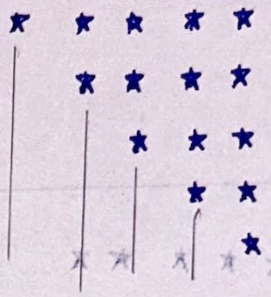
diagonal : $i+j == (n-1)$

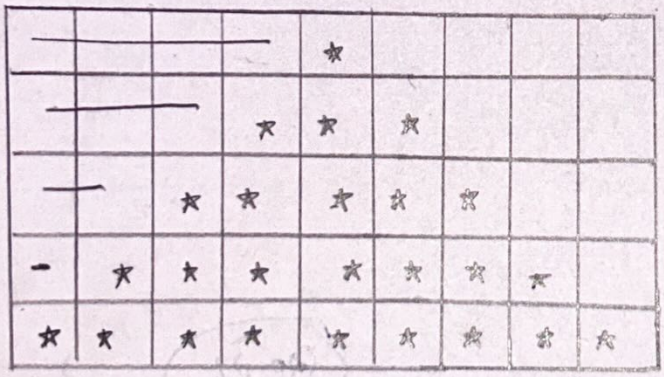
$i == 0$

$j == 0$

(for $i=0; i<n; i++$
for $j=0; j<n-i; j++$)







($i=0; i<n; i++$)

$k=0; k<n-i-1; k++$

space

$j=0; j<(2*i)+1; j++$

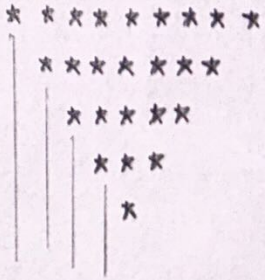
*

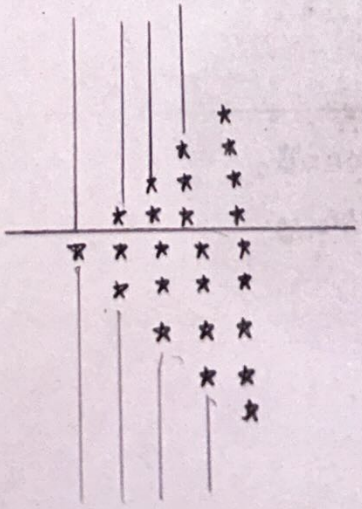
hollow: $j=0$, $i=n-1$, $j=(2*i)$

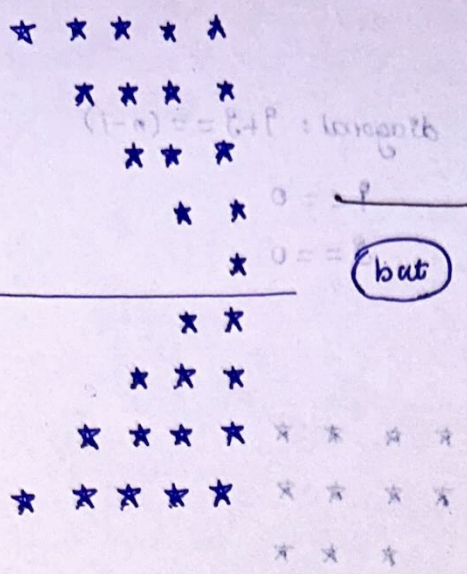
inverted!

hollow

$i == 0$

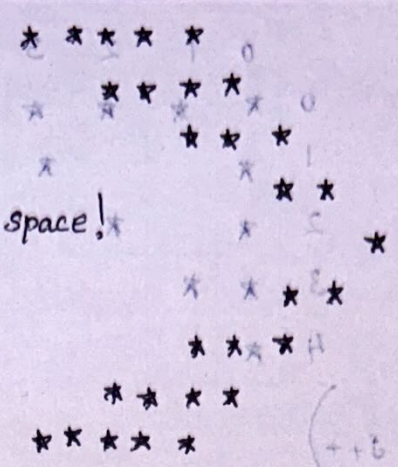




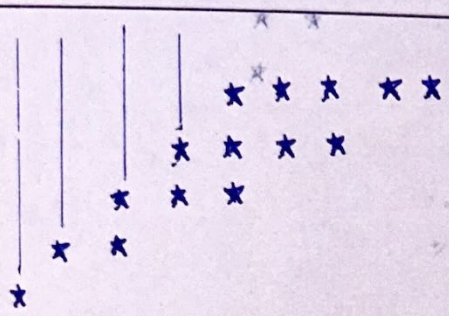


→ 2x space!

but

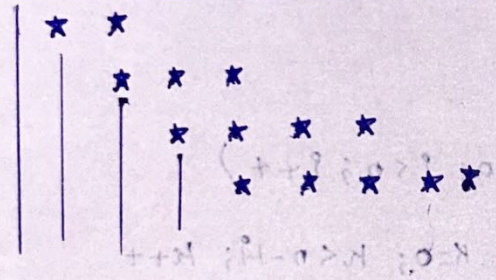


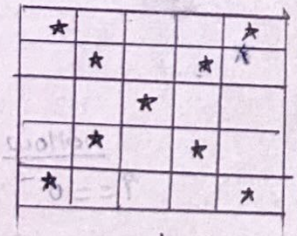
Right arrow?



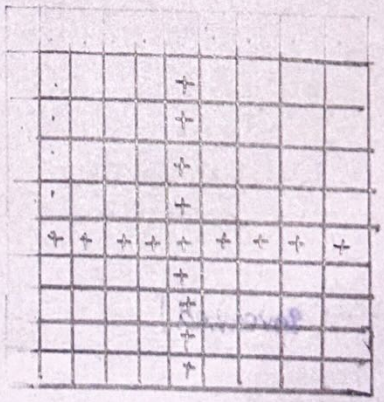
→ Inverted right Δle.

right Δle.



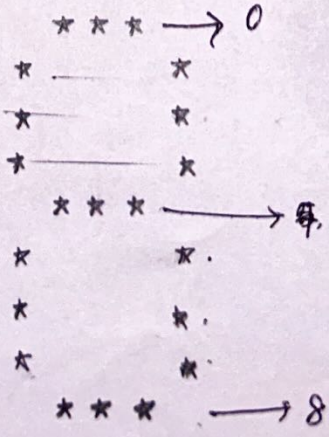


$i==j$ | $i+j == n-1$



$(2n-1)/2$

$i=1$ &(



→ 0

→ 8

n=5, Column=5,

Rows = 10.

Number pattern 14

```
5 5 5 5 5
5 4 4 4 4
5 4 3 3 3
5 4 3 2 2
5 4 3 2 1
```

Number pattern 15

```
1 2 3 4 5
2 3 4 5 5
3 4 5 5 5
4 5 5 5 5
5 5 5 5 5
```

Pattern 16

```
1 2 3 4 5
2 3 4 5 1
3 4 5 2 1
4 5 3 2 1
5 4 3 2 1
```

left shift.

Pattern 17

```
1 2 3 4 5
2 1 2 3 4
3 2 1 2 3
4 3 2 1 2
5 4 3 2 1
```

```
5 5 5 5 5 5 5 5 5
5 4 4 4 4 4 4 4 5
5 4 3 3 3 3 3 4 5
5 4 3 2 2 2 3 4 5
5 4 3 2 1 2 3 4 5
5 4 3 2 2 2 3 4 5
5 4 3 3 3 3 3 4 5
5 4 4 4 4 4 4 4 5
5 5 5 5 5 5 5 5 5
```

```
5 5 5 5 5 5 5 5 5
  4 4 4 4 4 4 4
    3 3 3 3 3
      2 2 2
        1
```

Pattern 18

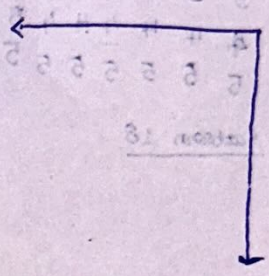
```
5 5 5 5 5 5 5 5 5
5 4 4 4 4 4 4 4 5
5 4 3 3 3 3 3 4 5
5 4 3 2 2 2 3 4 5
5 4 3 2 1 2 3 4 5
5 4 3 2 2 2 3 4 5
5 4 3 3 3 3 3 4 5
5 4 4 4 4 4 4 4 5
5 5 5 5 5 5 5 5 5
```

```
3 4 5
  4 5
    5
```

```
9 8 7 6 5 4 3 2
9 8 7 6 5 4 3
9 8 7 6 5 4
9 8 7 6 5
9 8 7 6
9 8 7
9 8
9
```

1	2	3	4	5	6	7	8	9	10
36	37	38	39	40	41	42	43	44	11
35	64	65	66	67	68	69	70	45	12
34	63	84	85	86	87	88	71	46	13
33	62	83	96	97	98	89	72	47	14
32	61	82	95	100	99	90	73	48	15
31	60	81	94	93	92	91	74	49	16
30	59	80	79	78	77	76	75	50	17
29	58	57	56	55	54	53	52	51	18
28	27	26	25	24	23	22	21	20	19

	0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9	10
1	36	37	38	39	40	41	42	43	44	11
2	35	64	65	66	67	68	69	70	45	12
3	34	63	84	85	86	87	88	71	46	13
4	33	62	83	96	97	98	89	72	47	14
5	32	61	82	95	100	99	90	73	48	15
6	31	60	81	94	93	92	91	74	49	16
7	30	59	80	79	78	77	76	75	50	17
8	29	58	57	56	55	54	53	52	51	18
9	28	27	26	25	24	23	22	21	20	19



① [0][0] to [0][9] | [1][9] to [9,9] | [9,8] to [9,0] | [9,0], [0,0]

'Left to right' | 'Right to left'

more detail

1) Go right
2) Down
3) Go left
4) Go up!

L→R	down	right to left	up
0,0 to 0,9	1,9 to 9,9	9,8 to 9,0	8,0 to 1,0
1,1 to 1,8	2,8 to 8,8	8,7 to 8,1	7,1 to 2,1
2,2 to 2,7	3,7 to 7,7	7,6 to 7,2	6,2 to 3,2
3,3 to 3,6	4,6 to 6,6	6,5 to 6,3	5,3 to 4,3
4,4 to 4,5	5,5	5,4	

9,8 i=0

1) j goes to 9 — arr[9][j] , i++

j goes to — arr[j][↓ 1 to 9

left

left = 0

right =

left right

→ from 0 to 9

① arr[left][j]

arr[j][down]

→ j =

1	2	3	4	5
16	17	18	19	6
15	24	25	20	7
14	23	22	21	8
13	12	11	10	9

left = 0
right = 4
top = 0
bottom = 4

* left to right
* Top to down
* Right to left.
* down to Top

① arr[left][j] [j = left (0) to right (4)]

arr[j][right] [j from top+1 to bottom]

arr[bottom][j] → [j from right+1 to left]

arr[j][left] . [j from bottom-1 to top+1]

Now left = 1 , right = 3

top = 1 , bottom = 3.

It seems like 2 variables enough

'As square' – Top = left
Right = bottom.

1	2	3	4	5
16	17	18	19	6
15	24	25	20	7
14	23	22	21	8
13	12	11	10	9

① left = 0, right = 4

0 to 4

acon [left] [3] → from left to right

acon [3] [right] → from left+1 (1) to right (4)

acon [right] [3] → from right − 1 (3) to left

acon [3] [left] → from right − 1 to left + 1.

② left = 1, right = 3

acon [left] [3] → from 1 to 3

acon [3] [right] → from left + 1 to right

acon [right] [3] → from right − 1 to left

acon [3] [left] → from right − 1 to left + 1.

(2 steps)

left = 2, right = 2.

* acon [2] [2] = 2 [2 to 2]
* acon [ ] [3 to 2] → No!
* acon [ ] [1 to 2] → No!) decrement!
* acon [ ] [1 to 3] → No

N/2 times

① 0,0 to 0,4
1,4 to 4,4
4,3 to 4,0
3,0 to 0,0

For: 10/2 = 5 times. 0 to 4 (5 times) ⇒ n/2
5/2 = 2

② 1,1 to 1,3
2,3 to 3,3
3,2 to 3,1
2,1 to 2,2

No. of steps : N/2.

0	1				
1	2	4			
2	1	3	5		
3	2	4	6	8	
4	1	3	5	7	9
	0	1	2	3	4

1								
1	3	1						
1	3	5	3	1				
1	3	5	7	5	3	1		
1	3	5	7	9	7	5	3	1

$\leq (2*i)+1$ $(4*2-1)$ $(3*2-1)$

① $2*(0)-1$

② $2*(1-0)-1 = 2-1 = 1$

③ $2*(2-0)-1 = 4-1 = 3$

④ $2*(3-0)-1 = 6-1 = 5$

⑤ $2*(4)-1 = 7$

$(2*(7-3))-1$

↓

No need

$2*i-1$ to 1

Start Stop

2								
2	4	2						
2	4	6	4	2				
2	4	6	8	6	4	2		
2	4	6	8	10	8	6	4	2

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

			1					
		3	2					
$i+1 \leftarrow$	4	5	4	3				
(Start)	5	6	7	6	5	4		
	6	7	8	9	8	7	6	5

Start from 3rd row

4 $\longrightarrow$ $2*2$ $(i-1)*2$

5 6 $\longrightarrow 3*2$

6 7 8 $\longrightarrow 4*2$

7 8 9 10 $\longrightarrow 5*2$

↓ Start from $i+1$

↓ end upto.

4

5 6

6 7 8

↓

$i >= 3$ &&

$i <= (2*(i-1))$

easy? → Split

0	numbers	1	numbers
0		2	
1		3	
2		4	
3		5	

1					
3	2				
5	4	3			
7	6	5	4		
9	8	7	6	5	

↓

Start $2*i-1$, end $(2*i-1)$

end $2*i - i$

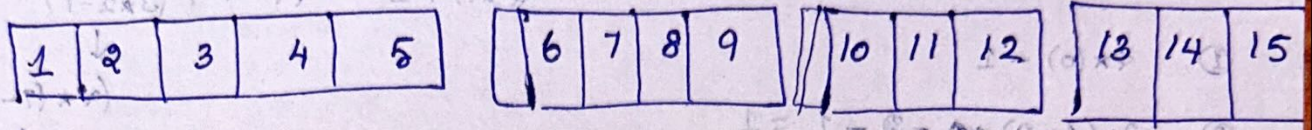
$2 - 1 = 1$

$4 - 2 = 2$

$6 - 3 = 3$

$8 - 4 = 4$

	0	1	2	3	4
0	1				
1	2	6			
2	3	7	10		
3	4	8	11	13	
4	5	9	12	14	15




```
for (i=0; i<n; i++)
    for (j=0; j<=i; j++)
        array[j][i]
```

① array [0] [0] 1

② array [0] [1] → 2

[0] [1] → 6

array [0] [2] → 3

[1] [2] → 7

[2] [2] → 10

note: need to leave - empty spaces (in arrays)

Efficient method - Inplace execution

1	0				
2	6				
3	7	10			
4	8	11	13		
5	9	12	14	15	
↓	↓	↓	↓	↓	
i	i+4	i+3	i+2	i+1	

1					
2	7				
3	8	12			
4	9	13	16		
5	10	14	17	19	
6	11	15	18	20	21
↓	↓	↓	↓	↓	↓
i	i+5	i+4	i+3	i+2	i+1

```
1
2  6
3  7  10
4  8  11  13
5  9  12  14  15
↓  ↓  ↓   ↓   ↓
9  9=9+4  9=9+3  9=9+2  9=9+1.
```

```
printf ("%d", 9)
for (9=2, K=n-1, l=9+K ;
     9<=9 ; 9++ , K--,
     l = K+l)
printf ("%d", l);
```

* Initialize incrementing value by $n-1$
* Then each time Value = previous value + incrementing value
  ↳ decreased by 1 each time.

```
1
2    4
7    11   16
22   29   37   46
56   67   79   92   106
```

1+1 = 2 | 16+6 = 22
2+2 = 4 | 22+7 = 29
4+8 = 7 | 37 = 29+8
7+4 = 11 | 37+9 = 46
11+5 = 16

```
1                  ⟶ 1,1
3    2             ⟶ 2,2
4    5    6        ⟶ 3,3
10   9    8    7   ⟶ 4,4
11   12   13   14  15. ⟶ 5,5.
```

∴ max = ~~9~~ 9 → Index

eg:

9=2 ⟶ 3 2

9=4 ⟶ 10 9 8 7

Stop: 9+1

Not working

K = 2 + 9 - 1

3 to 2

K = 7 + 9 - 1 = 10

10 to 7

```
if (9 % 2 != 0)
    9+9 ⟶ print
else
```

⟶ see code!

```
for(i=1; i<=n; i++)
{
  if (i % 2 != 0)
  {
    for (j=1; j<=i; j++, k++)
    {
      printf ("%d", k);
    }
  }
  else  temp=k;
  {  for (j=k+i-1; j>=temp; j--, k++)
    {
      printf ("%d", j)
    }
  }
}
```

Logic

```
1
3  2
4  5  6
10 9  8  7
11 12 13 14 15
```

i+j

K=1
K=2
K=3
⋮

```
   3   2
  10   9   8   7
```

Continuation of prev. row values. (Let take K)

```
1                  → K=2
3  2               → K=4
4  5  6            → K=7
10 9  8  7         → K=11
11 12 13 14 15     → K=16
```

At the end.

using K=2 (Row to 2 ele, 4 → 4 elements)

j = k+i-1	to	j=k
3	to	2
10	to	7

See pattern 51.