Sorting

TABLE OF CONTENTS

Introduction to Sorting

Sorting is arrangement of data in particular order on the basis of some parameter.

1	2	3	5	7

Parameter: magnitude
Data is sorted in INC order.

9	6	3	2	1

Parameter: magnitude
Data is sorted in DEC order.

Quiz1:

1	13	9	6	12
1	2	3	4	6

Parameter: no of factors
Data is sorted in INC order.

no of factors of 1 : 1
" " " " 13 : {1, 13} = 2
" " " 9 : {1, 3, 9} = 3
" " " 6 : {1, 2, 3, 6} = 4
" " " 12 : {1, 2, 3, 4, 6, 12} = 6

Why Sorting is Required?

Sorting is essential for organizing, analyzing, searching and presenting data effectively and efficiently in various applications & contexts.

Problem 1: Minimise the Cost to Empty Array

Given N elements, at every step remove an array element.

Cost to remove an element = Sum of array of elements present in an array

Find minimum cost to remove all elements.

NOTE : First add the cost of removal and then remove it.

0	1	2
2	1	4

ans : 11

2	1	4
	1	4
		4

2+1+4 = 7
1+4 = 5
4
7+5+4 = 16

2	4	1
	4	1
		1

2+4+1 = 7
4+1 = 5
1
7+5+1 = 13

1	2	4
	2	4
		4

1+2+4 = 7
2+4 = 6
4
7+6+4 = 17

1	4	2
	4	2
		2

1+4+2 = 7
4+2 = 6
2
7+6+2 = 15

4	1	2
	1	2
		2

4+1+2 = 7
1+2 = 3
2
7+3+2 = 12

4	2	1
	2	1
		1

4+2+1 = 7
2+1 = 3
1
7+3+1 = 11

Quiz 2:

0	1	2
4	6	1
	4	1
		1

$4+6+1 = 11$

$4+1 = 5$

1

$11+5+1 = 17$

Quiz 3:

0	1	2	3
3	5	1	-3
	3	1	-3
		1	-3
			-3

$3+5+1+(-3) = 6$

$3+1+(-3) = 1$

$1+(-3) = -2$

-3

$6+1-2-3 = 2$

	0	1	2	3
arr[4] =	a	b	c	d

Remove a : a + b + c + d
Remove b : b + c + d
Remove c : c + d
Remove d : d

a + 2b + 3c + 4d

↓

Minimise this cost

Observation:

1) Element occuring most : d (4 times)
Element occuring least : a (1 time)

To minimise cost, minimum element should have highest frequency of occurence and the maximum element should have least frequency of occurence.

Largest element : a
Smallest " : d

Order of elements : a > b > c > d

2) i^{th} index element is occuring (i+1) times.
Contribution of i^{th} element = (i+1) * a[i]

Approach :

1. Sort the array in descending order.
2. Iterate on array & add contribution for each element.

Code :

```
int minCost(int arr[], int n) {

    // Sort the array in descending order
    reverse_sort(arr);

    // Add contribution
    int ans = 0;
    for(i=0; i<n; i++) {
        ans += ((i+1) * arr[i]);
    }

    return ans;
}
```

T.C : $(n \log n + n) = (n \log n)$

S.C : $O(1)$

Problem 2: Count of Noble Integers (Distinct Elements)

Given an array of distinct elements, calculate number of noble integers.

An element ele in arr [ ] is said to be noble if { count of smaller elements = ele itself }

	0	1	2	3	4	5
	1	-5	3	5	-10	4
# ele smaller	2	1	3	5	0	4

ans : 3

Quiz 4 :

	0	1	2	3
	-3	0	2	5
# smaller elements	0	1	2	3

ans : 1

Brute force solution :

For each element, iterate & find no of smaller elements.

If count of smaller elements == arr[i]
then I can say element is noble

```
int nobleCount( int arr[], int n ){

    int ans = 0;
    for( int i = 0 ; i < n ; i++ ){

        int count = 0;
        // count smaller elements
        for( int j = 0 ; j < n ; j++ ){
            if( arr[j] < arr[i] )
                count++;
        }

        if( count == arr[i] )
            ans++;
    }
    return ans;
}
```

T.C : $O(N^2)$
S.C : $O(1)$

The extra work is done in 2nd loop.

We need to find technique to find count of smaller elements efficiently.

Observations :

0	1	2	3	4	5
1	-5	3	5	-10	4

↓ Sorting

0	1	2	3	4	5
-10	-5	1	3	4	5

In sorted array,
see smaller elements for element "i" will be [0, i-1]

Index gives me count of smaller elements.

If index == arr[i] → noble element.

Optimised Solution:

Approach:

1. Sort the array
2. If Index == arr[i] then it is noble element.

Code:

```
int nobleIntegers(int arr[], int n) {

    // Sort the array
    sort(arr);
    int ans = 0;
    // Count smaller elements
    for (i = 0; i < n; i++) {
        if (arr[i] == i)
            ans++;
    }

    return ans;
}
```

T.C: (n log n + n)
= n log n

S.C: O(1)

Problem 3: Count of Noble Integers (Not Distinct)

Same as previous question, but all elements need not be distinct.

	0	1	2	3	4	5
	0	2	2	3	3	6
# Smaller elements	0	1	1	3	3	5

ans: 3

Quiz 5 :

	0	1	2	3	4
	-10	1	1	3	100
# less ele	0	1	1	3	4

ans: 3

Quiz 6 :

	0	1	2	3	4	5	6	7	8
	-10	1	1	2	4	4	4	8	10
# less ele	0	1	1	3	4	4	4	7	8

ans: 5

Observations:

In sorted array,

Case I: If curr element != prev element
no of smaller elements will be given by index.

Case II: If curr element == prev element
no of smaller elements will be same as no of smaller element for prev element.

Code :

```
int countnoble ( int arr[ ], int n) {

    // Sort the array
     sort (arr);
      int ans =0;
    // Check for smaller elements
     int count =0;
     if (arr[0] == = 0)   ans ++;
    for (i= 1 ;  i <n ; i++) {
            // not a duplicate element
        if ( arr[i] ! = arr[i-1])
            count = i ;
        // Check for noble element
        if (count = = arr[i])
            ans ++;

    }

    return ans;

}
```

T.C : O(n log n + n)
= O(n log n)
S.C : O(1)

	0	1	2	3	4
	-10	1	1	3	100
count : 0	0	1	1	3	4
ans : 0	∅	1	2	3	3

ans : 3

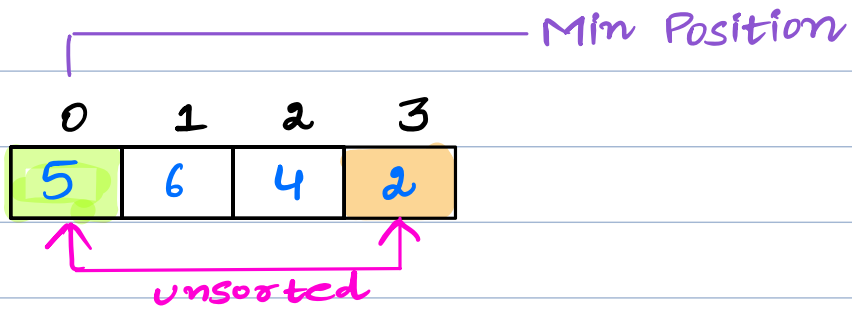
Selection Sort

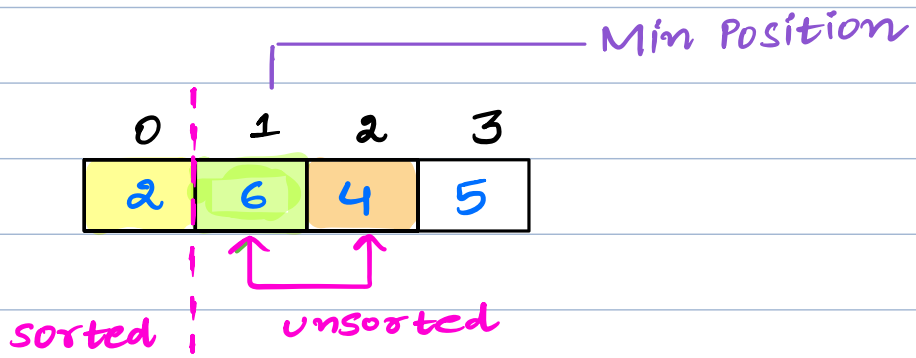
Break till 9:00

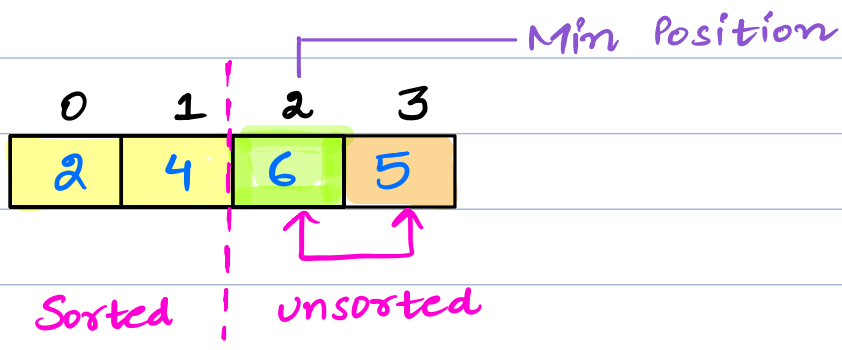
Arrange students in height wise order.

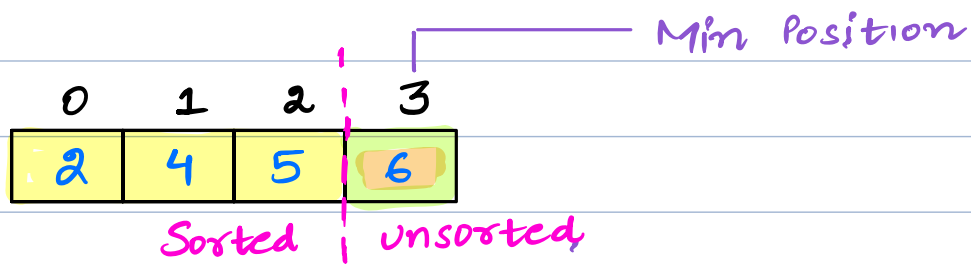
Selection sort works by repeatedly selecting the smallest element from the unsorted list and moving it to the sorted list.

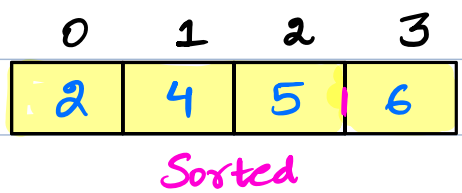
Idea : Repeatedly selecting the smallest element from unsorted list & then swapping it with first element of unsorted list.









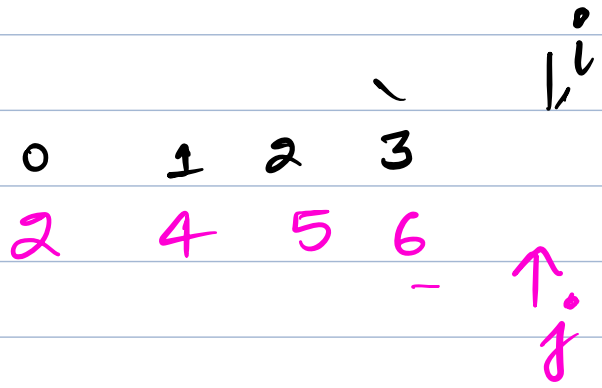


Code :

```
void SelectionSort(int arr[], int n){

    int minindex;

    for(i=0; i<n; i++){
        // set minindex to 1st element of unsorted part
        minindex = i;
        // Find min element
        for(j=i+1; j<n; j++){
            if(arr[j] < arr[minindex]
                minindex = j;
        }

        // Swap min element with element
           at min position

        Swap(arr[i], arr[minindex]);
    }
}
```

T.C : $O(N^2)$
S.C : $O(1)$

Dry Run :

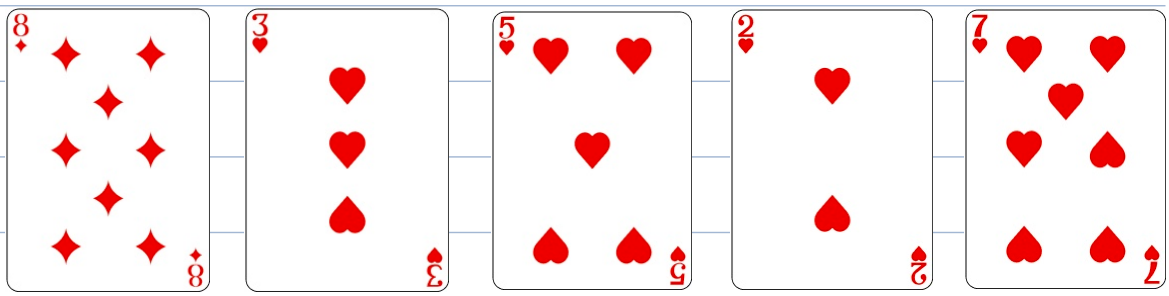


minindex = 0 ~~2~~ ~~3~~ ~~1~~ ~~2~~ ~~2~~ 3

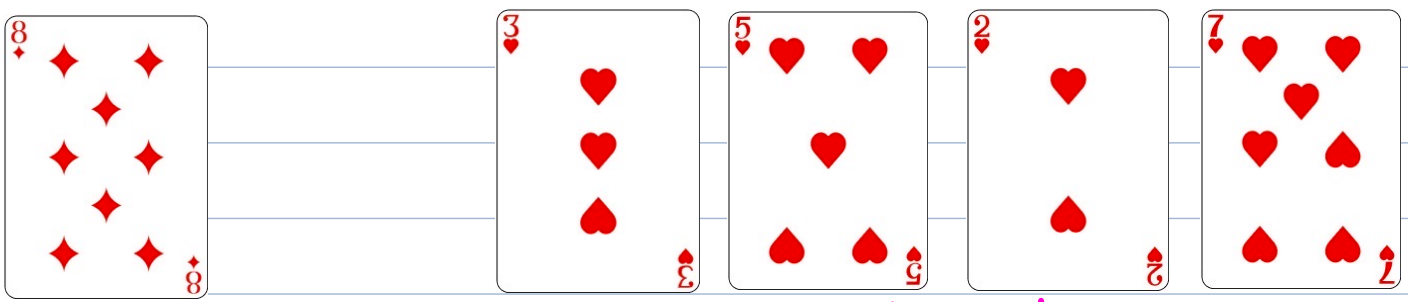
Insertion Sort

Rearranging deck of cards

Pick a card from unsorted group & put it in its correct position in sorted group.

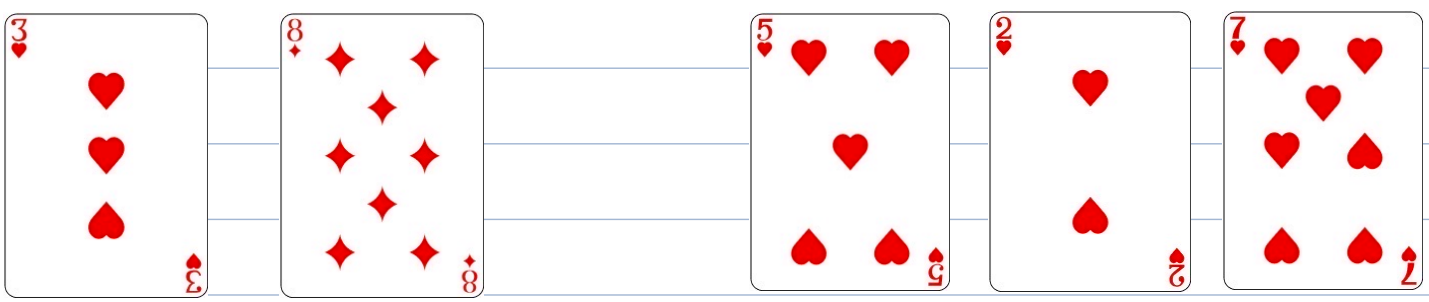


Unsorted



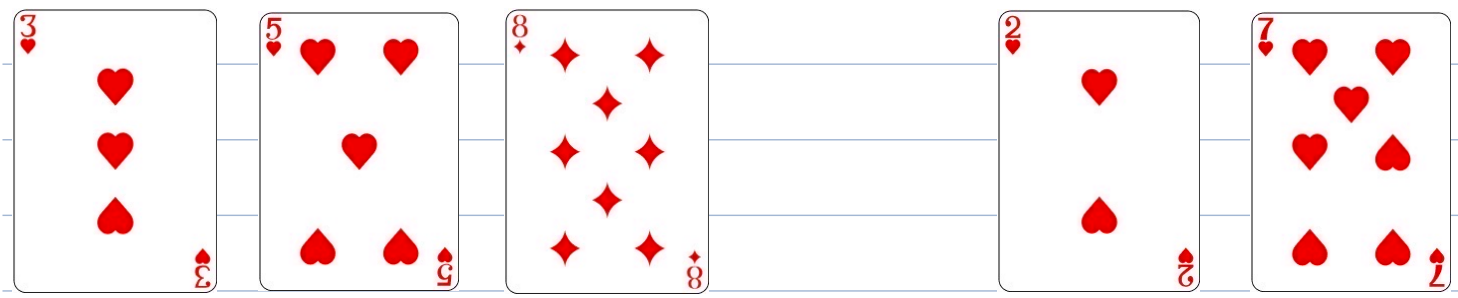
Sorted

Unsorted



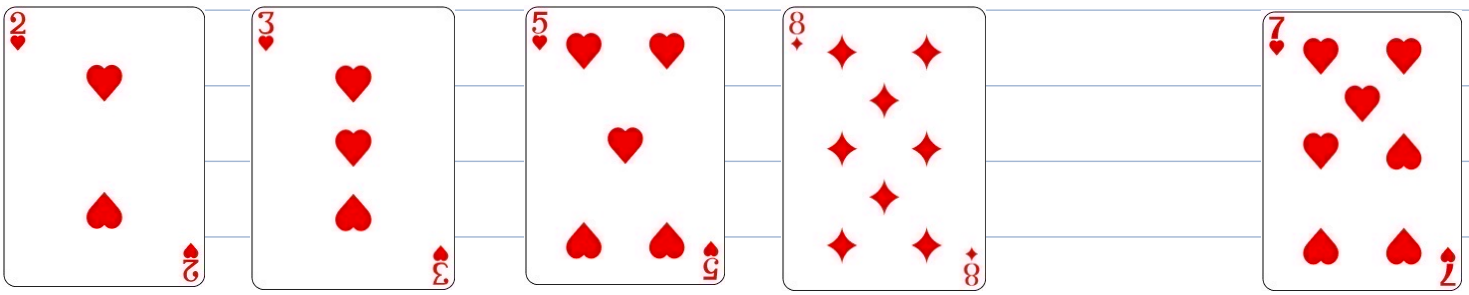
Sorted

Unsorted



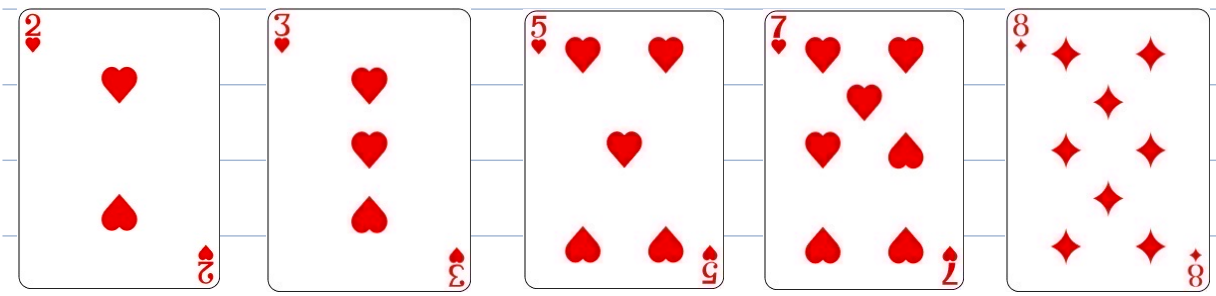
Sorted

Unsorted



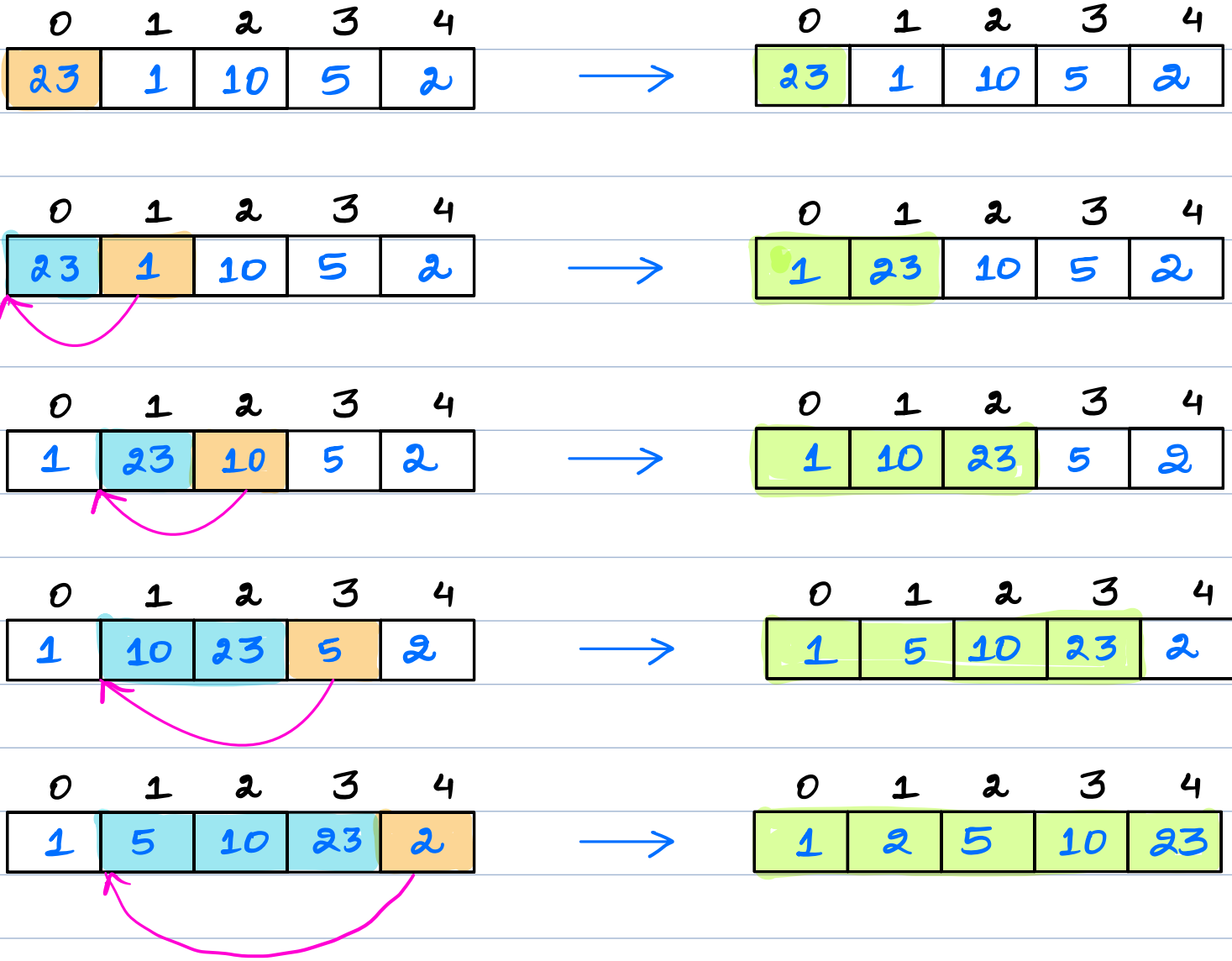
Sorted

Unsorted



Sorted

Idea : Iteratively picking an element from unsorting part & putting it into its correct position in sorted part.



Code :

```
void insertionSort(int arr[], int n){
    for(int i=1; i<n; i++){
        int key = arr[i];

        int j = i-1;

        // move elements of arr[0...i-1] that are
        greater than key to one position right.
        while(j>=0 && arr[j] > key){
            arr[j+1] = arr[j];
            j--;
        }

        // Put element at correct position
        arr[j+1] = key;

    }
}
```

worst case : T.C : $O(N^2)$

best case : T.C : $O(N)$

S.C : $O(1)$

Dry Run :

i

0	1	2	3	4
1	2	5	10	23

key = arr[i] = ~~10~~ ~~5~~ 2

Next Class

Bit manipulation

Bit manipulation is crucial in the world of computer science and programming! It's all about working directly with individual bits within binary data representations. Let's dive into why bit manipulation matters:

Memory and Space Efficiency: Computers use binary (0s and 1s) to store data. Tweaking these bits can lead to smaller data representations, saving precious memory and storage space. This is especially vital in resource-limited environments like mobile devices and embedded systems.

Performance Boost: Bit manipulation speeds up algorithms. Many hardware-level operations rely on bits, and using these directly in your code optimizes execution. This means faster results without costly higher-level processes.

Data Compression and Encryption: Tricks with bits are essential in data compression and encryption. Algorithms cleverly manipulate bits to represent data efficiently. Think of Huffman coding, where frequently used characters get shorter bit representations.

Masking and Flags: You can use individual bits for flags and settings. By toying with these bits, you can set, clear, toggle, or check settings without complex conditions. Simple yet powerful!

Logic and Algorithm Design: Fancy algorithms often rely on bit manipulations. Think `AND, OR, XOR, and NOT` operations directly on bits. This leads to slick and efficient solutions in various scenarios.

Puzzles and Challenges: Bit manipulation is a gem in programming puzzles and competitions. Mastering this skill helps you breeze through challenges like a champ!

To wrap it up, bit manipulation offers a nifty, efficient way to play with binary data. It shines in programming, optimization, data magic, and hardware interactions. While not needed in every task, understanding bit manipulation turbocharges your problem-solving and coding skills.