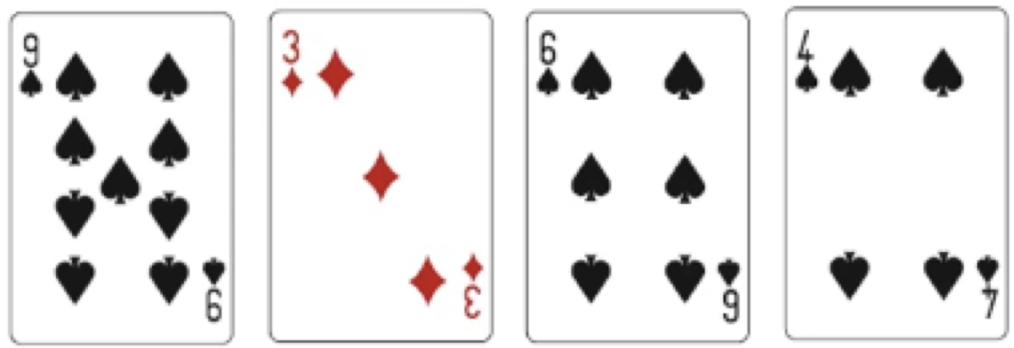
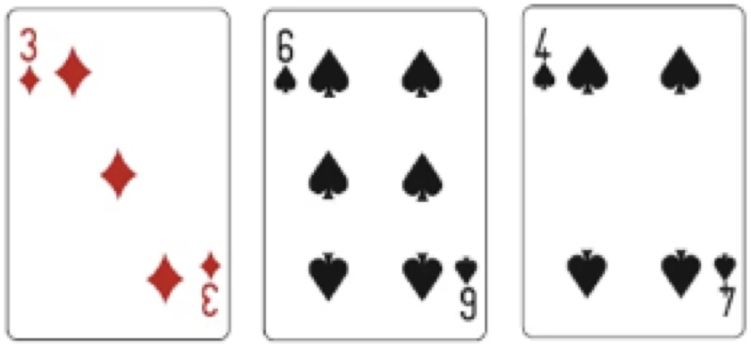
Insertion and Shell Sort

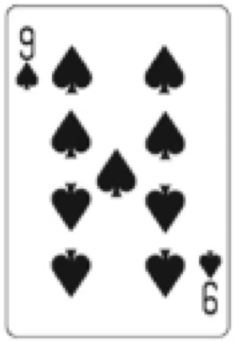
IIITS



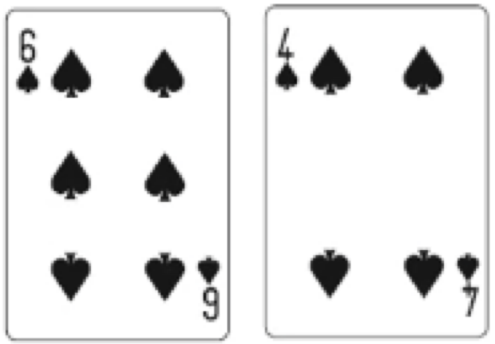
Left



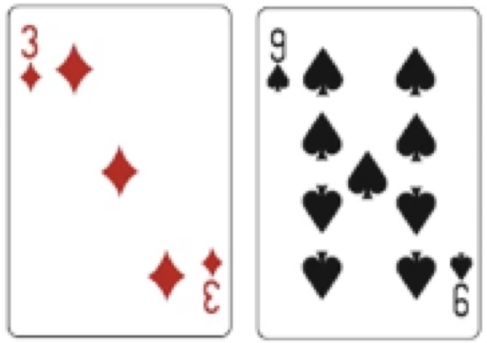
Right



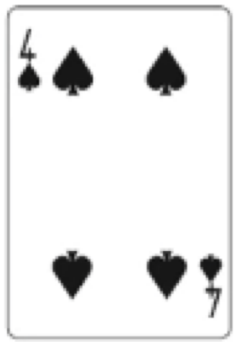
Left



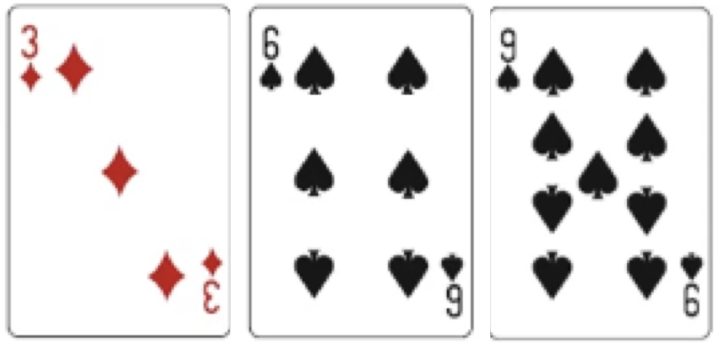
Right



Left

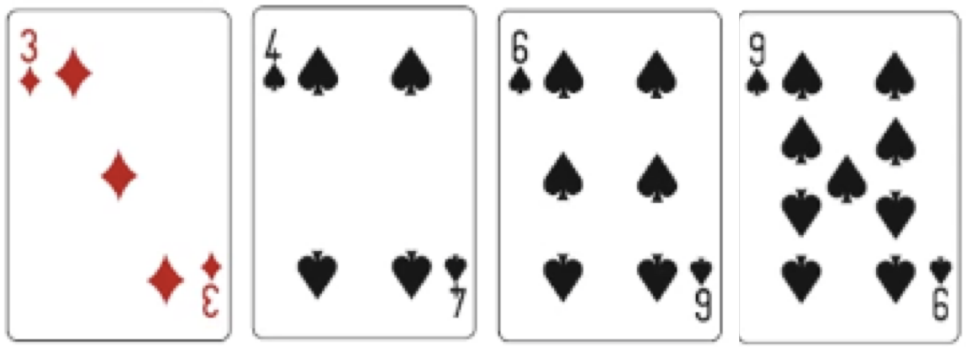


Right



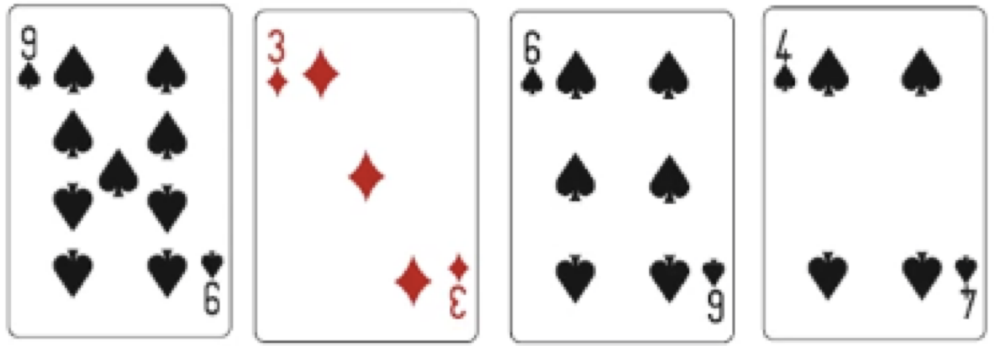
Left

Right



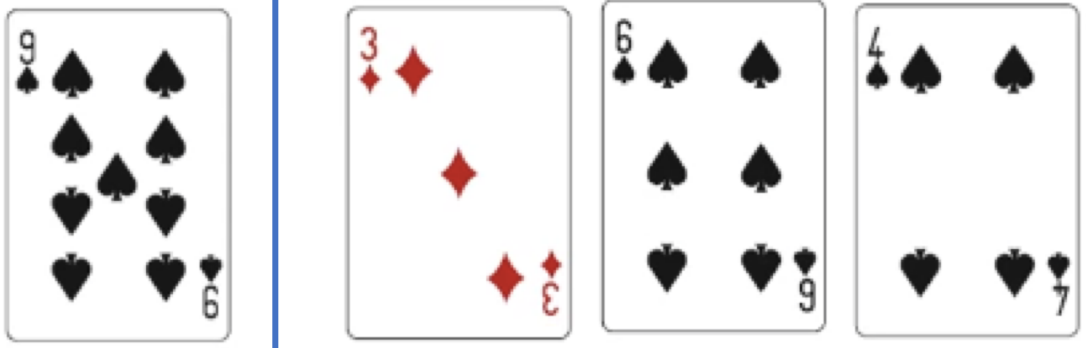
Insertion Sort – Pseudo Algorithm

Sorted Unsorted



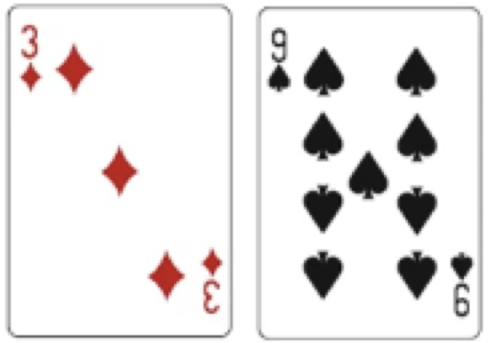
Insertion Sort – Pseudo Algorithm

Sorted Unsorted

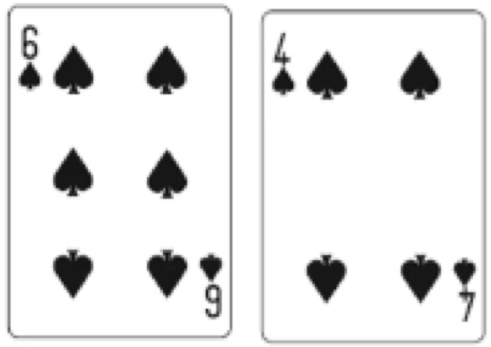


Insertion Sort – Pseudo Algorithm

Sorted

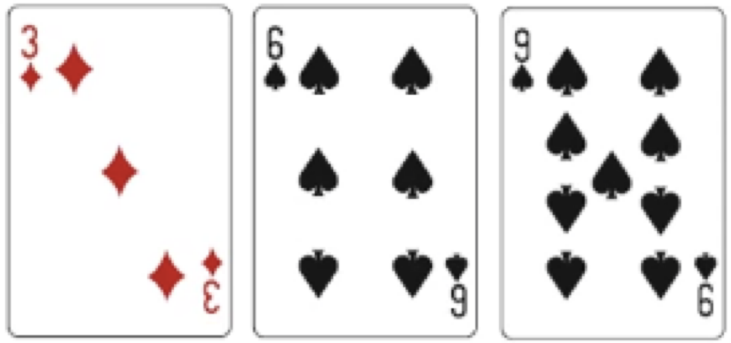


Unsorted

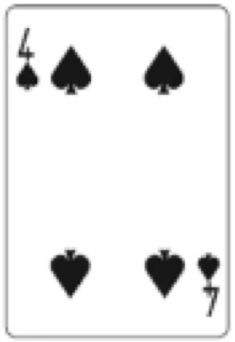


Insertion Sort – Pseudo Algorithm

Sorted



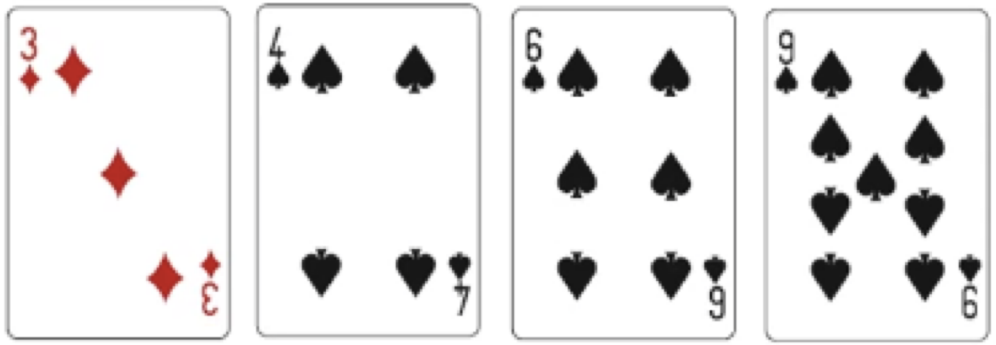
Unsorted



Insertion Sort – Pseudo Algorithm

Sorted

Unsorted



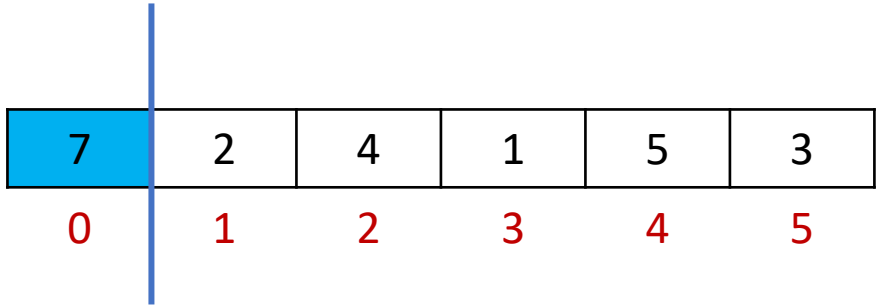
Insertion Sort – Pseudo Algorithm

7	2	4	1	5	3

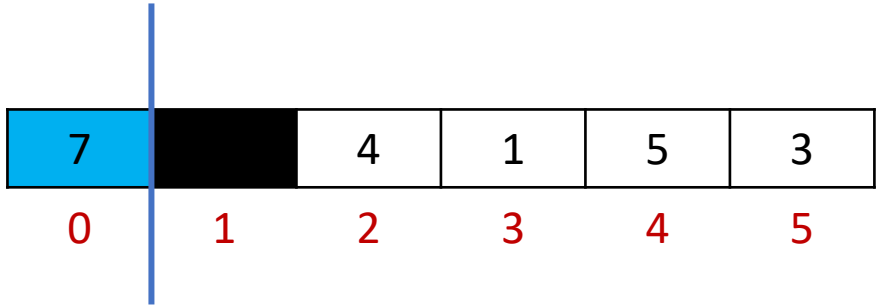
Insertion Sort – Pseudo Algorithm

7	2	4	1	5	3
0	1	2	3	4	5

Insertion Sort – Pseudo Algorithm

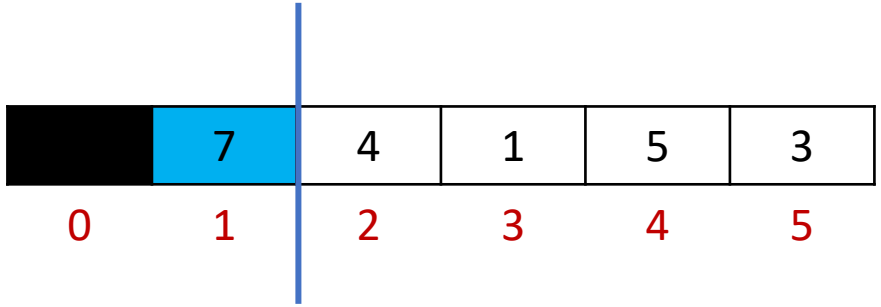


Insertion Sort – Pseudo Algorithm



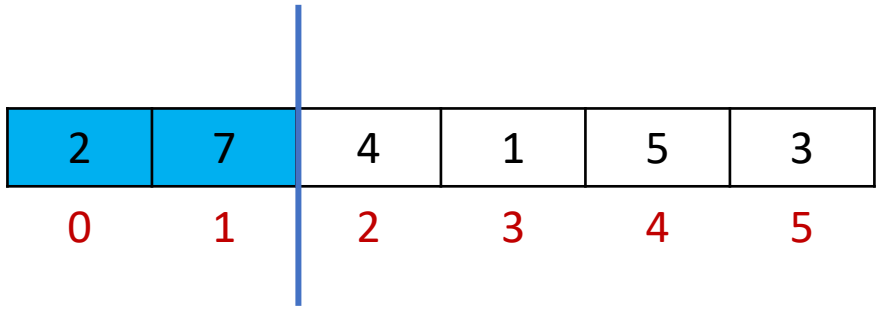
Value =2

Insertion Sort – Pseudo Algorithm



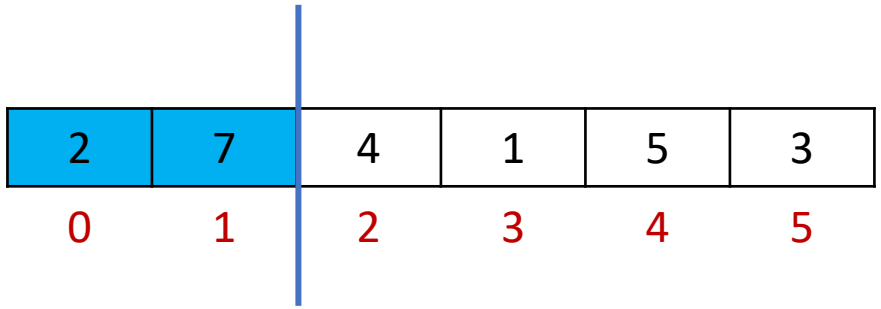
Value =2

Insertion Sort – Pseudo Algorithm



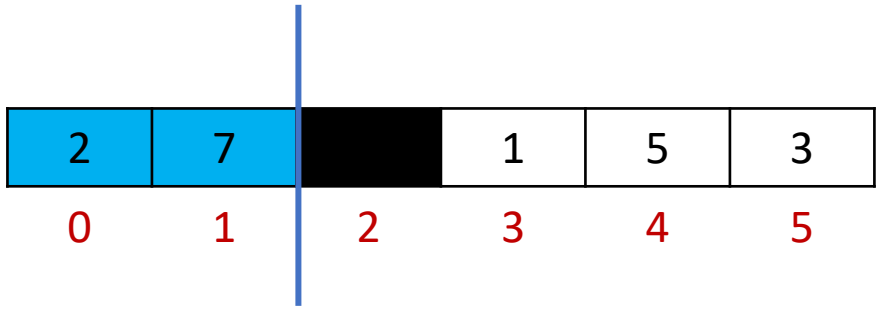
Value =null

Insertion Sort – Pseudo Algorithm



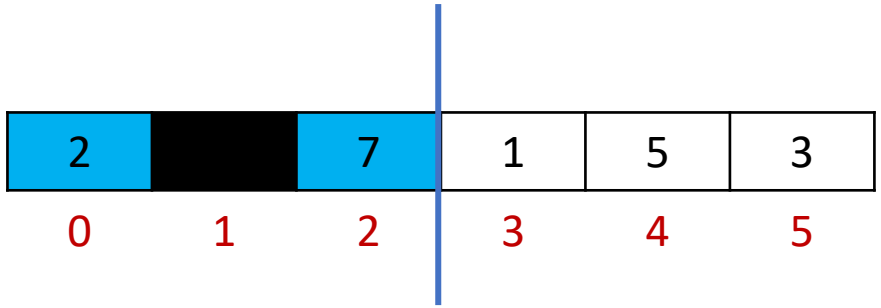
Value = 4

Insertion Sort – Pseudo Algorithm



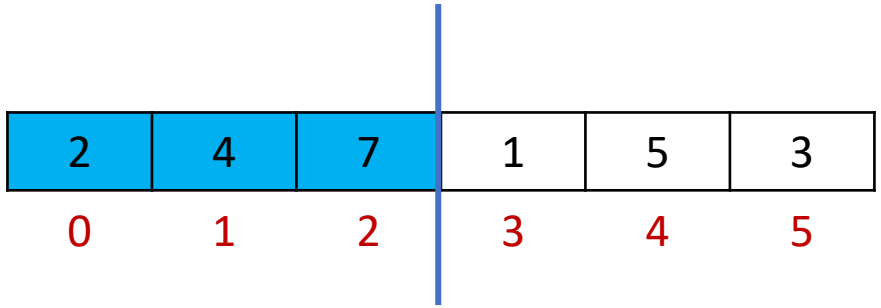
Value = 4

Insertion Sort – Pseudo Algorithm



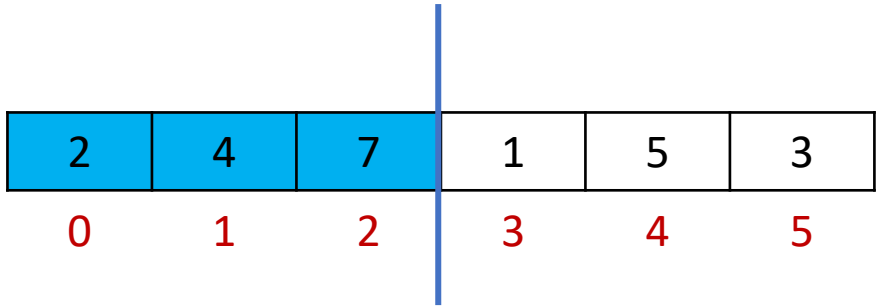
Value = 4

Insertion Sort – Pseudo Algorithm



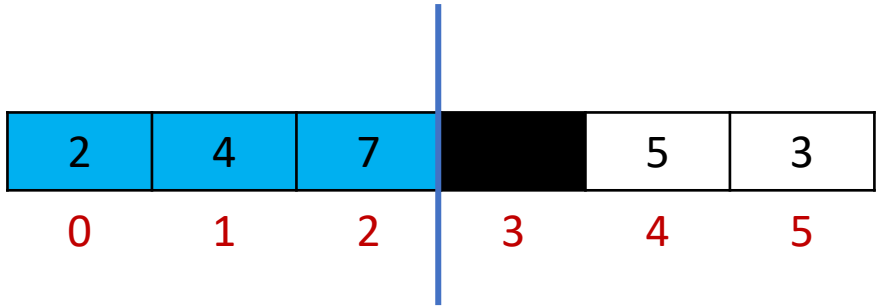
Value = null

Insertion Sort – Pseudo Algorithm



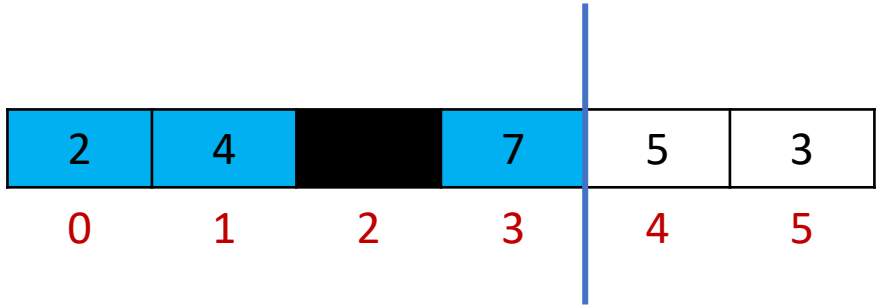
Value = 1

Insertion Sort – Pseudo Algorithm



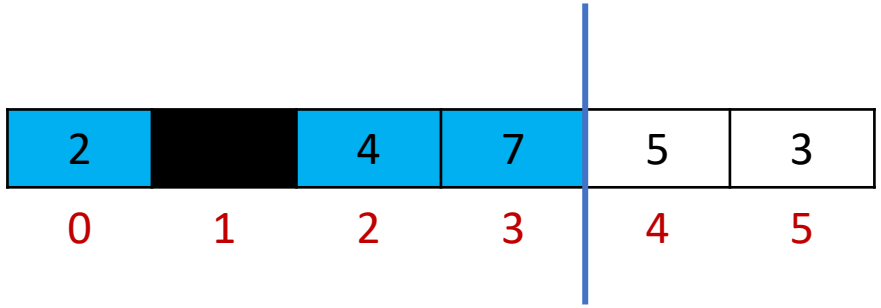
Value = 1

Insertion Sort – Pseudo Algorithm



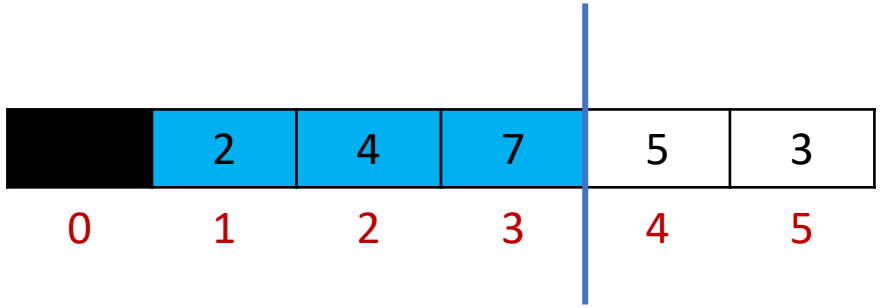
Value = 1

Insertion Sort – Pseudo Algorithm



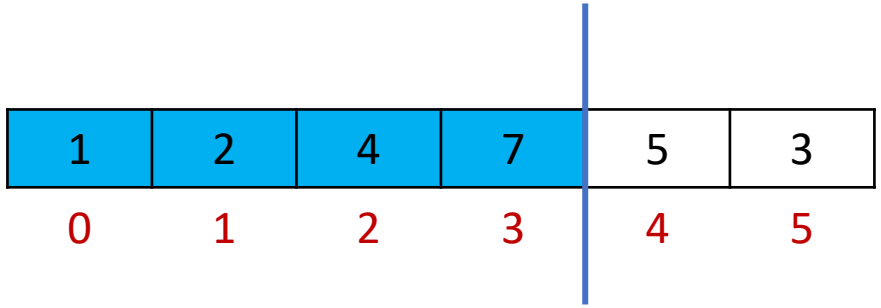
Value = 1

Insertion Sort – Pseudo Algorithm



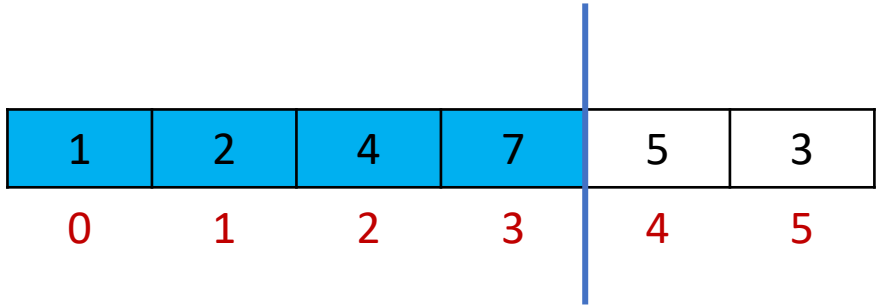
Value = 1

Insertion Sort – Pseudo Algorithm



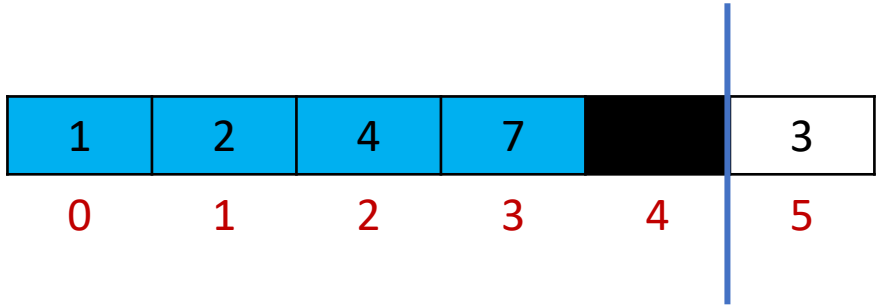
Value = null

Insertion Sort – Pseudo Algorithm



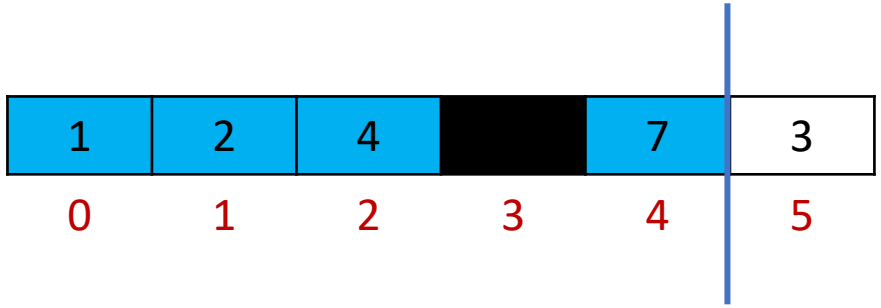
Value = 5

Insertion Sort – Pseudo Algorithm



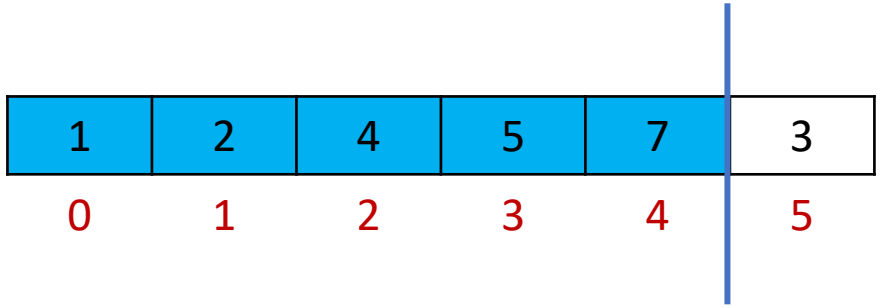
Value = 5

Insertion Sort – Pseudo Algorithm



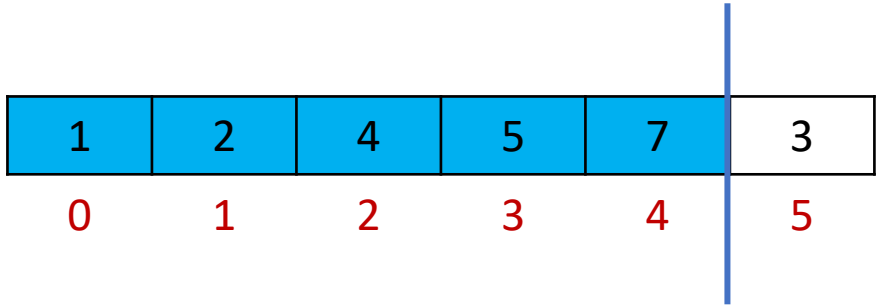
Value = 5

Insertion Sort – Pseudo Algorithm



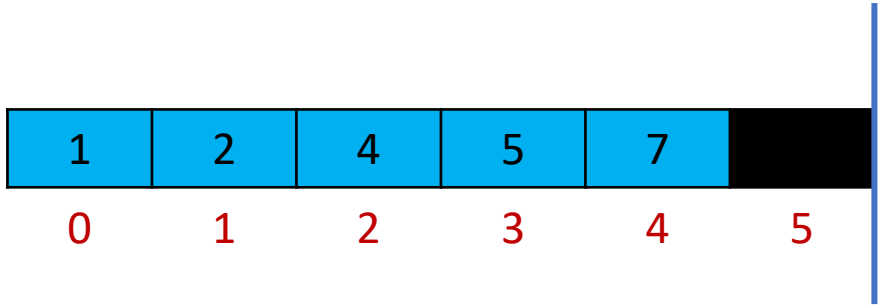
Value = null

Insertion Sort – Pseudo Algorithm



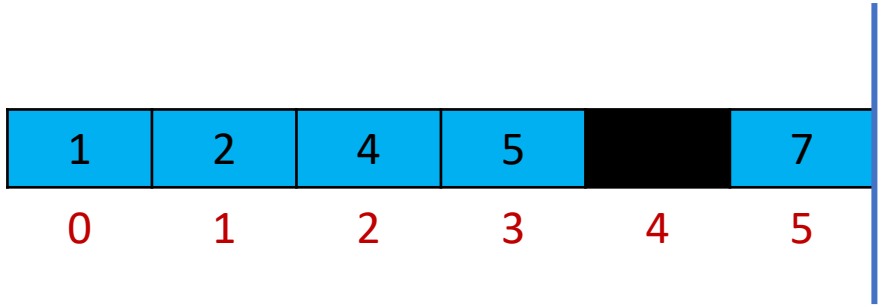
Value = 3

Insertion Sort – Pseudo Algorithm



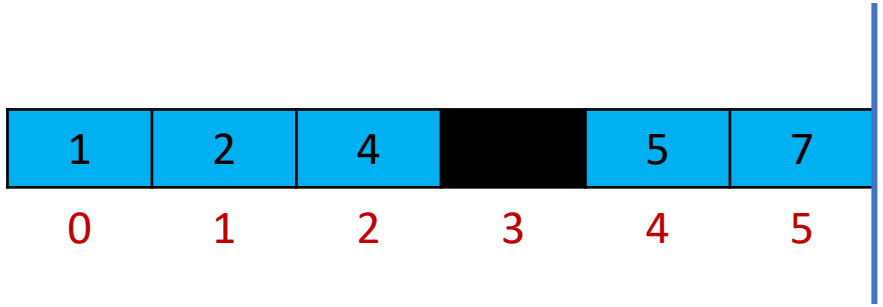
Value = 3

Insertion Sort – Pseudo Algorithm



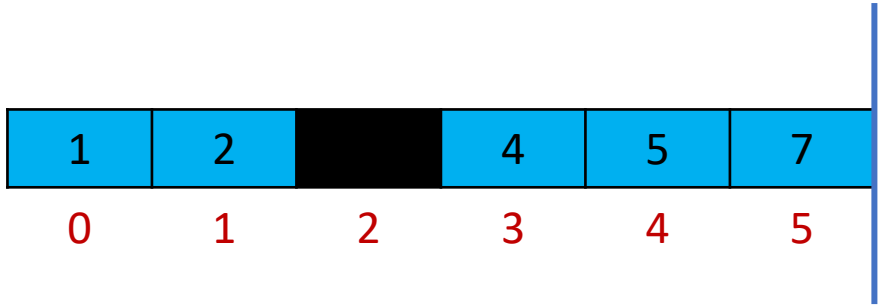
Value = 3

Insertion Sort – Pseudo Algorithm



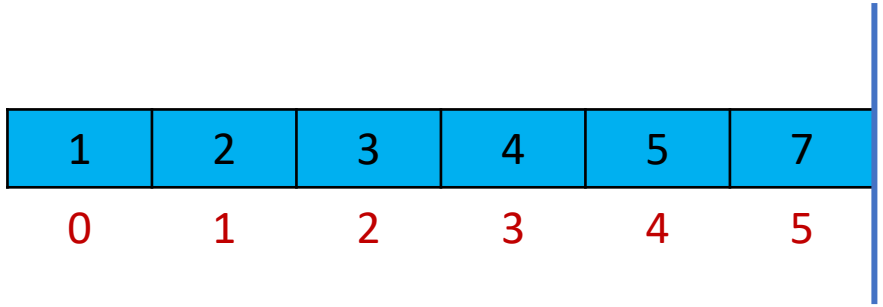
Value = 3

Insertion Sort – Pseudo Algorithm



Value = 3

Insertion Sort – Pseudo Algorithm



Value = null

```c
/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i-1;

        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j+1] = arr[j];
            j = j-1;
        }
        arr[j+1] = key;
    }
}
```

Insertion Sort - complexity

```
/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
C1      key = arr[i];
        j = i-1;

        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
        while (j >= 0 && arr[j] > key)
        {
C2          arr[j+1] = arr[j];
            j = j-1;
        }
C3      arr[j+1] = key;
    }
}
```

Insertion Sort - complexity

Best Case

1, 2, 3, 4, 5, 6 Sorted array

T (n) = (C1 + C3) (n-1)
== an +b
== O (n)

```
/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
C1      key = arr[i];
        j = i-1;

        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
        while (j >= 0 && arr[j] > key)
        {
C2          arr[j+1] = arr[j];
            j = j-1;
        }
C3      arr[j+1] = key;
    }
}
```

Insertion Sort - complexity

Worst Case

6, 5, 4, 3, 2, 1

Reversed sorted array

$T(n) = (C1 + C3)(n-1) +$
$\{1 + 2 + \ldots + (n-1)\} C2$
$== (C1 + C3)(n-1)$
$\{n(n-1)/2\} C2$
$== an^2 + bn + c$
$== O(n^2)$

```
/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
C1      key = arr[i];
        j = i-1;

        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
        while (j >= 0 && arr[j] > key)
        {
C2          arr[j+1] = arr[j];
            j = j-1;
        }
C3      arr[j+1] = key;
    }
}
```

Insertion Sort – Is it better than bubble or selection?

Although it is one of the elementary sorting algorithms with $O(n^2)$ worst-case time, insertion sort is the algorithm of choice either when the data is nearly sorted (because it is adaptive) or when the problem size is small (because it has low overhead of swapping).

Prove empirically – assignment

Insertion sort – linked list

Shell Sort

- When there's only a few hundred elements, insertion sort is ok. But **shell sort** is only a few extra lines of code, and scales considerably better.

Shell Sort - Donald Shell published the first version of this sort in 1959

- Its a generalisation of the insertion sort
- In this sorting algorithm we compare elements that are distant apart than adjacent
- We start by comparing elements that are at a certain distance apart.
- So, if there are ***n*** elements then we start with a value ***gap*** < ***n***
- In each pass we keep reducing the value of gap till we reach the last pass when gap is 1
- In the last pass shell sort is like insertion sort

Shell Sort – Walk Through

There are total 9 elements
So, **N = 9**

array index	0	1	2	3	4	5	6	7	8
elements	14	18	19	37	23	40	29	30	11

Now we will find a **gap** not greater than **N**

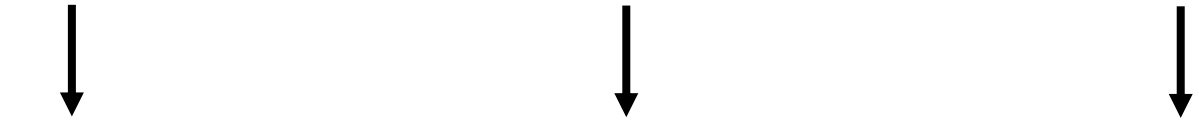
So, for pass 1
We have gap = 4

Where,
gap = floor (N/2)
= floor (9/2)
= floor (4.5)
= 4

We lower down the value to nearest integer

Shell Sort – Walk Through

pass = 1
gap = 4



array index	0	1	2	3	4	5	6	7	8
elements	14	18	19	37	23	40	29	30	11

gap 4 means

If we consider the first element at index 0

The next element will be at index 0+4 = 4

and third element will be at index 4+4 = 8

Shell Sort – Walk Through

pass = 1
gap = 4

array index	0	1	2	3	4	5	6	7	8
elements	14	18	19	37	23	40	29	30	11

Similarly, if we consider the element at index 1

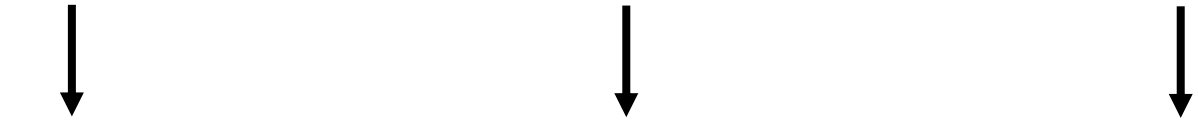
The next element will be at index 1+4 = 5

and the third element will be at index 5+4 = 9

but there is no 9^{th} index in the above array so we will just have index 1 and 5

Shell Sort – Walk Through

pass = 1
gap = 4



array index	0	1	2	3	4	5	6	7	8
elements	14	18	19	37	23	40	29	30	11

We then swap the elements if they are not in correct position

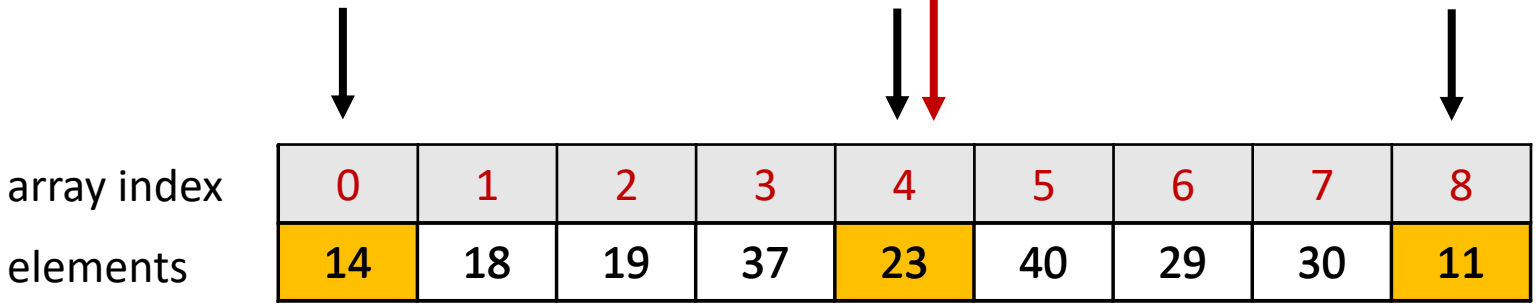
Lets dive in and sort these elements to see how it works

Shell Sort – Walk Through

pass = 1
gap = 4

We compare elements that are
at the left hand side of **end**

end



We start from index 0

and mark the elements at gap 4 apart

note!

end denotes the index covered so far.
we will complete a pass when **end** reaches the last index

Shell Sort – Walk Through

We compare elements that are at the left hand side of **end**

end

pass = 1
gap = 4

array index	0	1	2	3	4	5	6	7	8
elements	14	18	19	37	23	40	29	30	11

is 14 > 23 ?

NO

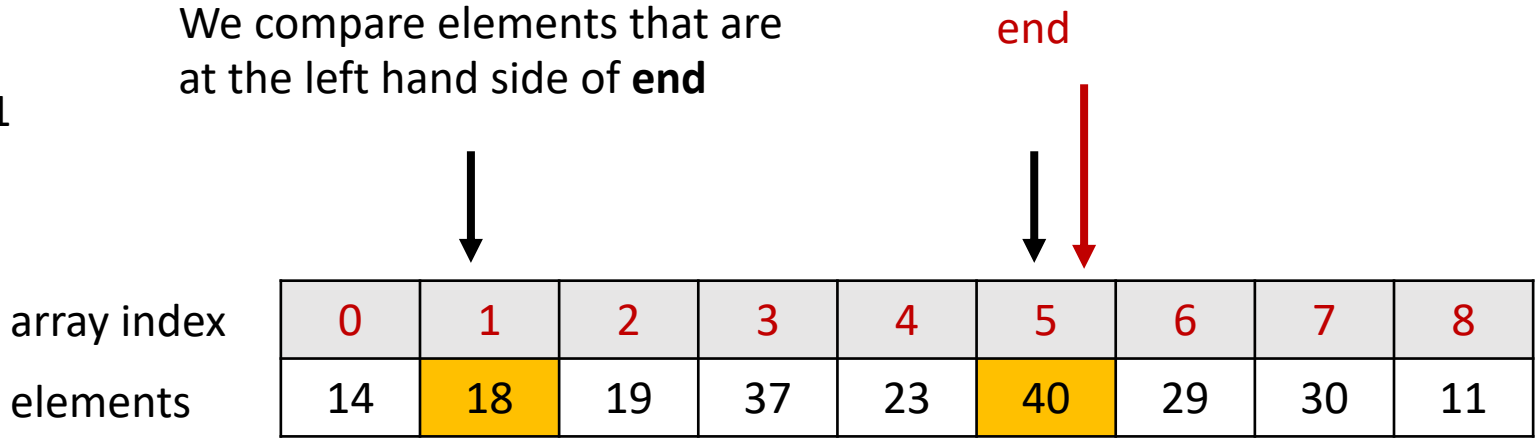
note!

so we shift one position towards right hand side
And mark the next group of elements gap 4 apart

Shell Sort – Walk Through

We compare elements that are
at the left hand side of **end**

pass = 1
gap = 4



is 18 > 40 ?

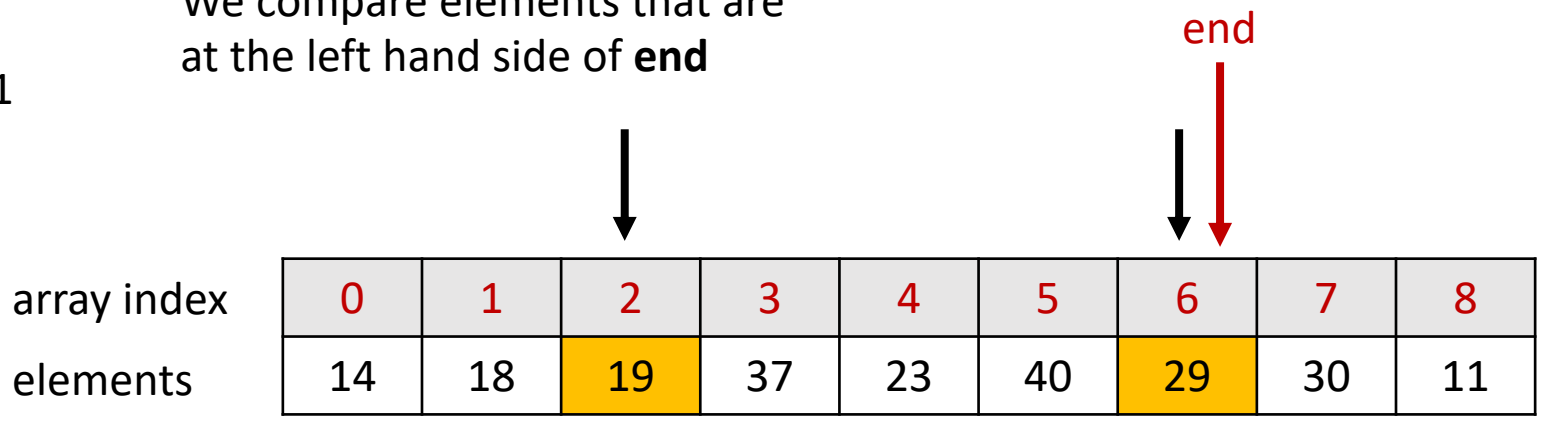
NO

so we shift one position towards right hand side
And mark the next group of elements gap 4 apart

Shell Sort – Walk Through

pass = 1
gap = 4

We compare elements that are
at the left hand side of **end**



array index	0	1	2	3	4	5	6	7	8
elements	14	18	19	37	23	40	29	30	11

is 19 > 29 ?

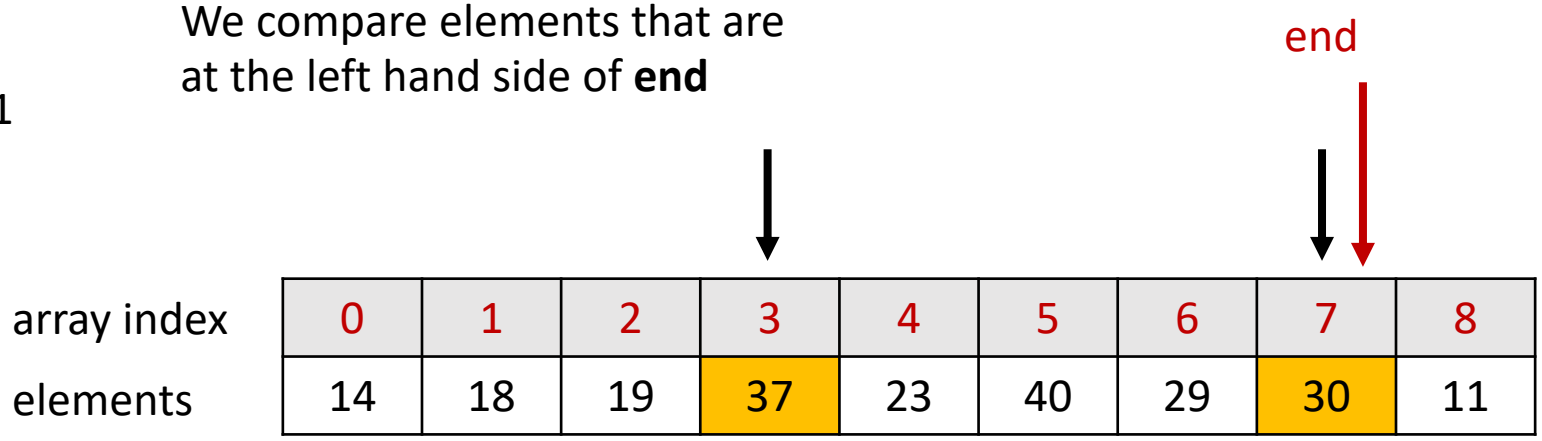
NO

so we shift one position towards right hand side
And mark the next group of elements gap 4 apart

Shell Sort – Walk Through

We compare elements that are at the left hand side of **end**

pass = 1
gap = 4



array index	0	1	2	3	4	5	6	7	8
elements	14	18	19	37	23	40	29	30	11

is 37 > 30 ?

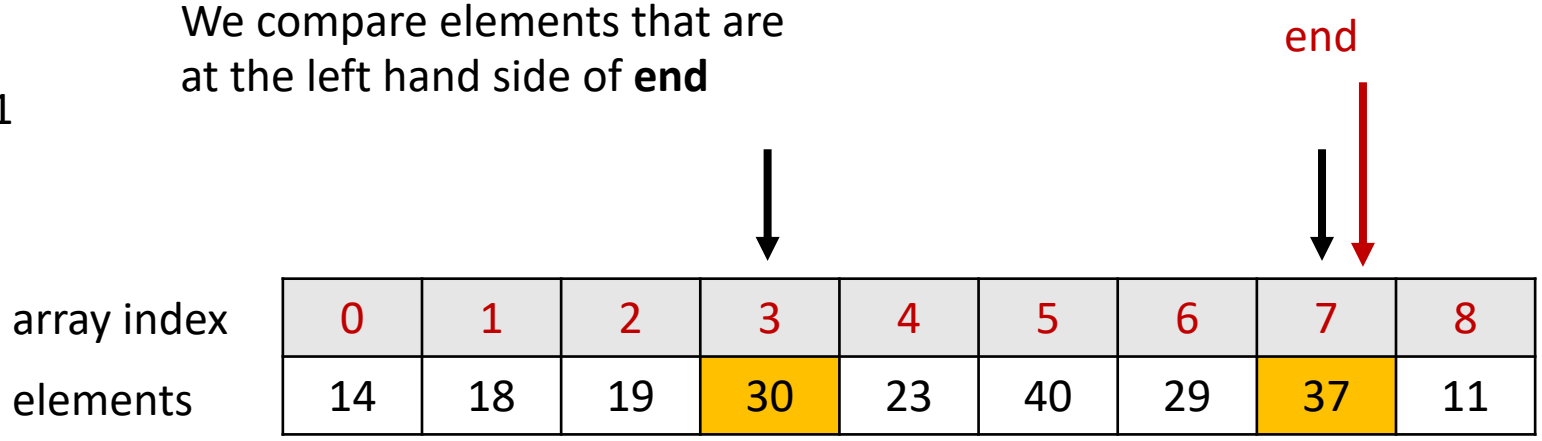
YES

so we swap them

Shell Sort – Walk Through

We compare elements that are
at the left hand side of **end**

pass = 1
gap = 4



array index	0	1	2	3	4	5	6	7	8
elements	14	18	19	30	23	40	29	37	11

is 37 > 30 ?

YES

so we swap them

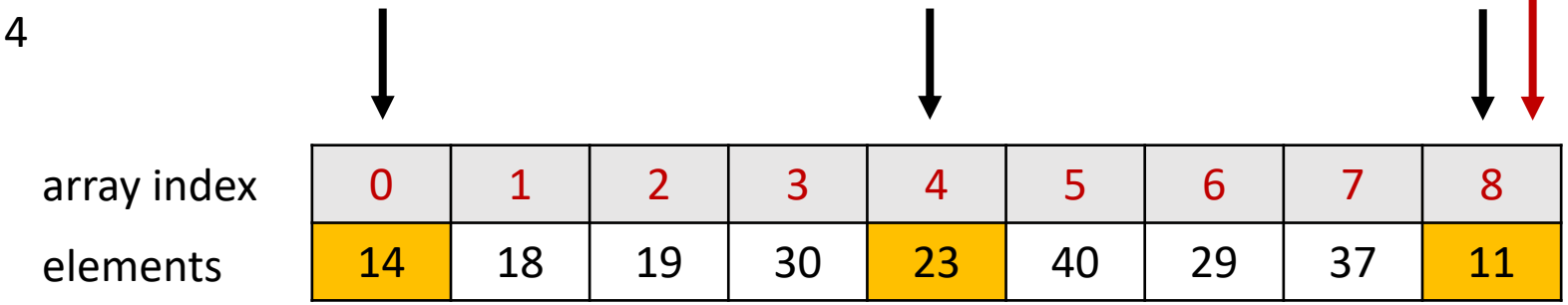
and shift one position towards right hand side
And mark the next group of elements gap 4 apart

Shell Sort – Walk Through

We compare elements that are
at the left hand side of **end**

end

pass = 1
gap = 4



array index	0	1	2	3	4	5	6	7	8
elements	14	18	19	30	23	40	29	37	11

is 23 > 11

YES

so we swap them

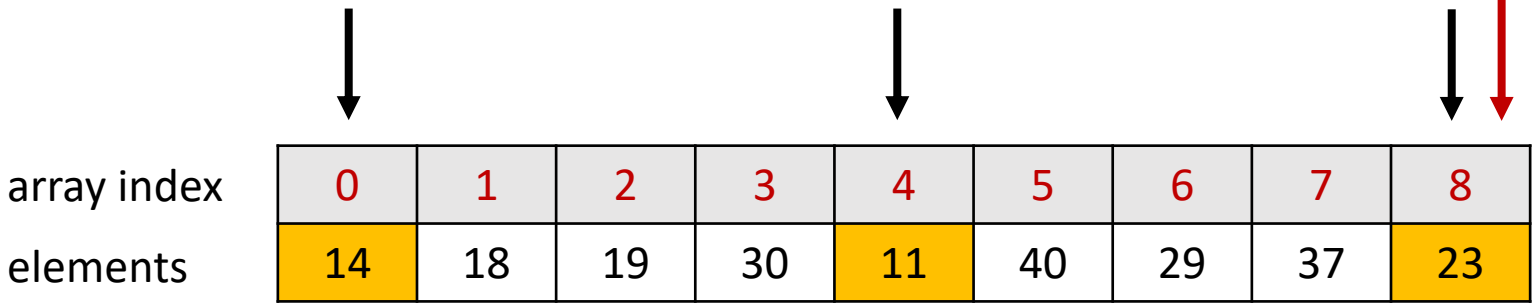
Note!
We marked three elements as they are gap 4 apart and our task is to put the elements in correct position

Shell Sort – Walk Through

pass = 1
gap = 4

We compare elements that are
at the left hand side of **end**

end



array index	0	1	2	3	4	5	6	7	8
elements	14	18	19	30	11	40	29	37	23

is 23 > 11

YES

so we swap them

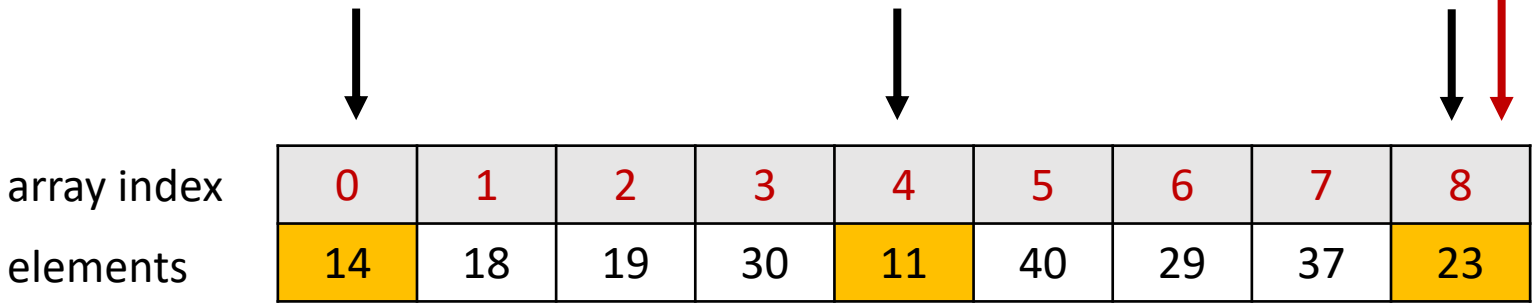
Note!
We marked three elements as they are gap 4 apart and our task is to put the elements in correct position

Shell Sort – Walk Through

pass = 1
gap = 4

We compare elements that are
at the left hand side of **end**

end



now we compare 14 and 11

is 14 > 11

YES

so we swap them

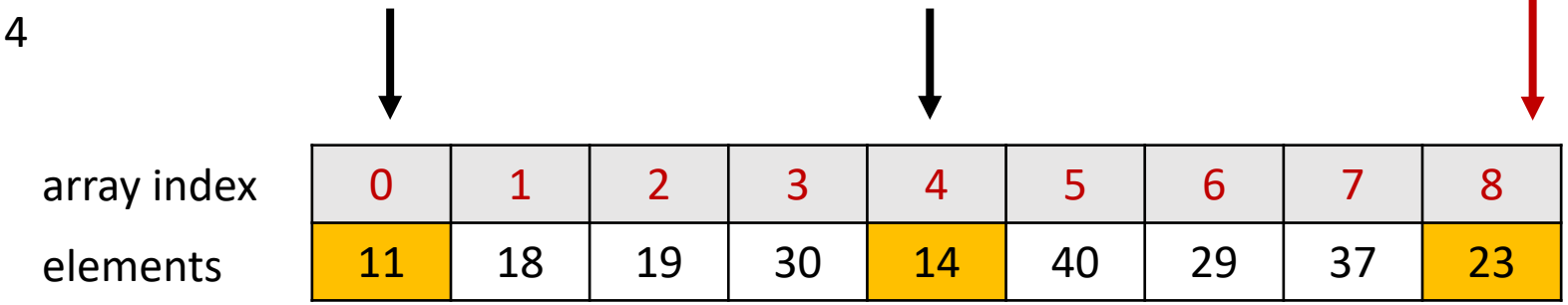
Note!
We marked three elements as they are gap 4 apart and our task is to put the elements in correct position

Shell Sort – Walk Through

We compare elements that are
at the left hand side of **end**

end

pass = 1
gap = 4



array index	0	1	2	3	4	5	6	7	8
elements	11	18	19	30	14	40	29	37	23

as **end** has reached the last index of the array
this means we have completed pass 1 and its
Time to move to pass 2

Note!
We marked three
elements as they
are gap 4 apart
and our task is to
put the elements in
correct position

Shell Sort – Walk Through

pass = 2
gap = 2

array index	0	1	2	3	4	5	6	7	8
elements	11	18	19	30	14	40	29	37	23

for pass 2 we will reduce the gap using the formula

gap = floor (gap/2)

= floor (4/2) [in pass 1, gap was 4]

= floor (2)

= 2

So, for pass 2
we have gap = 2

We lower down the value to nearest integer

Shell Sort – Walk Through

We compare elements that are at the left hand side of **end**

end

pass = 2
gap = 2

array index	0	1	2	3	4	5	6	7	8
elements	11	18	19	30	14	40	29	37	23

we start from index 0

and mark the elements at gap 2 apart

note!

end denotes the index covered so far
we will complete a pass when **end** reaches the last index

Shell Sort – Walk Through

We compare elements that are at the left hand side of **end**

end

pass = 2
gap = 2

array index	0	1	2	3	4	5	6	7	8
elements	11	18	19	30	14	40	29	37	23

is 11 > 19 ?

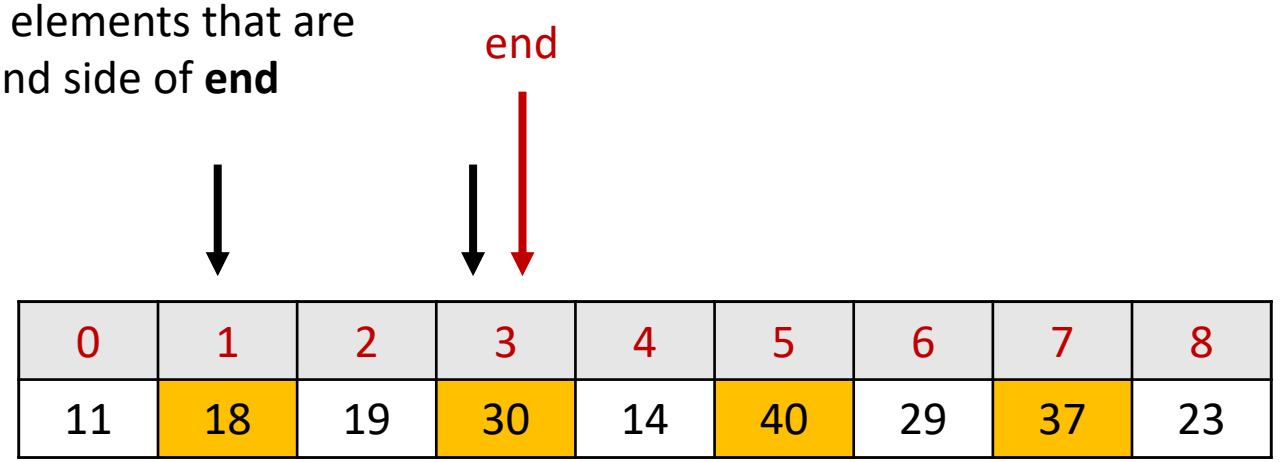
NO

so we shift one position towards right hand side
And mark the next group of elements gap 2 apart

Shell Sort – Walk Through

We compare elements that are
at the left hand side of **end**

pass = 2
gap = 2



array index	0	1	2	3	4	5	6	7	8
elements	11	18	19	30	14	40	29	37	23

is 18 > 30 ?

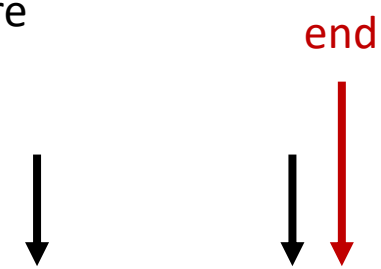
NO

so we shift one position towards right hand side
And mark the next group of elements gap 2 apart

Shell Sort – Walk Through

We compare elements that are
at the left hand side of **end**

pass = 2
gap = 2



array index	0	1	2	3	4	5	6	7	8
elements	11	18	19	30	14	40	29	37	23

is 19 > 14 ?

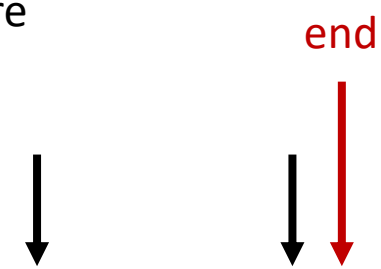
YES

so we swap them

Shell Sort – Walk Through

We compare elements that are
at the left hand side of **end**

pass = 2
gap = 2



array index	0	1	2	3	4	5	6	7	8
elements	11	18	14	30	19	40	29	37	23

is 19 > 14 ?

YES

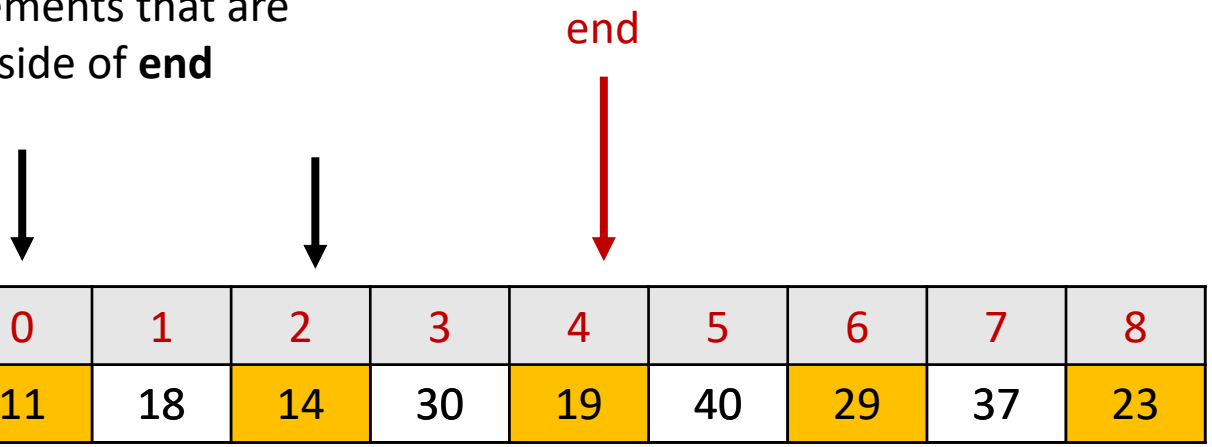
so we swap them

now we check the other pair left side of **end**

Shell Sort – Walk Through

We compare elements that are
at the left hand side of **end**

pass = 2
gap = 2



is 11 > 14 ?

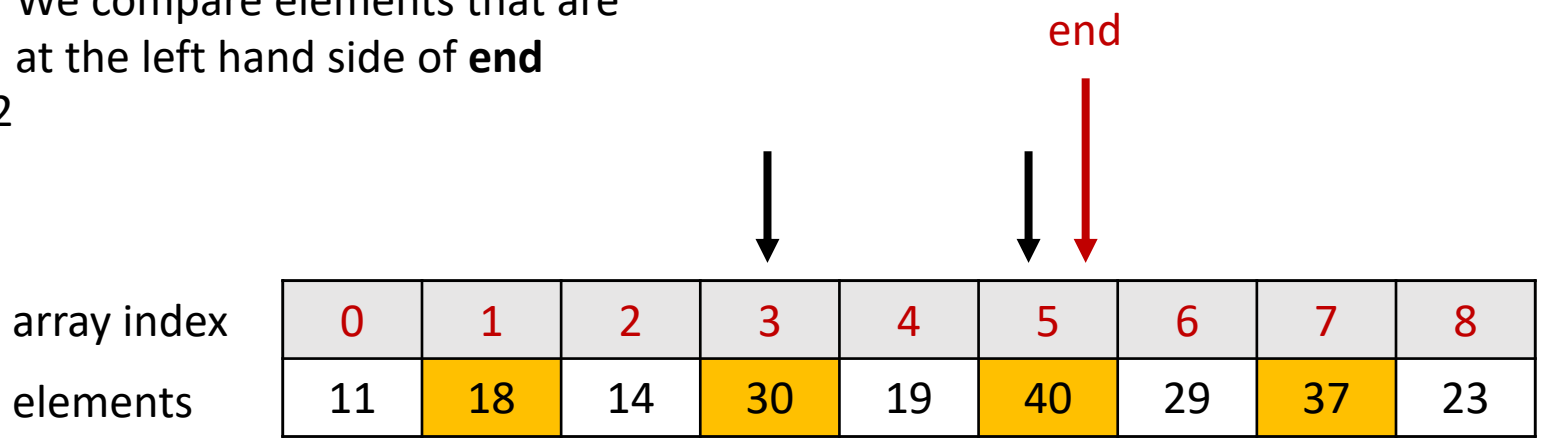
NO

so we shift one position towards right hand side
And mark the next group of elements gap 2 apart

Shell Sort – Walk Through

We compare elements that are
at the left hand side of **end**

pass = 2
gap = 2



is 30 > 40 ?

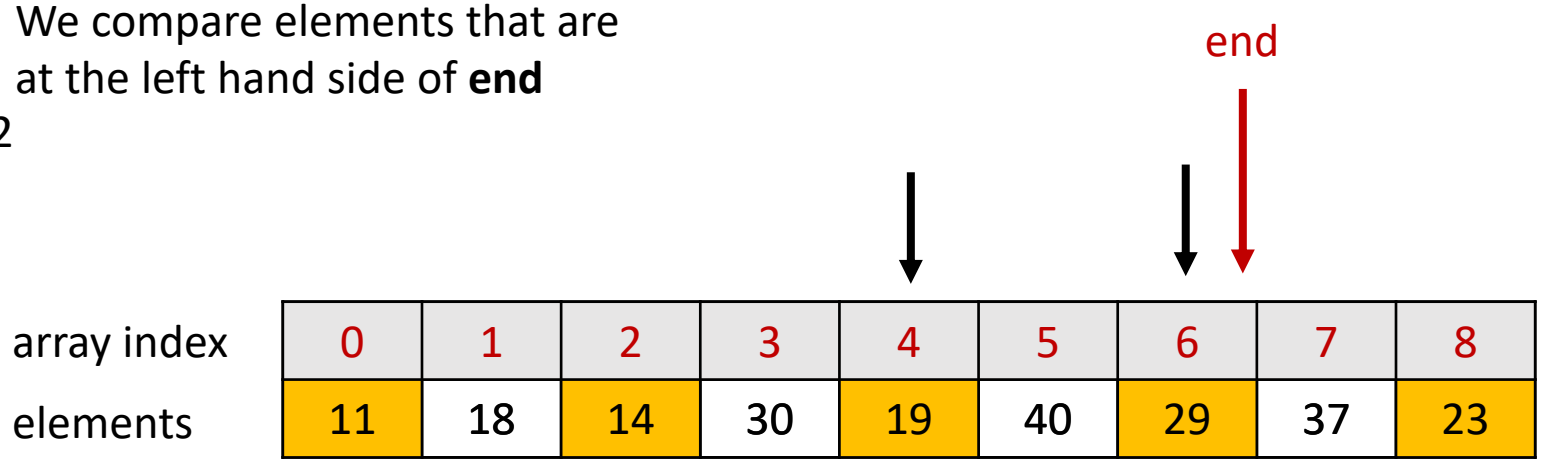
NO

so we shift one position towards right hand side
And mark the next group of elements gap 2 apart

Shell Sort – Walk Through

We compare elements that are
at the left hand side of **end**

pass = 2
gap = 2



array index	0	1	2	3	4	5	6	7	8
elements	11	18	14	30	19	40	29	37	23

is 19 > 29 ?

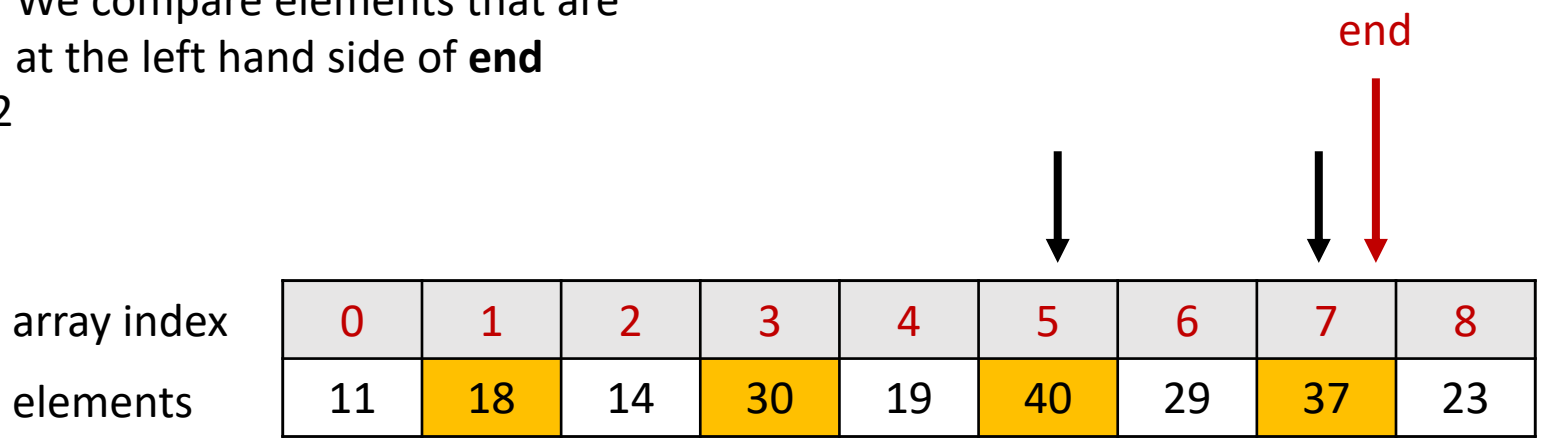
NO

so we shift one position towards right hand side
And mark the next group of elements gap 2 apart

Shell Sort – Walk Through

We compare elements that are
at the left hand side of **end**

pass = 2
gap = 2



array index	0	1	2	3	4	5	6	7	8
elements	11	18	14	30	19	40	29	37	23

is 40 > 37 ?

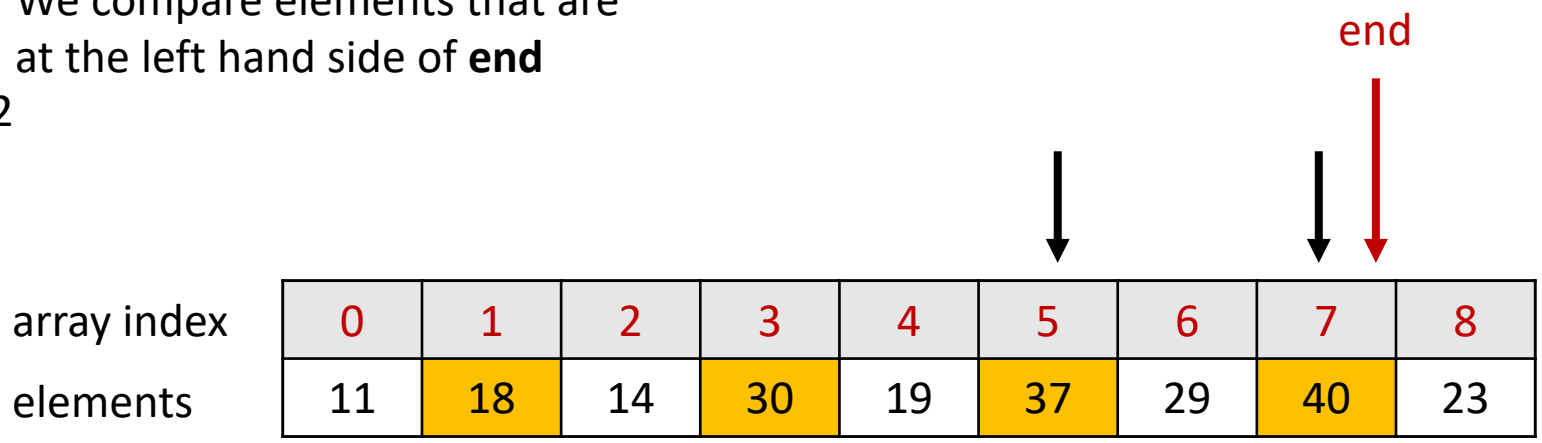
YES

so we swap them

Shell Sort – Walk Through

We compare elements that are
at the left hand side of **end**

pass = 2
gap = 2



array index	0	1	2	3	4	5	6	7	8
elements	11	18	14	30	19	37	29	40	23

is 40 > 37 ?

YES

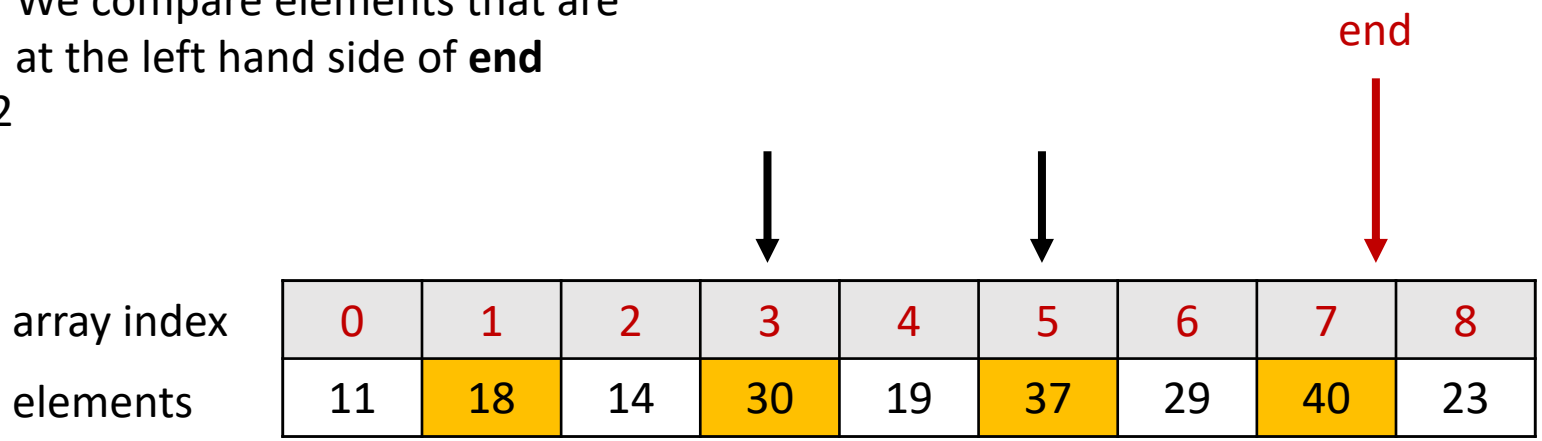
so we swap them

now we check the other pair left side of **end**

Shell Sort – Walk Through

We compare elements that are
at the left hand side of **end**

pass = 2
gap = 2



array index	0	1	2	3	4	5	6	7	8
elements	11	18	14	30	19	37	29	40	23

is 30 > 37 ?

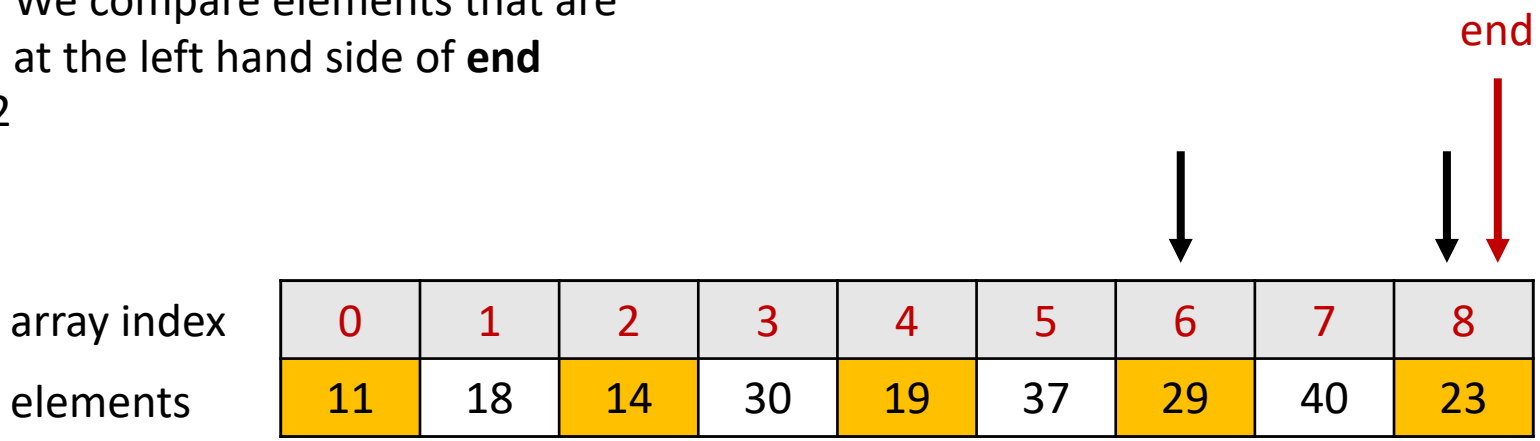
NO

so we shift one position towards right hand side
And mark the next group of elements gap 2 apart

Shell Sort – Walk Through

We compare elements that are
at the left hand side of **end**

pass = 2
gap = 2



is 29 > 23 ?

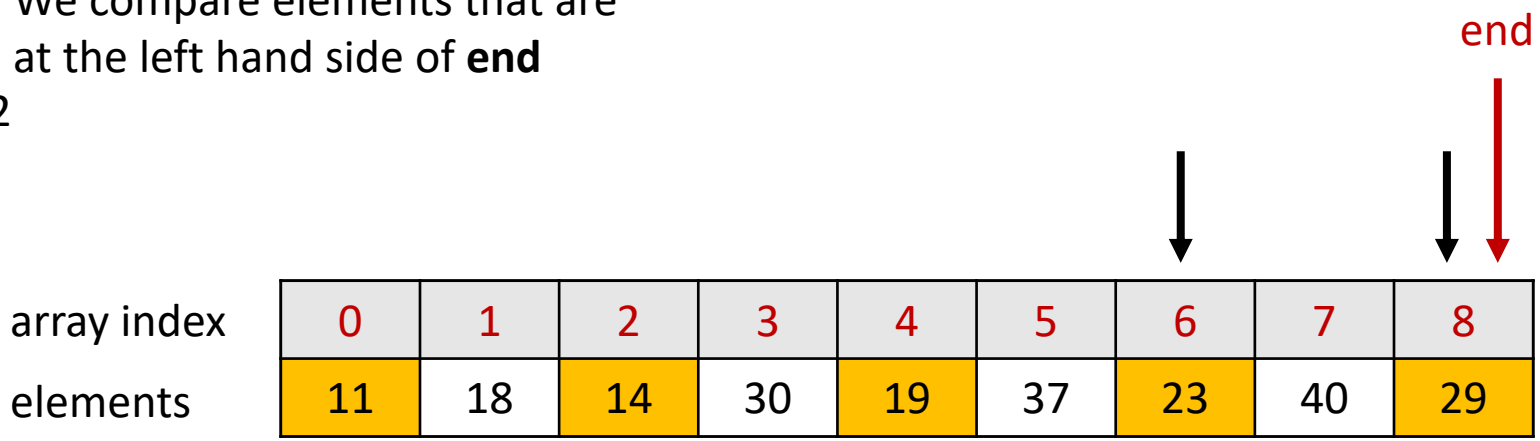
YES

so we swap them

Shell Sort – Walk Through

We compare elements that are
at the left hand side of **end**

pass = 2
gap = 2



array index	0	1	2	3	4	5	6	7	8
elements	11	18	14	30	19	37	23	40	29

is 29 > 23 ?

YES

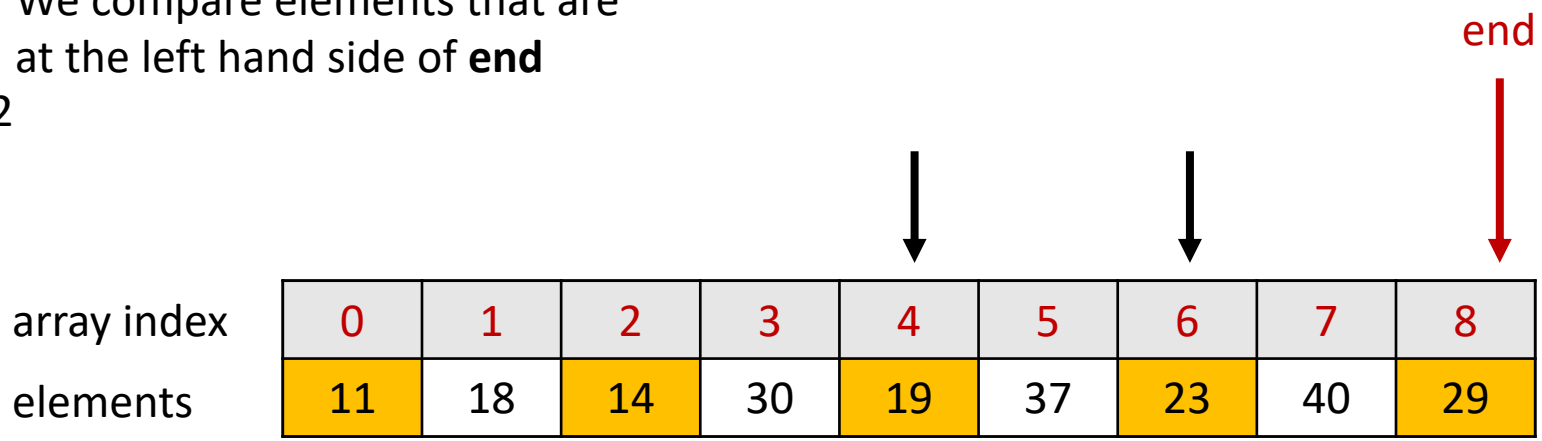
so we swap them

now we check the other pair left side of **end**

Shell Sort – Walk Through

We compare elements that are
at the left hand side of **end**

pass = 2
gap = 2



array index	0	1	2	3	4	5	6	7	8
elements	11	18	14	30	19	37	23	40	29

is 19 > 23 ?

NO

so we stop here as **end** has reached the last index of the array

Time to move to pass 3

Shell Sort – Walk Through

pass = 3
gap = 1

note!
when gap = 1
shell sort is like insertion sort
and we will have sorted elements at the end of this process

array index	0	1	2	3	4	5	6	7	8
elements	11	18	14	30	19	37	23	40	29

For pass 3 we will reduce the gap using the formula

gap = floor (gap/2)

= floor (2/2) [in pass 2, gap was 2]

= floor (1)

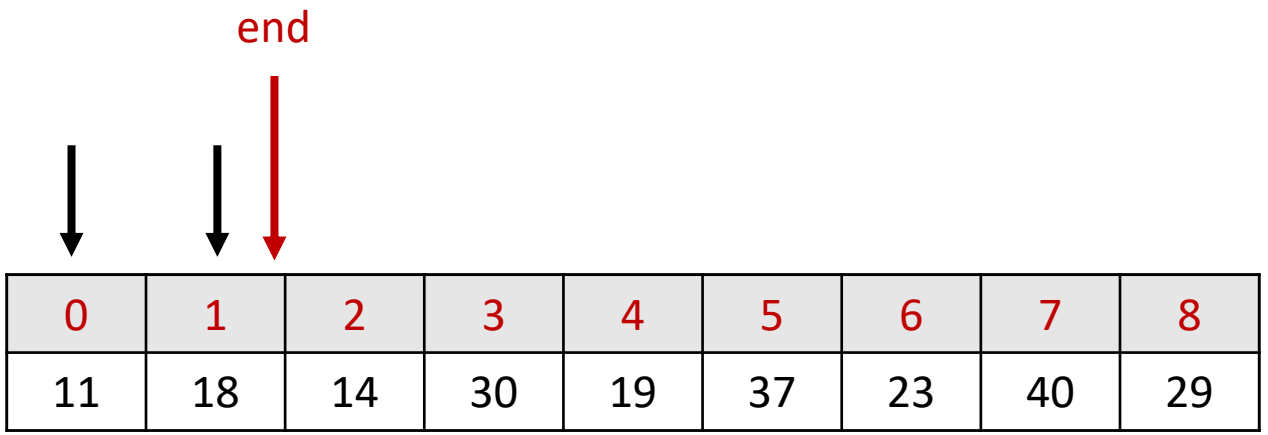
= 1

So, for pass 3
we have gap = 1

We lower down the value to nearest integer

Shell Sort – Walk Through

pass = 3
gap = 1



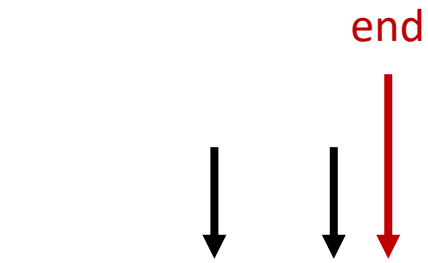
is 11 > 18 ?

NO

so we shift one position towards right hand side
And mark the next group of elements gap 1 apart

Shell Sort – Walk Through

pass = 3
gap = 1



array index	0	1	2	3	4	5	6	7	8
elements	11	18	14	30	19	37	23	40	29

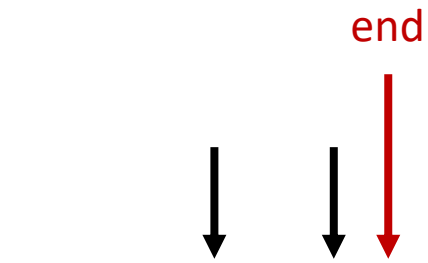
is 18 > 14 ?

YES

so we swap them

Shell Sort – Walk Through

pass = 3
gap = 1



array index	0	1	2	3	4	5	6	7	8
elements	11	14	18	30	19	37	23	40	29

is 18 > 14 ?

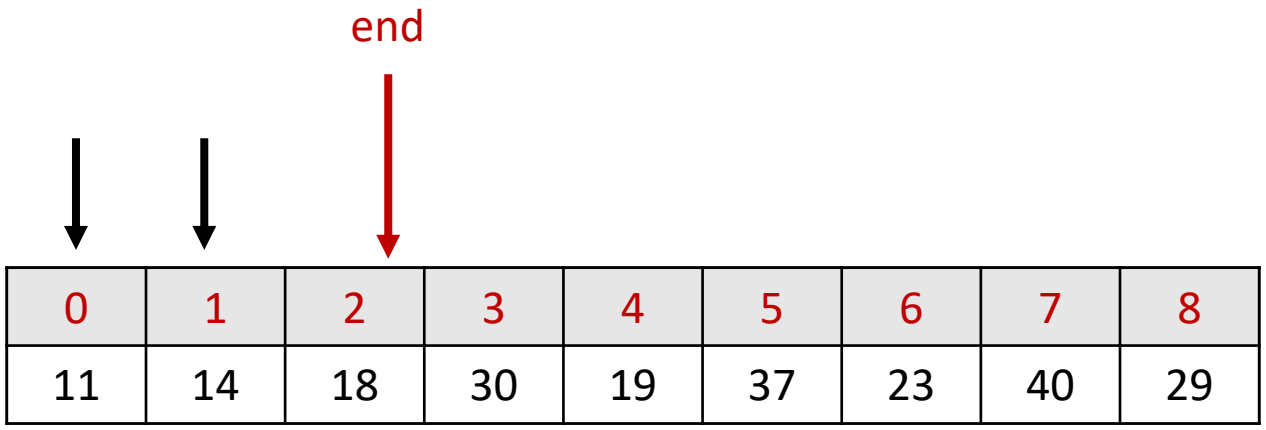
YES

so we swap them

now we check the other pair left side of **end**

Shell Sort – Walk Through

pass = 3
gap = 1



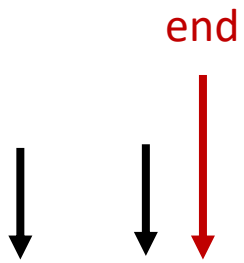
is 11 > 14 ?

NO

so we shift one position towards right hand side
And mark the next group of elements gap 1 apart

Shell Sort – Walk Through

pass = 3
gap = 1



array index	0	1	2	3	4	5	6	7	8
elements	11	14	18	30	19	37	23	40	29

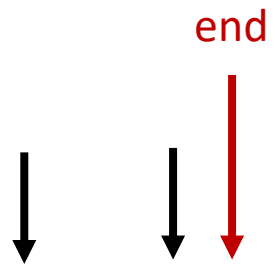
is 18 > 30 ?

NO

so we shift one position towards right hand side
And mark the next group of elements gap 1 apart

Shell Sort – Walk Through

pass = 3
gap = 1



array index	0	1	2	3	4	5	6	7	8
elements	11	14	18	30	19	37	23	40	29

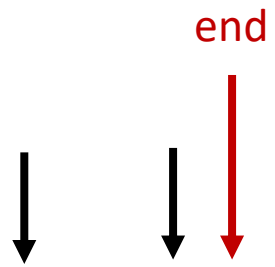
is 30 > 19 ?

YES

so we swap them

Shell Sort – Walk Through

pass = 3
gap = 1



array index	0	1	2	3	4	5	6	7	8
elements	11	14	18	19	30	37	23	40	29

is 30 > 19 ?

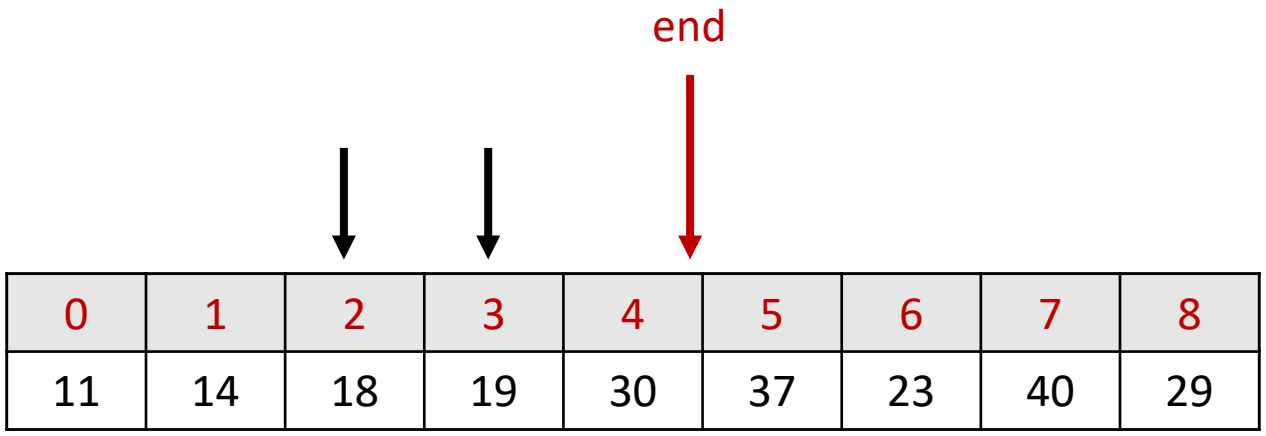
YES

so we swap them

now we check the other pair left side of **end**

Shell Sort – Walk Through

pass = 3
gap = 1



array index	0	1	2	3	4	5	6	7	8
elements	11	14	18	19	30	37	23	40	29

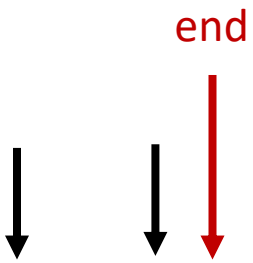
is 18 > 19 ?

NO

so we shift one position towards right hand side
And mark the next group of elements gap 1 apart

Shell Sort – Walk Through

pass = 3
gap = 1



array index	0	1	2	3	4	5	6	7	8
elements	11	14	18	19	30	37	23	40	29

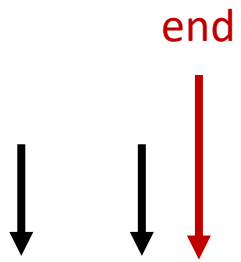
is 30 > 37 ?

NO

so we shift one position towards right hand side
And mark the next group of elements gap 1 apart

Shell Sort – Walk Through

pass = 3
gap = 1



array index	0	1	2	3	4	5	6	7	8
elements	11	14	18	19	30	37	23	40	29

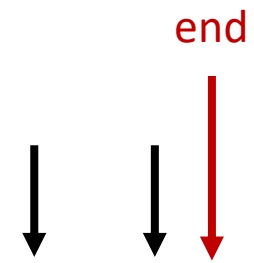
is 37 > 23 ?

YES

so we swap them

Shell Sort – Walk Through

pass = 3
gap = 1



array index	0	1	2	3	4	5	6	7	8
elements	11	14	18	19	30	23	37	40	29

is 37 > 23 ?

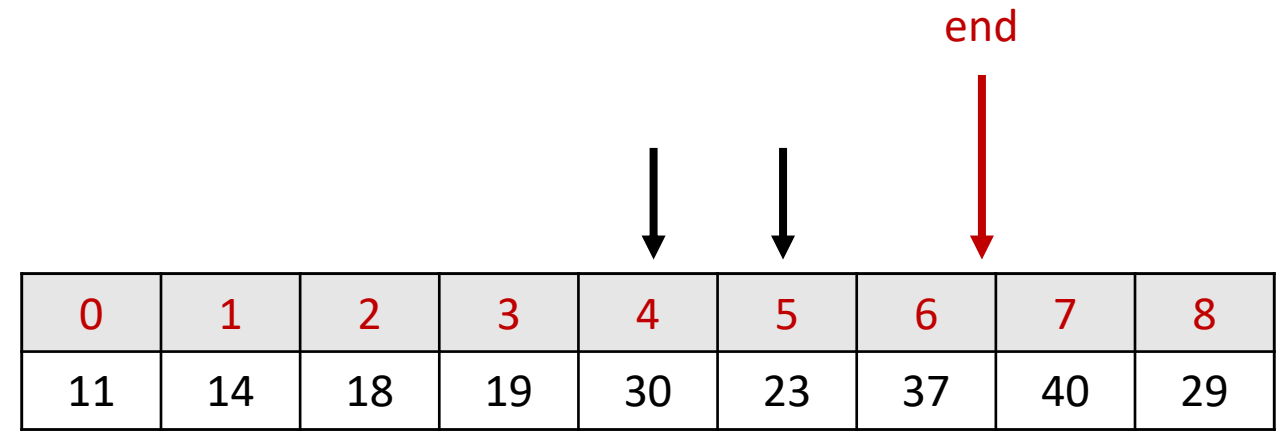
YES

so we swap them

now we check the other pair left side of **end**

Shell Sort – Walk Through

pass = 3
gap = 1



array index	0	1	2	3	4	5	6	7	8
elements	11	14	18	19	30	23	37	40	29

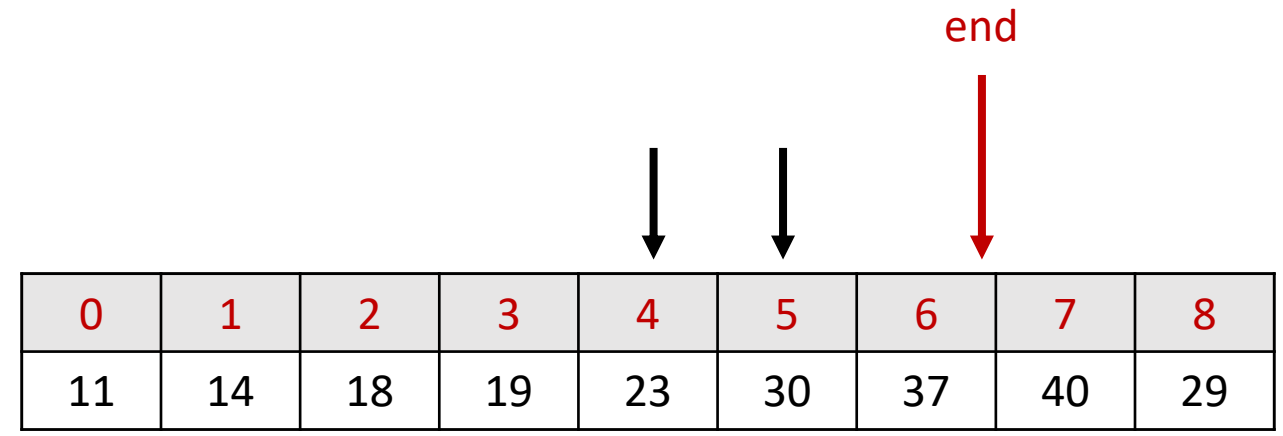
is 30 > 23 ?

YES

so we swap them

Shell Sort – Walk Through

pass = 3
gap = 1



array index	0	1	2	3	4	5	6	7	8
elements	11	14	18	19	23	30	37	40	29

is 30 > 23 ?

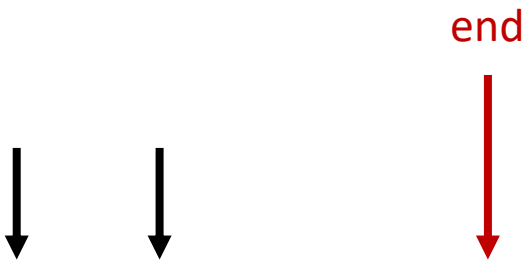
YES

so we swap them

now we check the other pair left side of **end**

Shell Sort – Walk Through

pass = 3
gap = 1



array index	0	1	2	3	4	5	6	7	8
elements	11	14	18	19	23	30	37	40	29

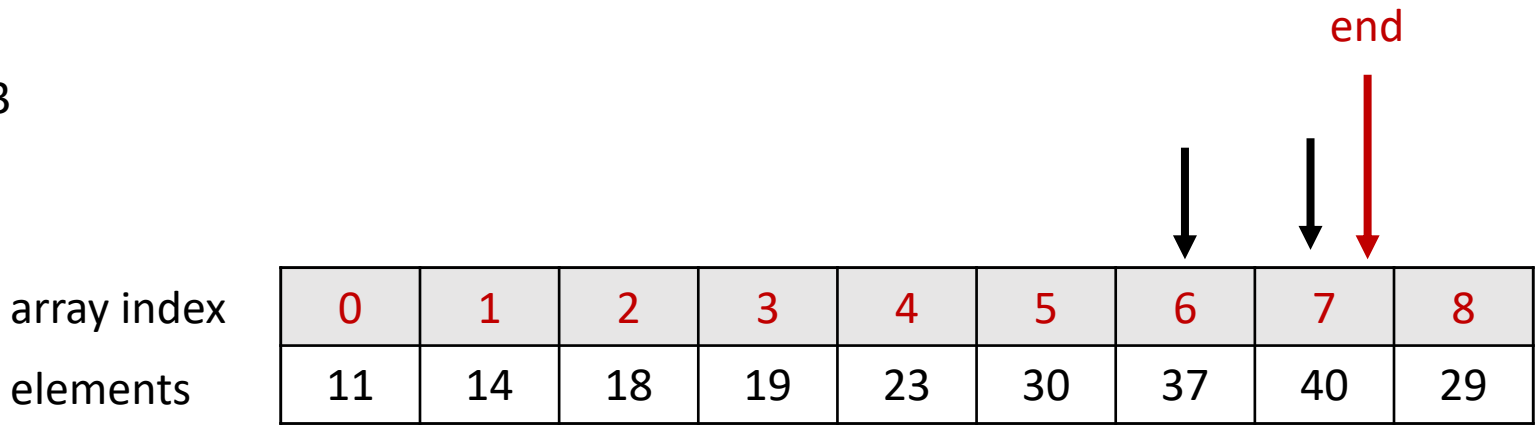
is 19 > 23 ?

NO

so we shift one position towards right hand side
And mark the next group of elements gap 1 apart

Shell Sort – Walk Through

pass = 3
gap = 1



array index	0	1	2	3	4	5	6	7	8
elements	11	14	18	19	23	30	37	40	29

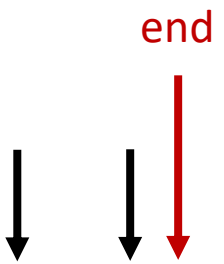
is 37 > 40 ?

NO

so we shift one position towards right hand side
And mark the next group of elements gap 1 apart

Shell Sort – Walk Through

pass = 3
gap = 1



array index	0	1	2	3	4	5	6	7	8
elements	11	14	18	19	23	30	37	40	29

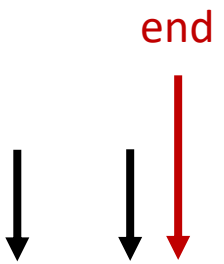
is 40 > 29 ?

YES

so we swap them

Shell Sort – Walk Through

pass = 3
gap = 1



array index	0	1	2	3	4	5	6	7	8
elements	11	14	18	19	23	30	37	29	40

is 40 > 29 ?

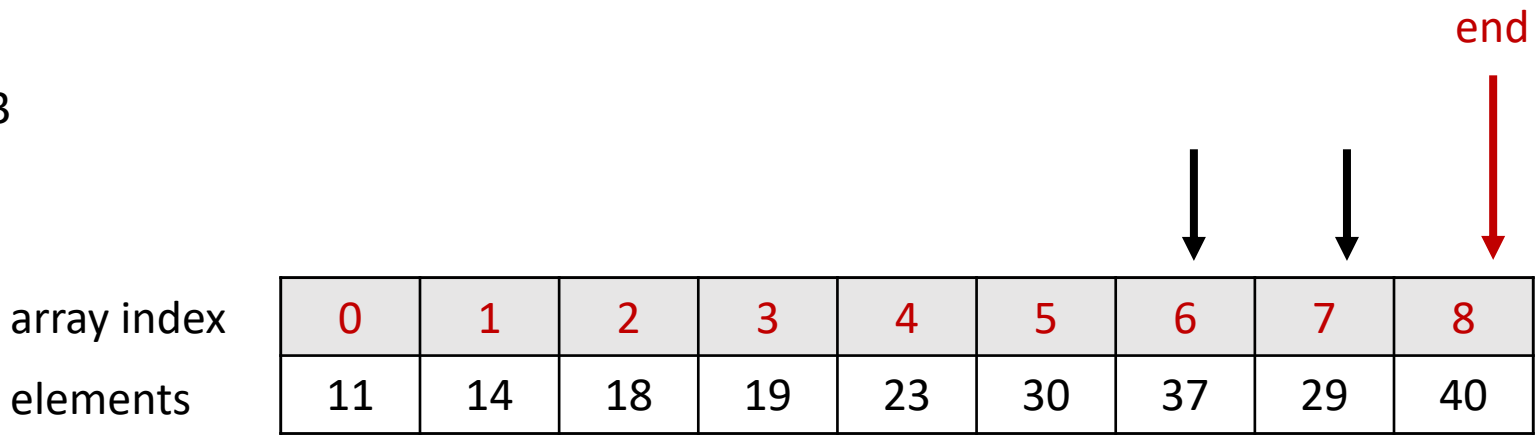
YES

so we swap them

now we check the other pair left side of **end**

Shell Sort – Walk Through

pass = 3
gap = 1



array index	0	1	2	3	4	5	6	7	8
elements	11	14	18	19	23	30	37	29	40

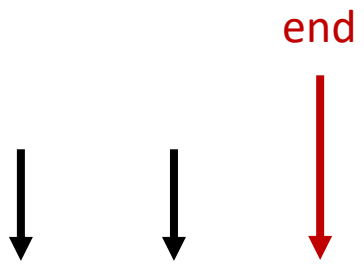
is 37 > 29 ?

YES

so we swap them

Shell Sort – Walk Through

pass = 3
gap = 1



array index	0	1	2	3	4	5	6	7	8
elements	11	14	18	19	23	30	29	37	40

is 37 > 29 ?

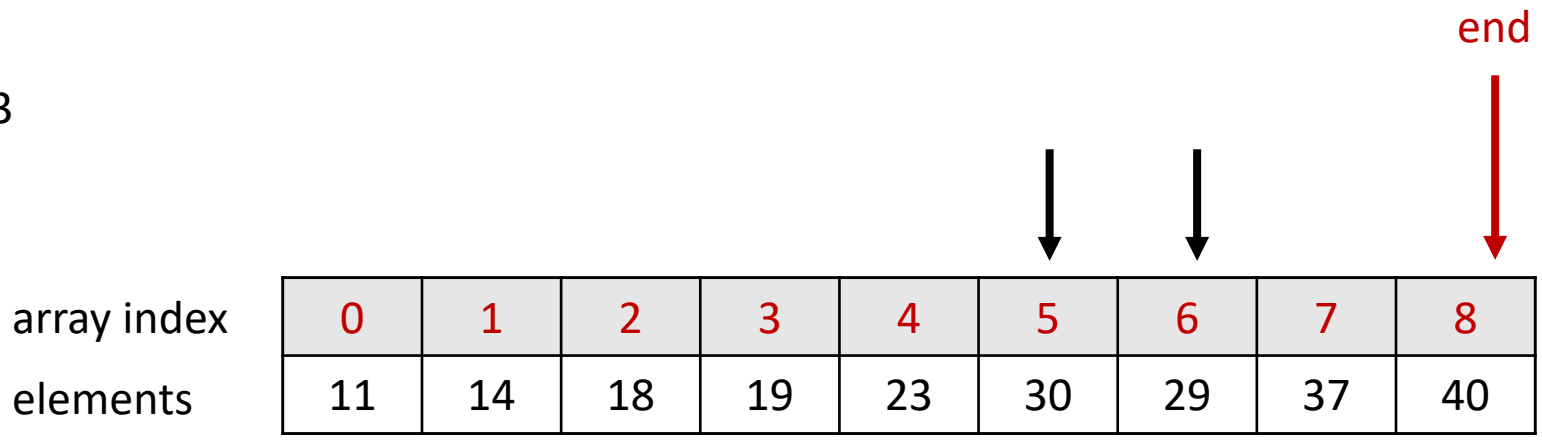
YES

so we swap them

now we check the other pair left side of **end**

Shell Sort – Walk Through

pass = 3
gap = 1



array index	0	1	2	3	4	5	6	7	8
elements	11	14	18	19	23	30	29	37	40

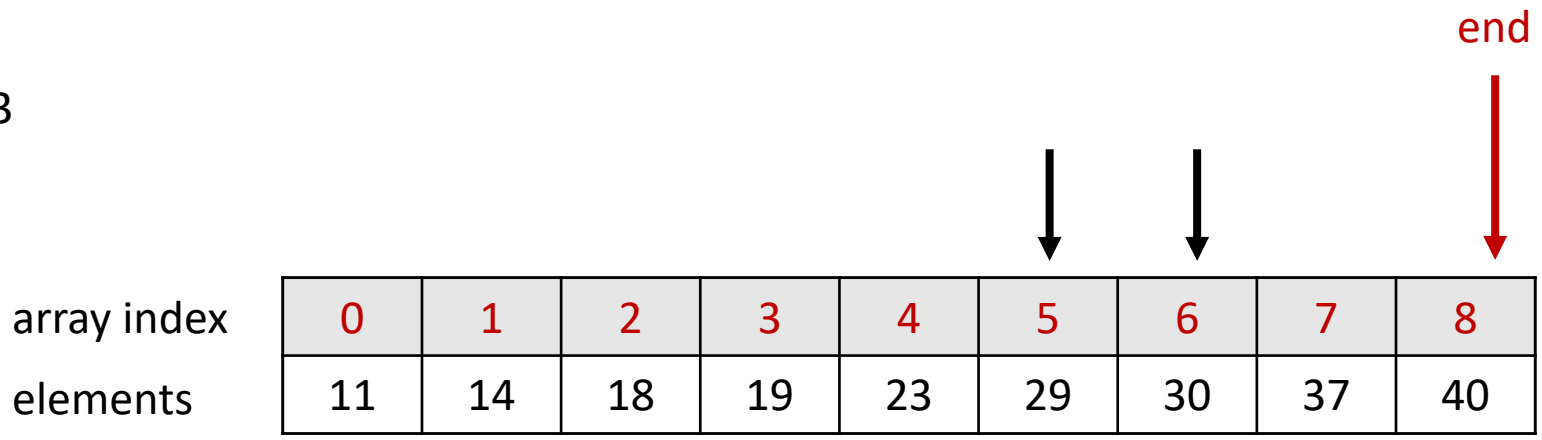
is 30 > 29 ?

YES

so we swap them

Shell Sort – Walk Through

pass = 3
gap = 1



array index	0	1	2	3	4	5	6	7	8
elements	11	14	18	19	23	29	30	37	40

is 30 > 29 ?

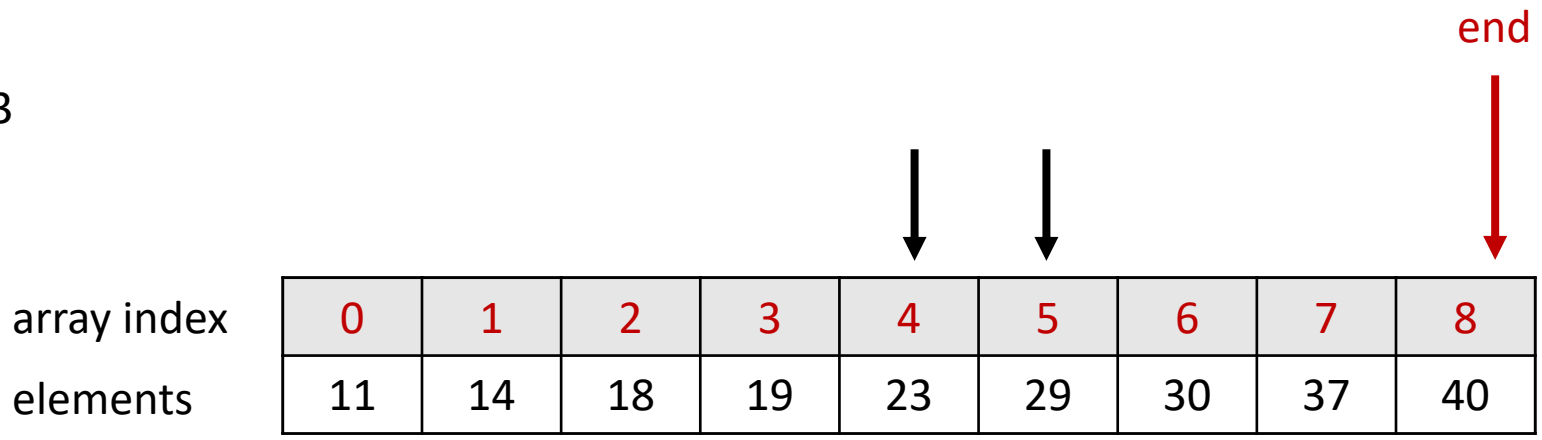
YES

so we swap them

now we check the other pair left side of **end**

Shell Sort – Walk Through

pass = 3
gap = 1



array index	0	1	2	3	4	5	6	7	8
elements	11	14	18	19	23	29	30	37	40

is 23 > 29 ?

NO

so we stop here as **end** has reached the last index of the array

Shell Sort – Walk Through

pass = 3
gap = 1

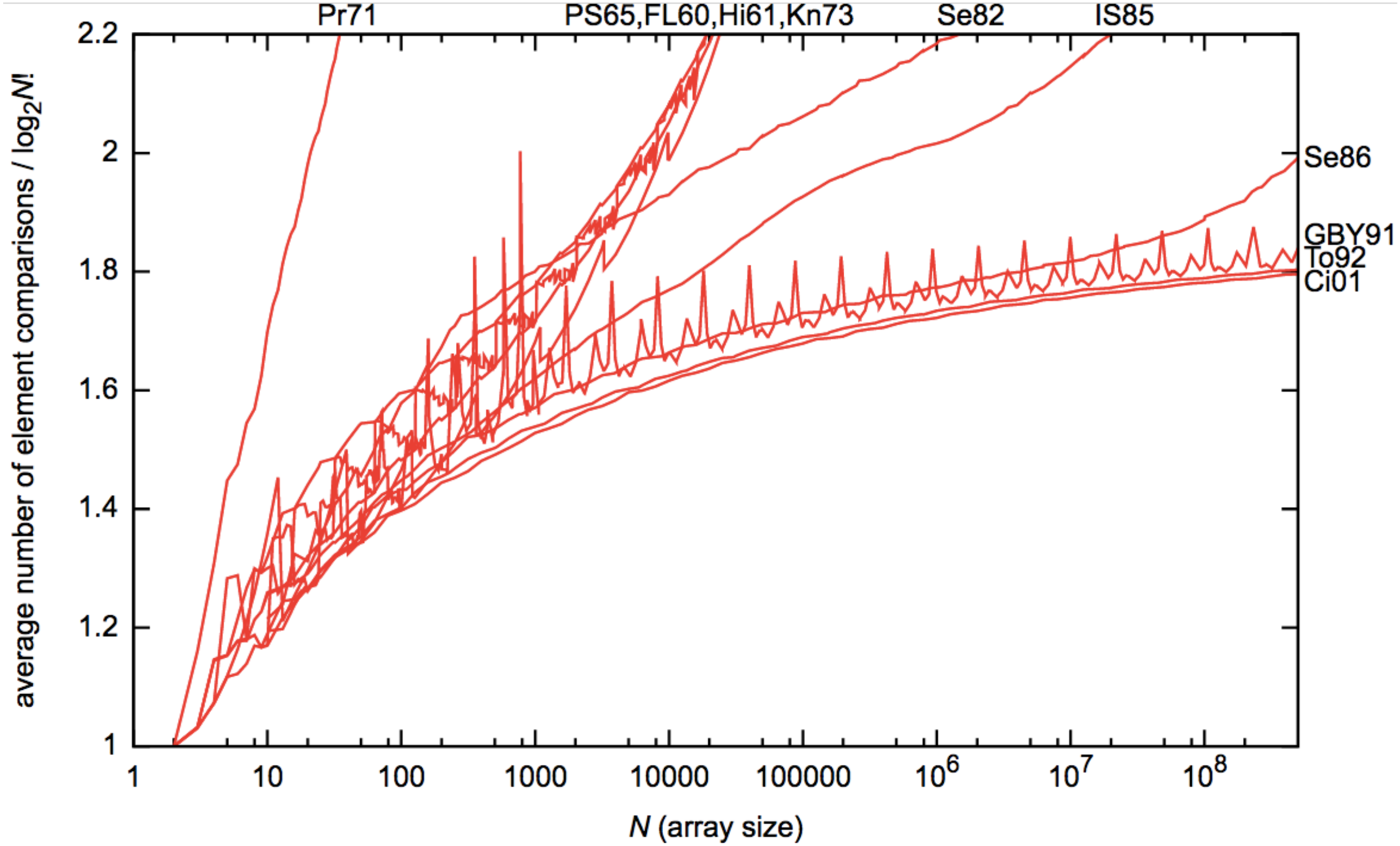
array index	0	1	2	3	4	5	6	7	8
elements	11	14	18	19	23	29	30	37	40

we have sorted the array

```
/* function to sort arr using shellSort */
int shellSort(int arr[], int n)
{
    // Start with a big gap, then reduce the gap
    for (int gap = n/2; gap > 0; gap /= 2)
    {
        // Do a gapped insertion sort for this gap size.
        // The first gap elements a[0..gap-1] are already in gapped order
        // keep adding one more element until the entire array is
        // gap sorted
        for (int i = gap; i < n; i += 1)
        {
            // add a[i] to the elements that have been gap sorted
            // save a[i] in temp and make a hole at position i
            int temp = arr[i];

            // shift earlier gap-sorted elements up until the correct
            // location for a[i] is found
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
                arr[j] = arr[j - gap];

            //  put temp (the original a[i]) in its correct location
            arr[j] = temp;
        }
    }
    return 0;
}
```


Pr71
PS65,FL60,Hi61,Kn73
Se82
IS85
Se86
GBY91
To92
Ci01
average number of element comparisons / log2N!
N (array size)
2.2
2
1.8
1.6
1.4
1.2
1
1
10
100
1000
10000
100000
10^6
10^7
10^8

Gap Sequence

- The question of deciding which gap sequence to use is difficult.
- Too few gaps slows down the passes, and too many gaps produces an overhead.
- However, the properties of thus obtained versions of Shellsort may be very different.

Gap Sequence

OEIS	General term ($k \geq 1$)	Concrete gaps	Worst-case time complexity	Author and year of publication
	$\left\lfloor \frac{N}{2^k} \right\rfloor$	$\left\lfloor \frac{N}{2} \right\rfloor, \left\lfloor \frac{N}{4} \right\rfloor, \ldots, 1$	$\Theta\left(N^2\right)$ [e.g. when $N = 2^p$]	Shell, 1959[4]
	$2\left\lfloor \frac{N}{2^{k+1}} \right\rfloor + 1$	$2\left\lfloor \frac{N}{4} \right\rfloor + 1, \ldots, 3, 1$	$\Theta\left(N^{\frac{3}{2}}\right)$	Frank & Lazarus, 1960[8]
A168604	$2^k - 1$	$1, 3, 7, 15, 31, 63, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Hibbard, 1963[9]
A083318	$2^k + 1$, prefixed with 1	$1, 3, 5, 9, 17, 33, 65, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Papernov & Stasevich, 1965[10]
A003586	Successive numbers of the form $2^p 3^q$ (3-smooth numbers)	$1, 2, 3, 4, 6, 8, 9, 12, \ldots$	$\Theta\left(N \log^2 N\right)$	Pratt, 1971[1]
A003462	$\frac{3^k - 1}{2}$, not greater than $\left\lceil \frac{N}{3} \right\rceil$	$1, 4, 13, 40, 121, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Pratt, 1971[1]
A036569	$\prod_I a_q$, where $a_q = \min\left\{n \in \mathbb{N} : n \geq \left(\frac{5}{2}\right)^{q+1}, \forall p : 0 \leq p < q \Rightarrow \gcd(a_p, n) = 1\right\}$ $I = \left\{0 \leq q < r \mid q \neq \frac{1}{2}\left(r^2 + r\right) - k\right\}$ $r = \left\lfloor \sqrt{2k + \sqrt{2k}} \right\rfloor$	$1, 3, 7, 21, 48, 112, \ldots$	$O\left(N^{1+\sqrt{\frac{8\ln(5/2)}{\ln(N)}}}\right)$	Incerpi & Sedgewick, 1985,[11] Knuth[3]
A036562	$4^k + 3 \cdot 2^{k-1} + 1$, prefixed with 1	$1, 8, 23, 77, 281, \ldots$	$O\left(N^{\frac{4}{3}}\right)$	Sedgewick, 1986[6]
A033622	$\begin{cases} 9\left(4^{k-1} - 2^{\frac{k}{2}}\right) + 1 & k \text{ even}, \\ 4^{k+1} - 6 \cdot 2^{(k+1)/2} + 1 & k \text{ odd} \end{cases}$	$1, 5, 19, 41, 109, \ldots$	$O\left(N^{\frac{4}{3}}\right)$	Sedgewick, 1986[12]
	$h_k = \max\left\{\left\lfloor \frac{5h_{k-1}}{11} \right\rfloor, 1\right\}, h_0 = N$	$\left\lfloor \frac{5N}{11} \right\rfloor, \left\lfloor \frac{5}{11}\left\lfloor \frac{5N}{11} \right\rfloor \right\rfloor, \ldots, 1$	Unknown	Gonnet & Baeza-Yates, 1991[13]
A108870	$\left\lceil \frac{1}{5}\left(9 \cdot \left(\frac{9}{4}\right)^{k-1} - 4\right) \right\rceil$	$1, 4, 9, 20, 46, 103, \ldots$	Unknown	Tokuda, 1992[14]
A102549	Unknown (experimentally derived)	$1, 4, 10, 23, 57, 132, 301, 701$	Unknown	Ciura, 2001[15]