Arrays

↓

a container/ basic DS, which stores same type of data together in continuous location.

```
data_type    name[size];    → declaration

int arr[ ]= {1, 2, 3, 4, 5}; ✓

int arr[5]= {1, 2, 3} ✓

int arr[5]= {2, 4, 9, 8, 0} ✓

int arr[3] ={4, 8, 1, 3, 5}  ✗  → can't initialize with more elements.
```

int arr[5] = {1, 2} → rest of the elements/indices are initialized with <u>0</u>.

int arr[n] = {0}; → all values to <u>0</u>.

→ If we don't initialize, garbage <u>value</u>.

cin >> arr[i]; ~~ta~~ taking input at index <u>i</u>.

cout << arr[i];

arr[i] = x;

arr[i] -= 5;

Search, finding unique elements, min, max, sum,
$\downarrow$ ind $\downarrow$ ind
average, pair to a sum,

reverse order print.

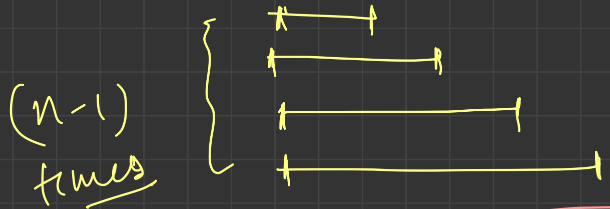
Sorting $\rightarrow$ Bubble Sort $\rightarrow$ (n-1) times {we compare every consecutive values, if $(a>b) \Rightarrow$ swap.}

$\rightarrow$ Selection Sort $\rightarrow$ (n-1) times) {we select the min. value and place it at the correct index.}

$\rightarrow$ Insertion Sort. $\rightarrow$ {we try to insert one element at a time to its correct position in the sorted part of the array.

Insertion

$\boxed{4}\ \overset{\downarrow}{2}\ \ \overset{\downarrow}{5}\ \ \overset{\downarrow}{1}\ \ \overset{\downarrow}{3}$



$\boxed{\,}$ ↳ one element

n=1 ⇒ already sorted ?!

insertion sort

$\boxed{\boxed{4}\ \ 2}\ \ \underbrace{5\ \ 1\ \ 3}$ → irrelevant at this step

↑ new element to be considered

i=1

arr[i] → insert 2 at its correct position.

sorted $\boxed{2\ \ 4}\ \ 5\ \ \{1\ \ 3\}$ → ignore.

arr[i]

i=2 ↑

→ insert 5 at its correct posⁿ in sorted part.

i=3 sorted $\boxed{2\ 4\ 5}$ | 3 → irrelevant.

→ insert 1 at its correct posⁿ

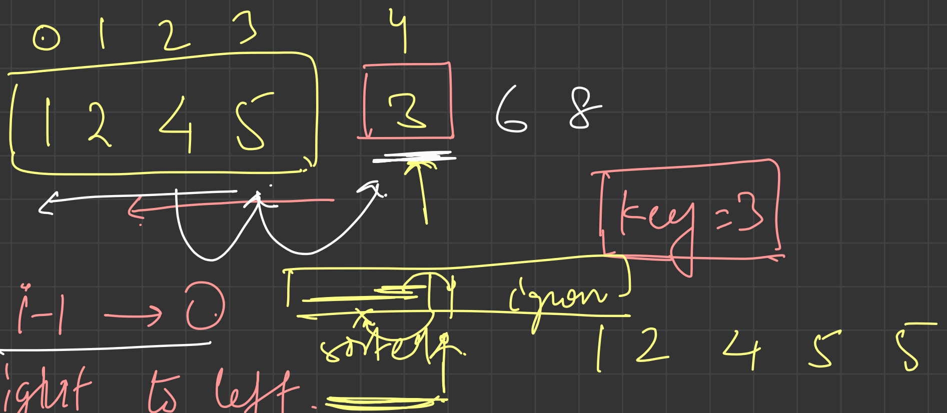
i=4 $\boxed{1\ 2\ 4\ 5}$ 3

→ insert 3 at its correct index.

$\boxed{1\ 2\ 3\ 4\ 5}$ ⇒ sorted.

{ from i=1 to n-1, simply insert values at their correct position in sorted part.

how to insert.



int key = arr[i];

int j = i-1;

if arr[j] > key

shift arr[j+1] = arr[j]

Same

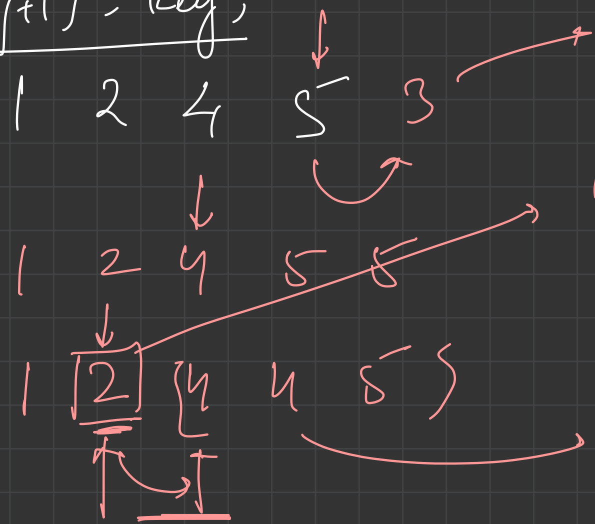
```
while (j>=0 && arr[j] > key){
    arr[j+1] = arr[j];

    j--;
}
↳ arr[j+1] = key;
```

else → stop

and finally

$\boxed{arr[j+1] = key}$



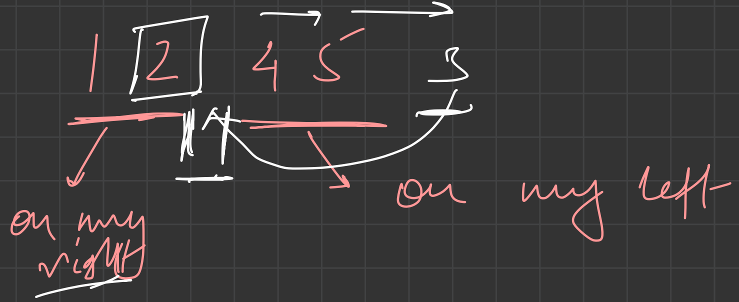
(key → 3)

key must be right to me

③

key must be left to me

③



$n-1=4$

4 1 2 5 3

1 →

1 4 2 5 3

1 2 4 5 3

1 2 4 5 3

1 2 4 3 5

2 →

1 2 4 3 5

1 2 4 3 5

1 2 3 4 5

1 2 3 4 5

3 4 ⇒ swaps ??

after some iteration if there are no swaps from left-to-right

⇒ array is already sorted.

⇒ we need not do anymore iterations

if (swaps == 0) ⇒ sorted, stop iteration

we can sort array in less than (n-1) iterations min index → correct index.

Pair to a sum

→ Given an array, a value X.
Find pairs in the array which sum upto X.

Q. → Given an array, a value, X.
Find triplets, which sum upto X.

① go over the array using i. $\Rightarrow$ arr[i]

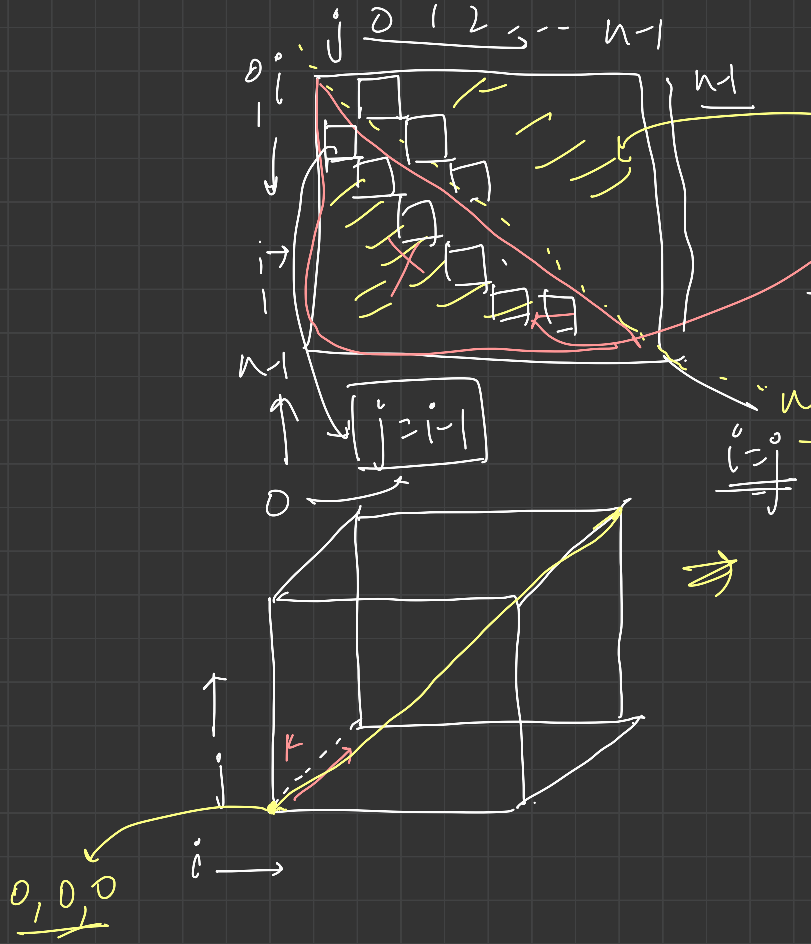
$\Downarrow$

② go over the array again. j $\Rightarrow$ arr[j]

$\Downarrow$

③ go over the array using k.

~~&~~ and find if there exists a value such that $\underline{arr[i] + arr[j] + arr}[k] = \underline{X}$;



$j \neq i$

$j = i+1 \rightarrow n-1$

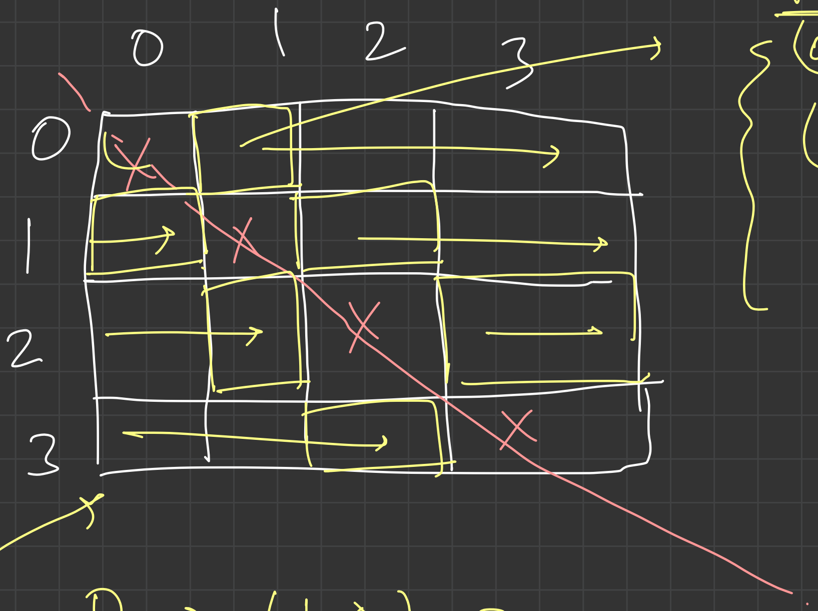
$j = 0 \leftarrow i-1$

$j = i+1$

$i = 0 \rightarrow n-1$	$i = 0 \rightarrow n-1$
$j = \underline{i+1} \rightarrow n-1$	$j = 0 - i-1$
$k = \underline{j+1} \rightarrow n-1$	$k = 0 - j-1$

avoid repetition

i j k   j i k



$i \quad j$

$$\left\{\begin{array}{c}(0,1)\\(1,2)\\(2,3)\\\cancel{(3,4)}\end{array}\right\} \Rightarrow \underline{j = i+1}$$

$<n$

$\boxed{n-1}$

$0 \to$

$$\left.\begin{array}{c}(1,0)\\(2,1)\\(3,2)\end{array}\right\} \Rightarrow \underline{j = i-1}$$

$\boxed{0 \to i-1}$

Ternary Operator:

```
if( i<2 ){
    arr[i]=0;
}
else {
    arr[i]=1;
}
```

```
( i<2 ) ? arr[i]=0 : arr[i]=1 ;
```

do-while:

```
do {
    - - - - body
}
while(___);
```

→ body executes at least once

```
while( cond^n ){        → false  ⇓ doesn't execute body
    body;
}
```

```
do {
    body;
} while(cond^n);        → false  → still at least one execution
```

<u>Functions</u>

↳ are a piece of code written together, which can be executed/invoked by any other part of the code using its name.

$$\begin{tikzcd} \text{p1} \arrow[r] & \text{alongwith their datatypes} \end{tikzcd}$$

```
return_type   func._name (p1, p2, --- ){

      body ⇒ whatever code is to be
             executed when the function
             is called/executed/invoked.
}
```

```
int sum (int a, int b){
    return a+b;
}
```

func. name

↓

→ variable naming rules follow

→ not any keyword.

main → int main( ){
special
        return 0;
    }

int
float
char
double
bool.

void → nothing

parameters : (data-type varname , datatype var _ _)

Scope / Visibility of variables

$$\begin{tikzcd}
 & \text{Scope / Visibility} \arrow[dl] \arrow[d] \arrow[dr, leftrightarrow] & \\
\text{main} \arrow[d, Rightarrow] & \text{function} \arrow[d, Rightarrow] & \text{global} \arrow[d, Rightarrow] \\
\text{only inside main} & \text{only inside that function} & \text{available everywhere}
\end{tikzcd}$$

main

×

main ( ) {

{ int a; }

} ×

Q. Write a function to calculate the factorial of a number.