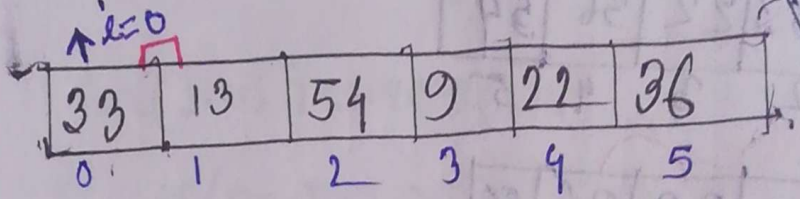
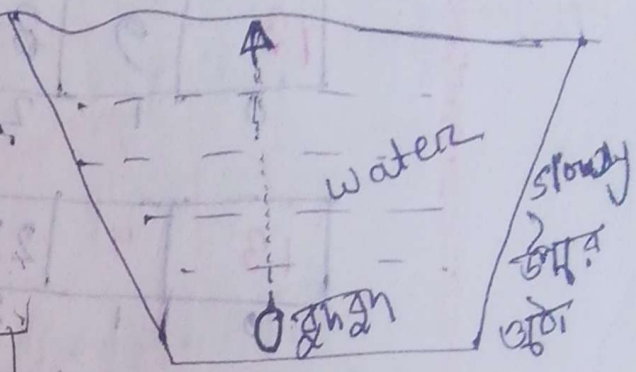
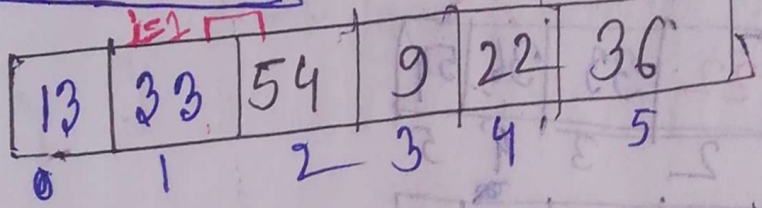
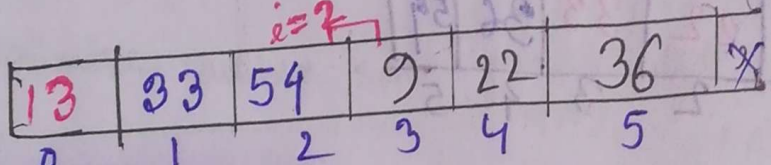
Bubble Sort

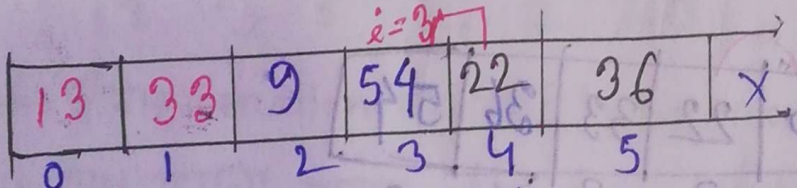


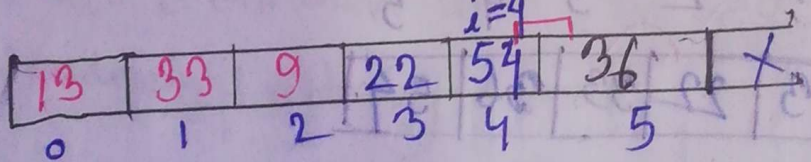


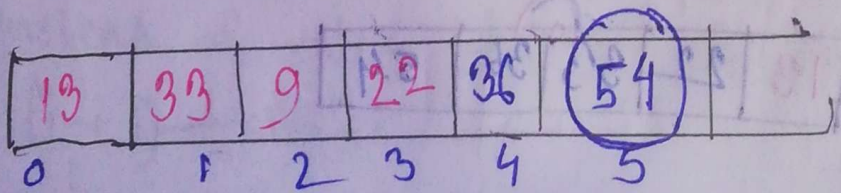
1st phase / 1st iteration











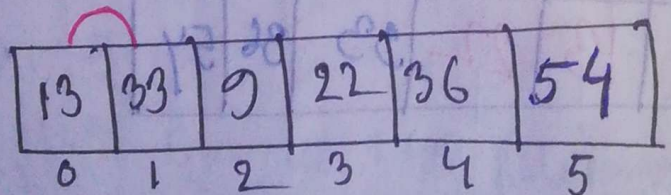
n = 6

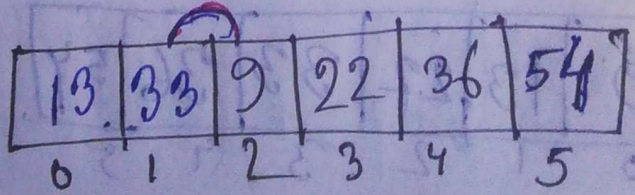
i = 0 → n-2

arr[i] ↔ arr[i+1]

1st phase में max value fixed हो जाती है

2nd phase

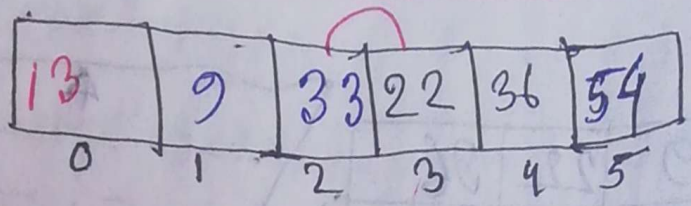


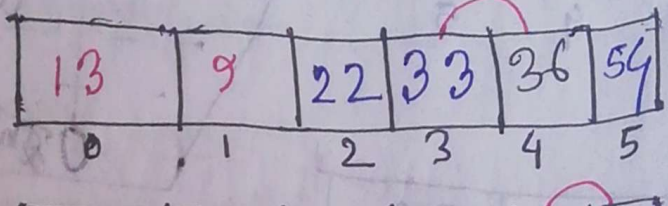


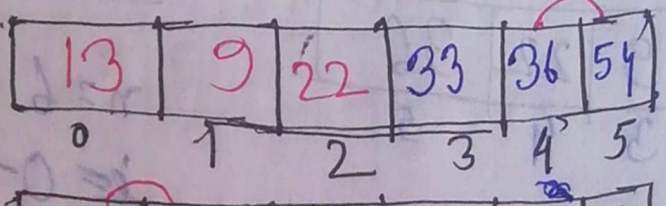
When not to use

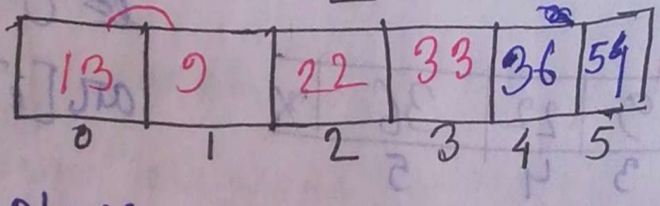
1) Dealing with a large set of data
2) When you are looking for a quick algorithm
3) Bubble sort is slow

140 156

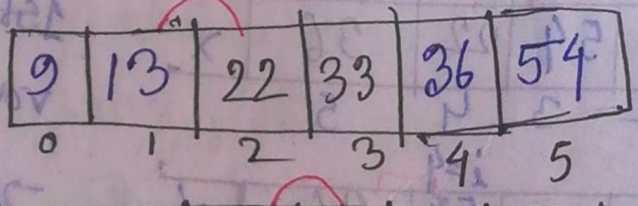


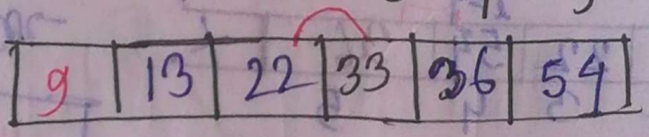


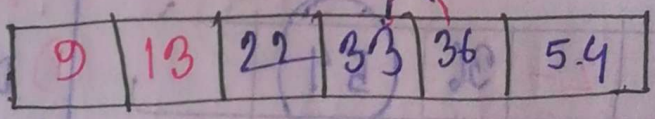




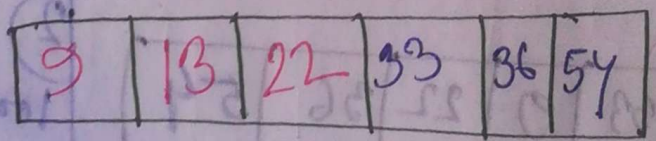
3rd phase



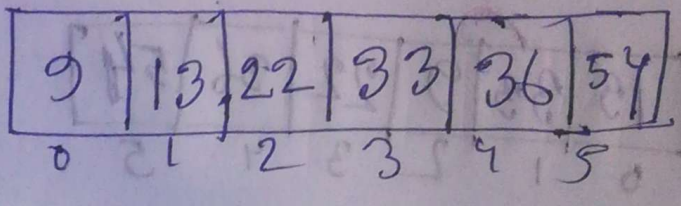




4th phase



5th phase



```
for(i=1; i<size; i++)          → iteration
{
  for(j=0; j<size-1; j++)      → Bubble up
  {
     if(Arr[j] > Arr[j+1])     → Ascending
     {
        swap(Arr[j], Arr[j+1]) / temp = A[j]
                                 A[j] = A[j+1]
                                 A[j+1] = temp
     }
  }
}
```

Descending

```
if(Arr[j] < Arr[j+1])
{
}
```

Optimize Bubble sort ! কম সংখ্যক test করে যেনো sort করতে পারি।

14 3 5 2 1 20

total 5 বার iteration করা লাগবে। for example 3rd iteration এই sort হয়ে গেলো। তাহলে বাকি 2 বার sort করার দরকার নাই। (Redundant testing)

1) Instead of value of j = size-2 (j=0; j<size-1; j++) use (j-i-1) → size-i

4 iteration →

1 5 4 4 14
1 5 4 4 14
1 4 5 4 14
1 4 5 4 14
1 4 5 4 14

already sorted don't need to compare again

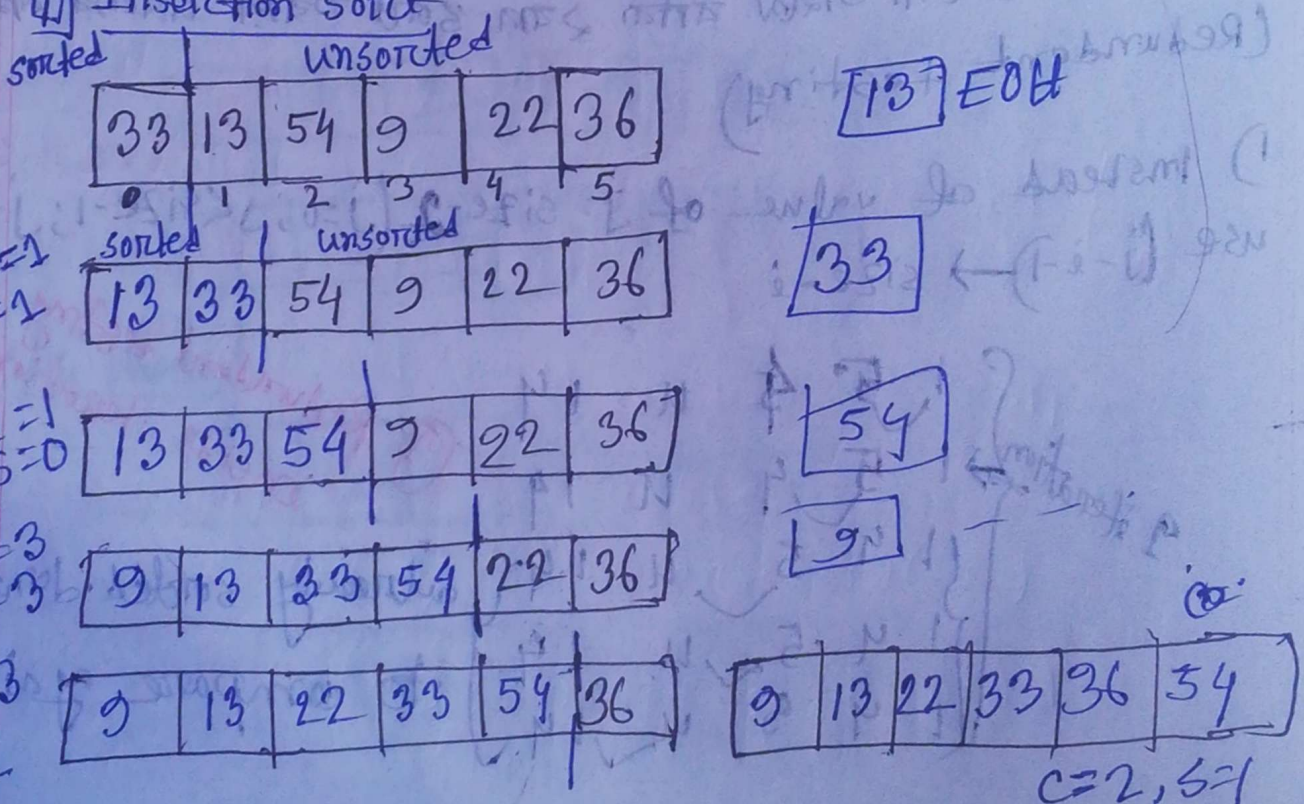
② Redundand testing inside iteration

2.

```
for (i=1; i<n; i++)
{
    int flog=0
    for(j=0; j<size-i; j++)
    {
        if(A[j] > A[j+1])
        {
            swap(A[j], A[j+1]); flog=1
        }
    }
    printArray(A, n)
    if(flog==0)
        break;
}
```

② Redundant iteration

Insertion Sort



```
for(i=1; i<n; i++)  → i = 1 to n-1 traversal
{
    key = arr[i]
    // j = i-1 to j=0
    j = i-1
while (j>=0 && key < arr[j])
    {
        arr[j+1] = arr[j]; j--;
    }
    arr[j+1] = key
}
```

key ↑

33	13	54	9	22	36
0	1	2	3	4	5

i হবে,

j = i-1 = 0, key = 13

33 > 13 && j = 0

arr[1] = 33

j--

arr[0] = 13

Selection Sort

Selection sort: selection sort কাজ করার আগে select করে। যে minimum তাকে select করবে।

33	13	54	9	22	36

i=0

9	13	54	33	22	36

index 0 sorted

i=1

9	13	54	33	22	36

index 0,1 sorted

i=3

9	13	22	33	54	36

index 0,1,2 sorted

i=4

9	13	22	33	54	36

i=5

9	13	22	33	36	54

0	1	2	3	4	5
33	13	54	9	22	36

```
for(i=0; i<n-1; i++)
{
    int min=i
    for(j=i+1; j<n; j++)
    {
        if (A[j] < A[min])
        {  min=j
        }
    }
    if (min!=i)
    {
        swap (A[i], A[min]
    }
}
```

i=0, min=0
j=1 2 3
A[j] < A[i]
13 < 33, 9<13
min = 1, 3

* Bubble sort এ ২টি for loop use হয়।
* Insertion sort এ for loop এর ভিতরে while / do while loop use হয়।
* selection sort এ ২টি for loop use হয়।

Insertion	Bubble
Best case: $O(n)$	Best case: $O(n)$
worst " : $O(n^2)$	worst case: $O(n^2)$ → Reverse

→ Not comparison sort

Array Consolidation

Counting sort

maximum

arr → 1 2 5 9 2 6 5 4

Step 1: Find maximum num from unsorted array

Step 2: Make new array. size will be max+1

count [max+1]

count

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

Step 3: Find frequency of each element. Then press these element into count array.

1=1
2=2
3=0
5=2

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

Step 4: Find cumalitive sum [Prefix sum] করবো এখন

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

c[i] += c[i-1]

Step 5: Do actual sorting. Make a new final array. size will same as main array.

Scan the main array from back [backtracking]. 4 found. Go to the count array in the 4th index. Decrement the value of 4th index, put the value of main array (4) in the decrement index (3.)

Final

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

Q. Why backtracking instead of frontracking?

ans: concept: FIFO

Pseudo code

S-1 for(0 → n-1) → O(n) (main array)
{
// find max
}

S-2
int count[max+1]
for( ) → O(n)
{
// initialize with 0
}

S-3: for(0 → n-1) → O(n)
{ arr[i]
count[arr[i]]++;

S-4 for(1 → Max) → O(n)
{ count[i] += count[i-1]

S-5 for(n-1 → 0) → O(n) → array
count[arr[i]]--
k = count[arr[i]]
Final[k] = arr[i]

Time complexity

O(n)