(*) Things to know in c++.

⇒ Generic Skeleton

-library inclusion ⟶ #include <iostream>

main function ⟶ int main()
{ //statements }

⇒ cout to display output in c++.
function from iostream library.
belongs to std namespace

↑
You have to mention / specify.

for example

#include <iostream>
using namespace std;
int main()
{ cout << "Hello world";
return 0;
}

OR

#include <iostream>
int main()
{ std::cout
<< "Hello";
return 0;
}

→ newline → "\n" eg. → cout << "Hello \n";
cout << "world";
output: Hello
world

⇒ cin taking user input.

#include ...
using ...
int ... ()
{ int x;
cin >> x;
cout << "value of x : " << x; ... }

This is HOW you can take input from the user.

cin and cout are the functions of iostream library.
So you have to include this library, whenever you wanted to use cin and cout.

⇒ Note :

To make the process more convenient, there's a shortcut that allows you to include almost all standard libraries at once using

#include <bits/stdc++.h>

```
#include<bits/stdc++.h>
using namespace std;
int main()
{ int x;
  int y;
  cin >> x >> y;
    cout << "value of x and Y : " << x << " "
                                 << y;
  return 0;
}
```

input : 10 20
output : value of x and y 10 20.

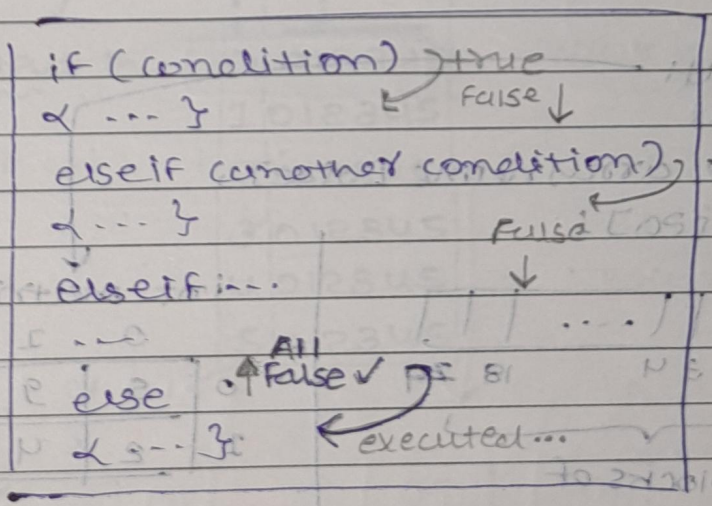
⇒ Datatypes in c++.

Primary (32 bits)	user defined	string
int (4)	class	↑ not a user defined datatype / not a built in
char (1)	structure	
float (4)	union	
1 bit Bool (1)	enum	
Double (8)	typedef	
void		

⇒ if else

```
....
if (age>18 ){
cout << " you are an adult " ; }
else
cout << " ..... " ; }
...
```

if elseif else



⇒ switch case

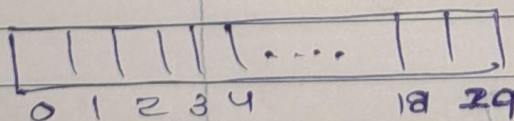
```
....
int day = 5;
switch (day)
{ case 1:  cout << "monday";
  break;
  case 2:  cout << "tuesday";
  break;
    :
    :
  case 7:  cout << "sunday";
  break;
  default: cout << "Invalid";
}
....
```

output: friday.

⇒ Arrays & Strings

one multi dimensional

```
datatype arrayname[size];
int arr[20];
```



20 blocks of memory

```
int matrix[2][2];
```

	0	1
0	5	9
1	2	4

```
cout << matrix[0][1];
```

output: 9.

String s = "Jaymin"

J	a	y	m	i	n	\0
0	1					

cout << s[1];

output :- a.

⇒ **For loops**

initialization condition increement/ decrement.

```
for (int i=0; i<5; i++)
{ cout<<i; }
```

output :- 01234

⇒ **Nested loops**

```
for (int i=0; i<=5; i++)
  { cout<<i;
    for (int j=0; j<=5; j++)
      { cout<<j; }
    cout<<"\n";
  }
```

output :-

i	j
0	012345
1	012345
2	012345
3	012345
4	012345
5	012345

⇒ **While loop**

```
while(condition)
{ // statement }
```

for example

```
while (n!=0)
{ factorial *= n;
  n--;
}
```

⇒ functions → Pass by value
→ Pass by reference (address)

* Actual → parameters
formal

Actual parameters
add (m, n)

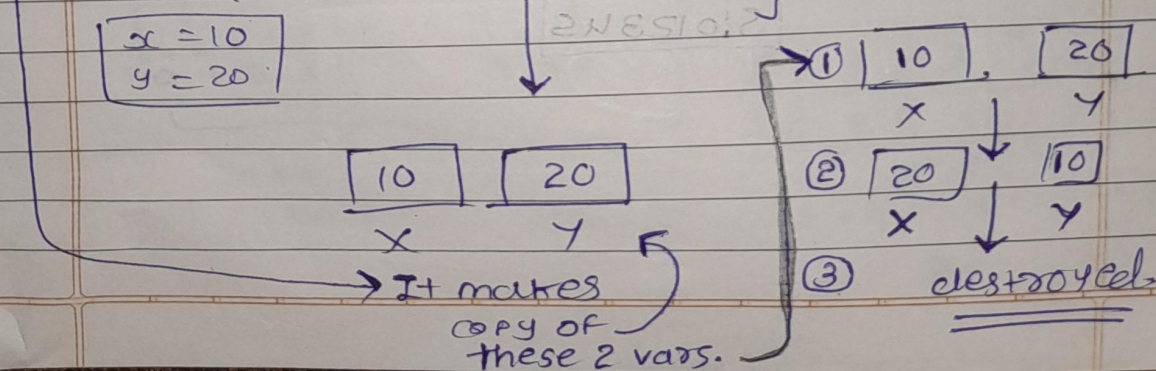
formal parameters
int add (int a, int b)
{ return a+b; }

* Pass/call by value — **swapping**

```
int x=10, y=20;
fun(x, y);
cout << x << y;
```

x = 10
y = 20

```
int fun(int x, int y)
{ x = 20;
  y = 10;
}
```



Here we are passing addresses instead of copy of values.

* pass / call by reference

swapping ✓

```
int x=10, y=20;
fun(&x,&y);
cout<<x<<y;
```

x=20
y=10

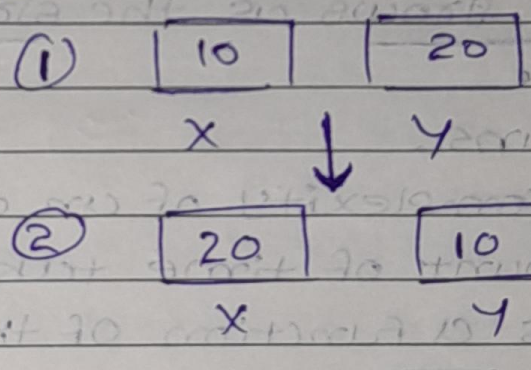
we are passing address of x & y using "&" (address of) operator.

```
int fun(int *x, int *y)
{
  *x=20;
  *y=10;
}
```

Pointers as formal parameters

Here * means dereference operator, i.e we are accessing the value at address.

So Here

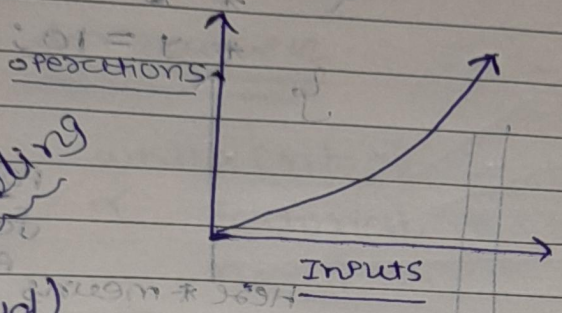


* Time & Space Complexity. (Next page)

Time complexity (TC)

* What does it mean?

→ TC ≠ actual time taken by system. (not means) TC ≠ runtime



for better understanding (Read)

→ Time complexity is a behavior or a function which takes number of inputs

→ so it is the analysis of how an algorithm's runtime grows as the size of its input increases.

→ That means.

Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the size of the input.

OR

✓ Time complexity is a measure of how the running time of an algorithm increases as the number of input (size) grows.

Here algorithm's runtime doesn't mean the actual execution time of the machine on which this algorithm is running on.

That means System's execution time ≠ Algorithm's Runtime.

Notations

(1) Ω (omega) — Best case — Big / small
(2) Θ (theta) — Avg. case — Big / small
(3) O — Worst case — Big / small ✓

Big O small "o" — Alphabet "O"

↑ Most popular

Big O Notation (O) → stands for "order of"

Rules:
1. avoid including constant terms
2. avoid lower values
3. always worst case.

→ **Best case** :- this term refers to the case where code takes the least amount of time to get executed.
(lower bound)

→ **Worst case** :- this term refers to the case where " " " maximum amount "
(upper bound)

→ **Avg. case** :- This term is pretty self explanatory case between the worst & the best.

So let's say we have a function (mathematical)

$t(n) = 3n^2 + 5n + 9$

1. avoid constant terms ✓

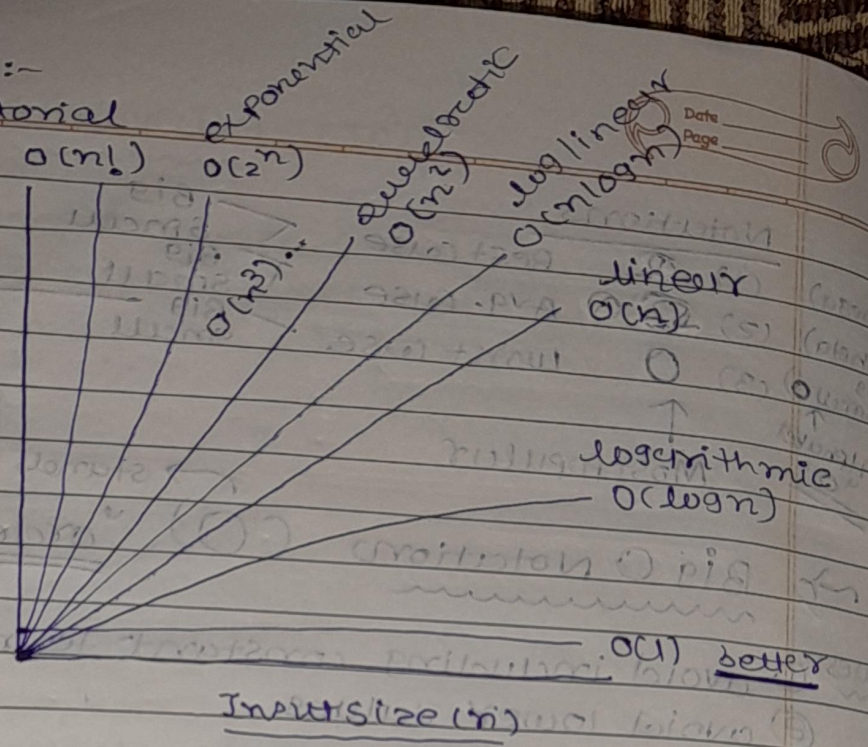
$f(n) = n^2 + n$

2. avoid lower values ✓

n^2 → $O(n^2)$ will be our time complexity

↑ Worst case

Worst :-



Common time complexities :-

O(1) (constant time complexity (order of 1)
(increase)
→ Input size ⇒ number of operations (constant)

example function to print "Hello"
accessing an element of an array
arr[5] = {10, 20, 30, 40, 50};
→ cout << arr[3];

O(n) linear time complexity (order of n)
Input size = number of operations
for example for loop
linear search

1	2	3	4	5

n = 5
search = 5 (worst case) = n

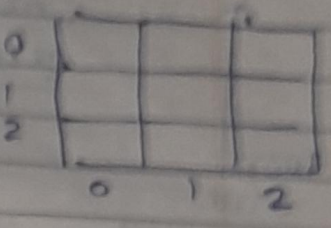
Remember :- (assuming) while calculating time complexity

Always take larger value of No. (consider)

$O(n^2)$ example nested loops // matrix reversal etc.

↓

quadratic time complexity.
(order of n^2)

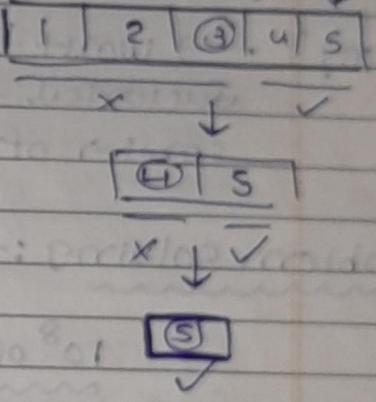


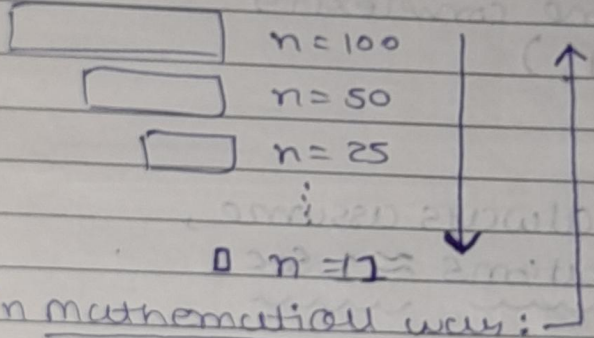
$O(\log n)$ better than $O(n)$.

↳ logarithmic time complexity
(order of $\log n$)

example Binary Search → sorting

sorted array:





in mathematical way:

$1 \times 2 \times 2 \times 2 \ldots$ (x times)

$1 \times 2^x = 100 = n$

$1 \times 2^x = n$

$x = \log_2 n$

$\boxed{x = \log n}$

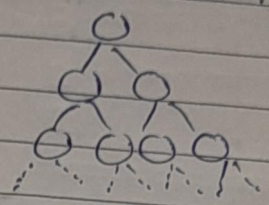
$O(n \log n)$ loglinear time complexity
(order of $n \log n$)

example sorting algorithms

And then follow 3 Rules. (Udemy pg 42 ✓)

$O(2^n)$ exponential time complexity
(order of 2^n) → brute force approach

example Recursion



worse time complexity

↳ we use Dynamic Programming to make it better.

$O(n!)$ → worst time complexity
factorial time complexity
(order of $n!$)

Problem solving :-

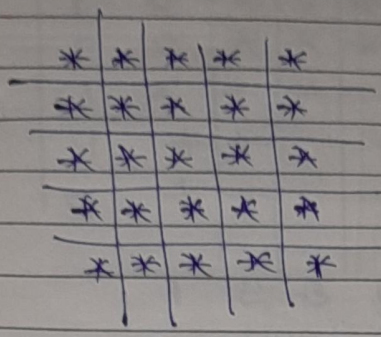
always assume,
10^8 operations ≈ 1 sec

$n > 10^8$	$O(1)$, $O(\log n)$
$n \leq 10^8$	$O(n)$
$n \leq 10^6$	$O(n \log n)$ → sorting allowed
$n \leq 10^4$	$O(n^2)$
$n \leq 500$	$O(n^3)$
$n \leq 25$	$O(2^n)$ → bruteforce → recursion
$n \leq 12$	$O(n!)$

Always Use Pen & Paper
While Coding ↑ whenever it is needed!

Patterns : Build w logical thinking.

(1)



	ROWS	Ittical columns
start	1	1 2 3 4 5
	2	1 2 3 4 5
	3	1 2 3 4 5
	4	1 2 3 4 5
end	5	1 2 3 4 5

start end

start ↓ end ↓

```
for (rows = 1 ; rows <= 5 ; rows++)
{
    start ↓   end ↓
    for (cols = 1 ; cols <= 5 ; cols++)
    {
        cout << "*" ;
    }
    cout << endl ;
}
```

* Inner loop → columns
* outer loop → rows
* Print stars always in inner loop.
  focus on columns
  and connect / relate with rows and make proper conditions (in for loop)

* Understand the pattern and think how will you create indexes of rows and columns and according that create for loop for rows (outer loop) → give a starting point and condition to end that loop.
  and another inner for loop for columns → give proper starting point according to indexes and relate/connect with rows
  ↓
  make proper conditions to end that loop. (And also to create proper pattern)

(2)

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

ROWS	filled columns
start 1	1 →
2	1 2
3	1 2 3
4	1 2 3 4
end 5	1 2 3 4 5
	↑ start

condition ↓

columns <= rows

```
for (rows = 1 ; rows <= 5 ; rows++)
{ for (cols = 1 ; cols <= rows ; cols++)
  { cout << "*" ; }
  cout << endl ;
}
```

(3)

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

same as (2)

```
cout << cols ;
```

(4)

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

same as (2) & (3)

```
cout << rows ;
```

(5)

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

ROWS	filled columns (forming condition earlier)
start 1	5 4 3 2 1
2	5 4 3 2
3	5 4 3
4	5 4
end 5	5

cols >= rows

cols = 5 (starting point)

cols--

inner loop.

```
=>
for (rows=1; rows<=5; rows++)
{ for (cols=5; cols>=rows; cols--)
  { cout << " * "; }
  cout << endl;
}
```

(6)

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

ROWS	filled columns
start 5	1 2 3 4 5
4	1 2 3 4
3	1 2 3
2	1 2
end 1	1

cols <= rows

```
for (rows=5; rows>=1; row--)
{ for (cols=1; cols<=rows; cols++)
  { cout << cols; }
  cout << endl;
}
```

(7)

	filled columns
start (1)	(5)
(2)	45(6)
(3)	3456(7)
(4)	234567(8)
end (5)	12345678(9)

1st Relation :
1+4 = 5
2+4 = 6
...
5+4 = 9
rows + 4 ≠ cols

cols <= rows+4

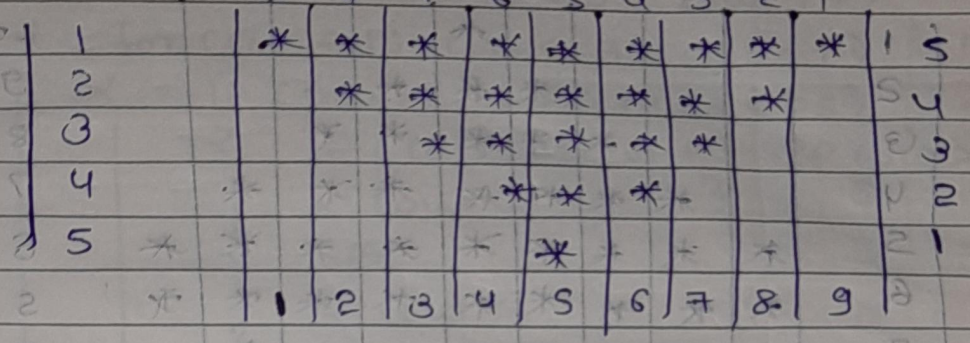
cols start = 1

1st Relation → condition to end inner loop
→ cols <= rows + 4

2nd Relation
cols >= 6 - rows because 6-1 = 5, 6-2 = 4, ... 6-5 = 3
↑ (n+1) value
6 - rows = cols
so we will use it in if condition (to check when to or when not to print)

```
for (rows = 1; rows <= 5; rows++)
  for (cols = 1; cols <= rows+4; cols++)
    if (cols >= 6-rows)
      cout << "*";
    else
      cout << " ";
  cout << endl;
```

(8)



rows	filled columns	
✓ 1	1 to 9	9 to 1 ✓
✓ 2	2 to 8	8 to 2 ✓
✓ 3	3 to 7	7 to 3 ✓
✓ 4	4 to 6	6 to 4 ✓
✓ 5	5	5 to 5 ✓

to check when to print

1st Relation :- rows <= cols / cols >= rows ✓

2nd Relation :- 10 - rows => cols start point

10-1=9
10-2=8
10-5=5

(cols <= 10-rows) ✓

Code :-

```
for (rows=1; rows<=5; rows++)
{ for (cols=9; cols>=rows; cols--)
  { if (cols<=10-rows)
    { cout<<"*"; }
    else
    { cout<<" "; }
  }
cout<<endl;
}
```

Rows	filled columns
1	5 to 5
2	4 to 6
3	3 to 7
4	2 to 8
5	1 to 9
6	1 to 9
7	2 to 8
8	3 to 7
9	4 to 6
10	5 to 5

if (1) (1 to 5)

cols <= rows+4

cols >= 6-rows,

if (2) (6 to 10)

cols <= 10-(rows-5)

cols >= rows-5

Here we will start & end loop from start to end only

means rows will be from 1 to 10

columns " " 1 to 9

and both will be incremented,

conditions will be applied using

(2) if conditions 1st for △

2nd for ▽

```
for (rows = 1; rows <= 10; rows++)
{  for (cols = 1; cols <= 9; cols++)
   {  if (rows >= 1 && rows <= 5)
      {  if (cols <= rows + 4 && cols >= 6 - rows)
         {  cout << "*"; }
         else
         {  cout << " "; }
      }
      else if (rows > 5 && rows <= 10)
      {  if (cols <= 10 - (rows - 5) && cols >=
                                   rows - 5)
         {  cout << "*"; }
         else
         {  cout << " "; }
      }
   }
}
```

(10)

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

rows	Fill col columns	
1	1	①
2	1 2	
3	1 2 3	
4	1 2 3 4	
5	1 2 3 4 5	
6	1 2 3 4	②
7	1 2 3	
8	1 2	
9	1	

rows → start = 1, end = 9 cols → start = 1, end = 5

② conditions

rows [(1 to 5)
       (6 to 9)

[cols <= rows

cols <= 10 - rows

(11)

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

rows	filled columns
1	1
2	1 2
3	1 2 3
4	1 2 3 4
5	1 2 3 4 5

Columns <= rows

Same like (2)/(3)/(4) Here for printing 0 and 1 we will try to use some logic

odd row → starts from 1 < odd column → 1, even column → 0
even row → starts from 0 < odd column → 0, even column → 1

```
for (ROWS 1 to 5)
{  for (columns 1 to cols<=rows)
   {  if (rows%2==0) // even row
      {  if (cols%2==0) //even col.
         {  cout<<"1";}
         else  // odd col.
         {  cout<<"0";}
      }
      else  // odd row
      {  if (cols%2==0) // even col
         {  cout<<"0";}
         else // odd col
         {  cout<<"1";}
      }
   cout<<endl;
}
```

(12)

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

	rows	~~filled~~ columns	
4	1	1 | 8	8 | 1
3	2	1 2 | 7 8	8 7 | 2 1
2	3	1 2 3 | 6 7 8	8 7 6 | 3 2 1
1	4	1 2 3 4 | 5 6 7 8	8 7 6 5 | 4 3 2 1

we will print only when

1st condition :- columns >= 9 - rows

(or) 2nd " " :- columns <= rows

Here we will take count variable because we have to decrement the values ~~as~~ while we shift to right side of this pattern, How? Let's see the code:

```
int count = 0;
for (rows=1; rows<=4; rows++)
{  for (cols=1; cols<=8; cols++)
   {   if (cols<=rows)          // left side
       {  count++;              // incrementing
          cout << count;        // printing
       }
       else if (cols>=9-rows)   // right side
       {  cout << count;        // printing
          count--;              // decrementing
       }
       else
       {  cout << " "; }        // blank spaces. (empty)
   } cout << endl;
}
```

(13)

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

```
for (rows 1 to 5)++
{ for (cols 1 to <=rows)
  { count++;
    cout << count << " ";
  }
  cout << endl;
}
```

Same as pattern ② & ③ & ④
- 1 : just take counter variable and increment it.

(14)

1	A				
2	A	B			
3	A	B	C		
4	A	B	C	D	
5	A	B	C	D	E
	1	2	3	4	5

```
~~char num = 65;~~ // A
for (i to 5) ++
  char num = 65;
{ for (cols 1 to <=rows)++
  { cout << num;
    num++;
  }
  cout << endl;
}
```

ASCII values A to Z → 65 to 90
a to z → 97 to 122

(15)

	5	4	3	2	1
1	A	B	C	D	E
2	A	B	C	D	
3	A	B	C		
4	A	B			
5	A				

```
for (rows=1; rows<=5; rows++
  char num = 65;
{ for (cols=5; cols>=rows; col
  { cout << num;
    num++;
  }
  cout << endl;
}
```

(16)

1	A				
2	B	B			
3	C	C	C		
4	D	D	D	D	
5	E	E	E	E	E
	1	2	3	4	5

```
char num = 65;
for (rows 1 to 5) ++
{ for (cols 1 to i <= rows) ++
{ cout << num; }
  num++;
  cout << endl;
}
```

(17)

1				A			
2			A	B	A		
3		A	B	C	B	A	
4	A	B	C	D	C	B	A
5							
	1	2	3	4	5	6	7

rows	cols
1	4
2	3 4 5
3	2 to 6
4	1 to 7

cols >= 5-rows
cols <= rows+3

```
For (1 to 4) ++
{ char num = 65;
  for (1 to cols <= rows+3)
  { if (cols >= 5-rows)
    { cout << num;
      if (cols < 4)
      { num++; }
      else
      { num--; }
    }
    else
    { cout << " "; }
  }
  cout << endl;
}
```

(18)

1	E				
2	D	E			
3	C	D	E		
4	B	C	D	E	
5	A	B	C	D	E
	1	2	3	4	5

(cols <= rows)

```
char num = 69;
for(int rows=1; rows<=5; rows++)
{  char temp = num;
   for(int cols=1; cols<=rows; cols++)
   {  cout << temp;
      temp++;
   }
   num--;
   cout<<endl;
}
```

(19)

1	*	*	*	*	*	*	*	*	*	*
2	*	*	*	*			*	*	*	*
3	*	*	*					*	*	*
4	*	*							*	*
5	*									*
6	*									*
7	*	*							*	*
8	*	*	*					*	*	*
9	*	*	*	*			*	*	*	*
10	*	*	*	*	*	*	*	*	*	*
	1	2	3	4	5	6	7	8	9	10

when we will print space " " (empty) → Another way

rows	columns
1	-
2	5 6
3	4 5 6 7
4	3 4 5 6 7 8
5	2 3 4 5 6 7 8 9
6	2 3 4 5 6 7 8 9
7	3 4 5 6 7 8
8	4 5 6 7
9	5 6
10	-

rows 1 to 5
cols <= rows+4
ff
cols >= 7-rows

rows 6 to 10
cols <= 15-rows
ff
cols >= rows-5

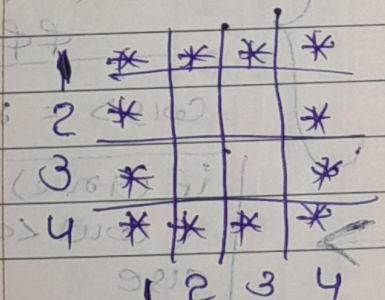
```
for (1 to 10) rows++
  for (1 to 10) cols++
    if (rows 1 to 5)
      if (conditions)
        cout << " ";
      else
        cout << "*";
    else if (rows 6 to 10)
      if (conditions)
        cout << " ";
      else
        cout << "*";
  cout << endl;
```

```
if (true)
  cout << " "
else
  cout << "*";
```

(20)

1	1	10
2	12	910
3	123	8910
4	1234	78910
5	12345	678910
6	1234	78910
7	123	8910
8	12	910
9	1	10

(21)



rows	filled cols.
1	1 2 3 4
2	1 4
3	1 4
4	1 2 3 4

(2 for loops) rows cols 1 to 4

```
if(rows>1 && rows<4)
    if (cols>1 && cols<4)
        { cout << " " ; }
    else
        { cout << "*" ; }
else
cout << "*";
```