Q1:

* A is an ordered integer array with 10 elements from small to large:
let’s assume A as:

1	2	3	4	5	6	7	8	9	10

gap = 5

Shell Sort Algorithm

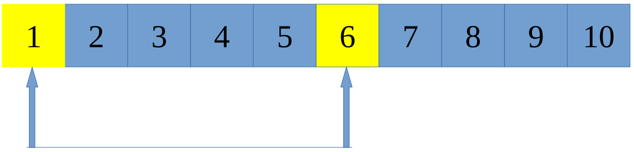
1. Set the initial value of gap to n / 2
2. while gap > 0
3. for each array element from position gap to the last element
4. Insert this element where it belongs in its subarray.
5. if gap is 2, set it to 1
6. else gap = gap / 2 // chosen by experimentation

Refinement of Step 4, the Insertion Step

4.1 nextPos is the position of the element to insert
4.2 Save the value of the element to insert in nextVal
4.3 while nextPos > gap and the element at nextPos – gap > nextVal
4.4 Shift the element at nextPos – gap to position nextPos
4.5 Decrement nextPos by gap
4.6 Insert nextVal at nextPos

Sort subarray 1

GAP	**5**

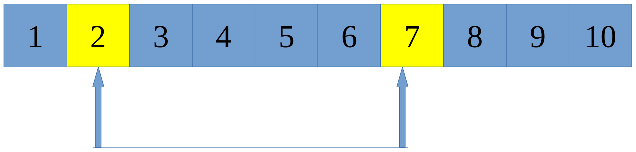


1	2	3	4	5	6	7	8	9	10

comparisons	**1**
displacements	**0**

Sort subarray 2

GAP	**5**

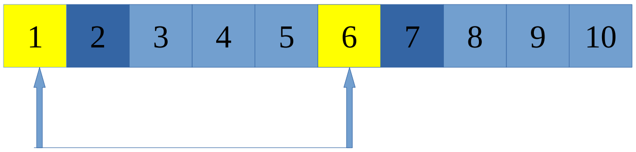


1	2	3	4	5	6	7	8	9	10

comparisons	**1**
displacements	**0**

Sort subarray 3

GAP	**5**

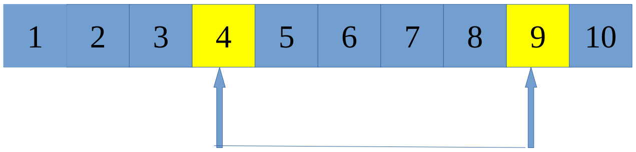


1	2	3	4	5	6	7	8	9	10

comparisons	**1**
displacements	**0**

Sort subarray 4

GAP	**5**

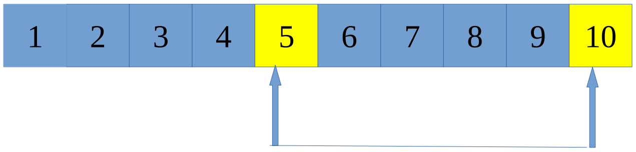


1	2	3	4	5	6	7	8	9	10

comparisons	**1**
displacements	**0**

Sort subarray 4

GAP	5



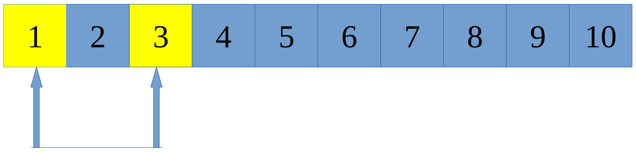
1	2	3	4	5	6	7	8	9	10

comparisons	**1**
displacements	**0**

Total Comparison with Gap Value 4	**5**
Total Displacement with Gap Value 4	**0**

Sort subarray 1

GAP	2

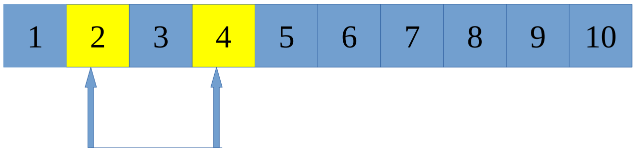


1	2	3	4	5	6	7	8	9	10

comparisons	**1**
displacements	**0**

Sort subarray 2

GAP	2

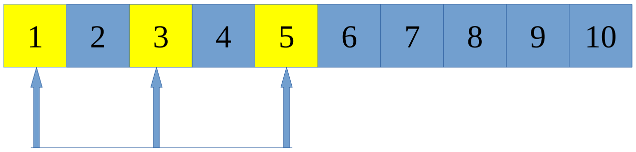


1	2	3	4	5	6	7	8	9	10

comparisons	1
displacements	0

Sort subarray 1

GAP	2

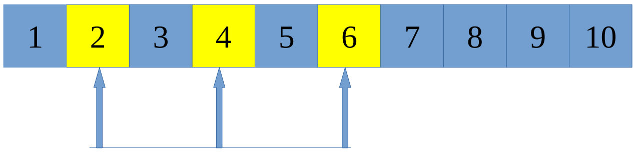


1	2	3	4	5	6	7	8	9	10

comparisons	1
displacements	0

Sort subarray 2

GAP	2

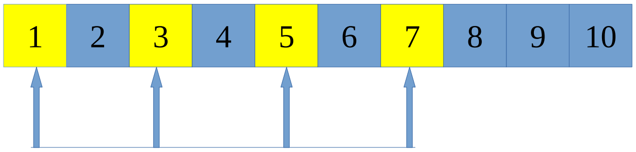


1	2	3	4	5	6	7	8	9	10

comparisons	1
displacements	0

Sort subarray 1

GAP	2

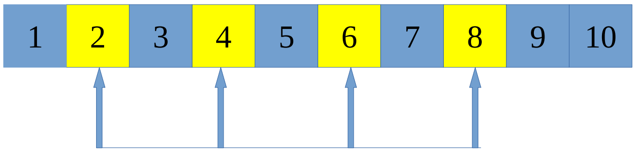


1	2	3	4	5	6	7	8	9	10

comparisons	**1**
displacements	**0**

Sort subarray 2

GAP	**2**

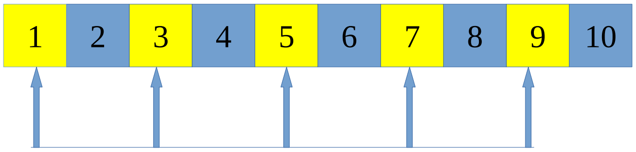


1	2	3	4	5	6	7	8	9	10

comparisons	**1**
displacements	**0**

Sort subarray 1

GAP	**2**

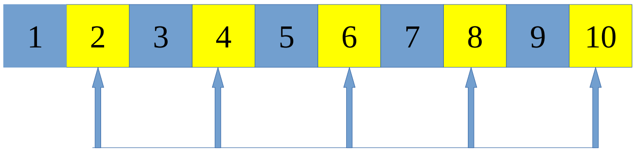


1	2	3	4	5	6	7	8	9	10

comparisons	**1**
displacements	**0**

Sort subarray 2

GAP	**2**



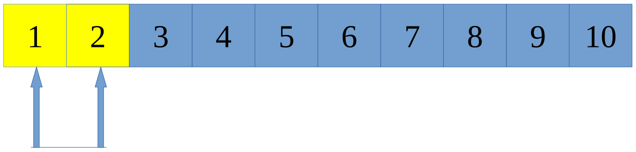
1	2	3	4	5	6	7	8	9	10

comparisons	**1**
displacements	**0**

Total Comparison with Gap Value 2	**8**
Total Displacement with Gap Value 2	**0**

Regular insertion sort

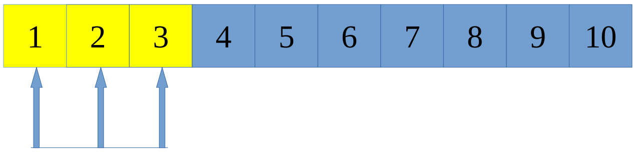
GAP	1



comparisons	**1**
displacements	**0**

Regular insertion sort

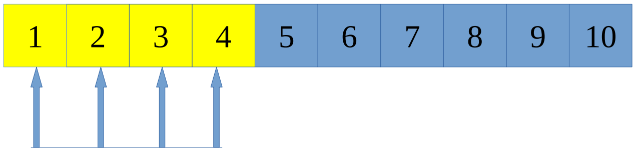
GAP	1



comparisons	**1**
displacements	**0**

Regular insertion sort

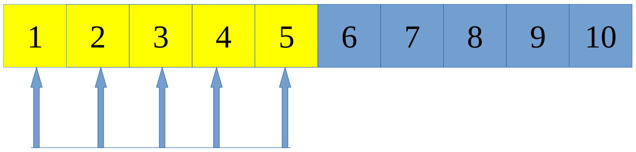
GAP	1



comparisons	**1**
displacements	**0**

Regular insertion sort

GAP	**1**

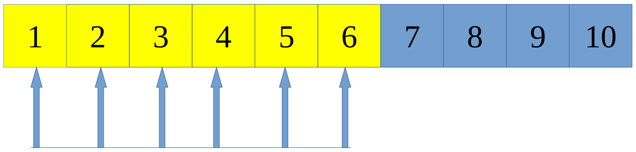


1	2	3	4	5	6	7	8	9	10

comparisons	**1**
displacements	**0**

Regular insertion sort

GAP	**1**

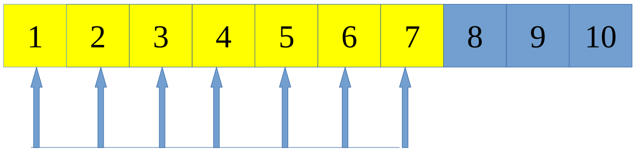


1	2	3	4	5	6	7	8	9	10

comparisons	**1**
displacements	**0**

Regular insertion sort

GAP	**1**

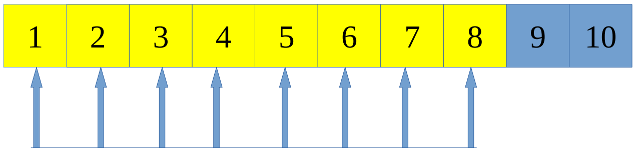


1	2	3	4	5	6	7	8	9	10

comparisons	**1**
displacements	**0**

Regular insertion sort

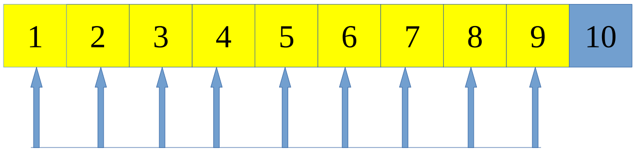
GAP	1



comparisons	1
displacements	0

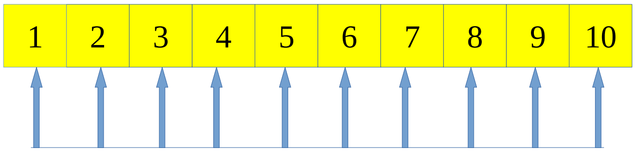
Regular insertion sort

GAP	1



comparisons	1
displacements	0

GAP	1



comparisons	1
displacements	0

Total Comparison with Gap Value 1	**9**
Total Displacement with Gap Value 1	**0**

Total comparison for A is 23 and Total Displacement for A is 0.

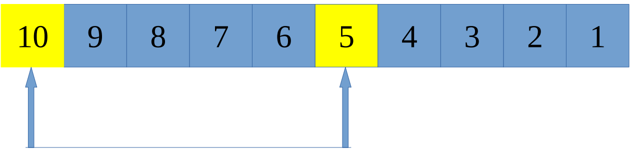
* B is an ordered integer array with 10 elements from large to small

let’s assume B as:

10	9	8	7	6	5	4	3	2	1

Sort subarray 1

GAP	**5**

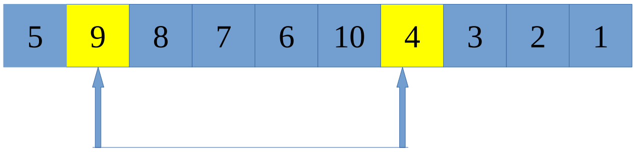


10	9	8	7	6	5	4	3	2	1

comparisons	**1**
displacements	**1**

Sort subarray 2

GAP	**5**

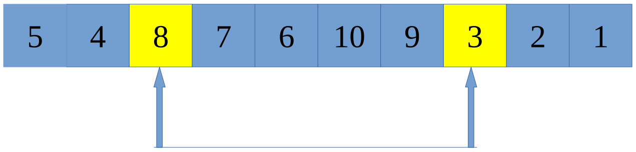


5	9	8	7	6	10	4	3	2	1

comparisons	**1**
displacements	**1**

Sort subarray 3

GAP	5

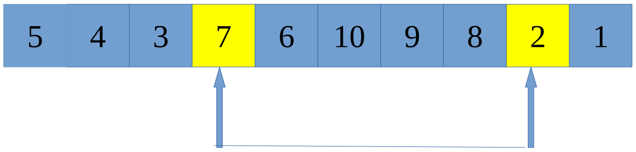


5	4	8	7	6	10	9	3	2	1

comparisons	**1**
displacements	**1**

Sort subarray 4

GAP	5

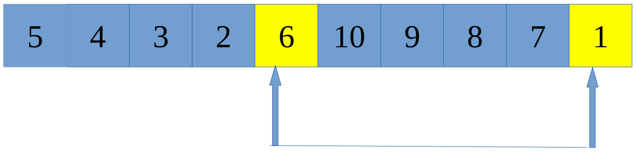


5	4	3	7	6	10	9	8	2	1

comparisons	**1**
displacements	**1**

Sort subarray 4

GAP	5



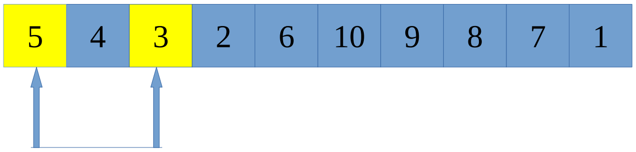
5	4	3	2	6	10	9	8	7	1

comparisons	**1**
displacements	**1**

Total Comparison with Gap Value 4	**5**
Total Displacement with Gap Value 4	**5**

Sort subarray 1

GAP	**2**

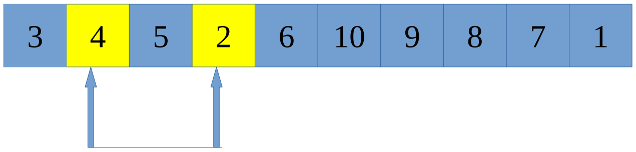


5	4	3	2	6	10	9	8	7	1

comparisons	**1**
displacements	**1**

Sort subarray 2

GAP	**2**

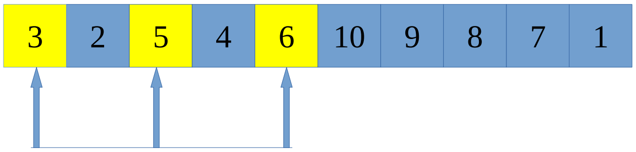


3	4	5	2	6	10	9	8	7	1

comparisons	**1**
displacements	**1**

Sort subarray 1

GAP	**2**

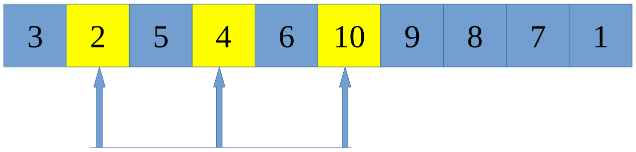


3	2	5	4	6	10	9	8	7	1

comparisons	**1**
displacements	**0**

Sort subarray 2

GAP	2

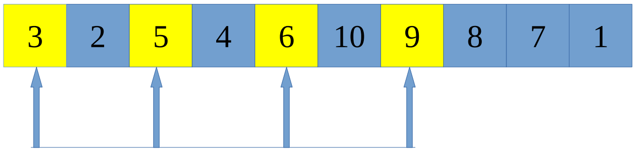


3	2	5	4	6	10	9	8	7	1

comparisons	1
displacements	0

Sort subarray 1

GAP	2

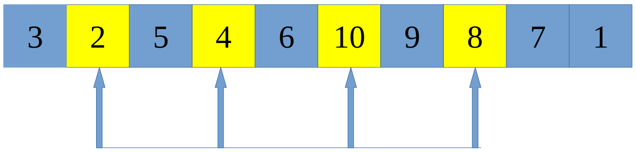


3	2	5	4	6	10	9	8	7	1

comparisons	1
displacements	0

Sort subarray 2

GAP	2

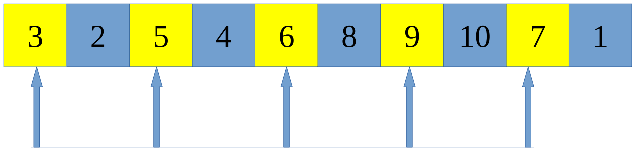


3	2	5	4	6	10	9	8	7	1

comparisons	2
displacements	1

Sort subarray 1

GAP	2

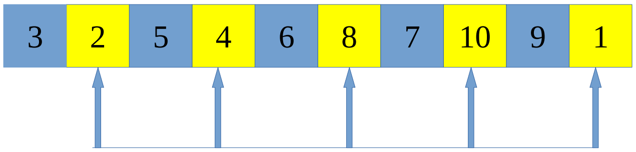


3	2	5	4	6	8	9	10	7	1

comparisons	**2**
displacements	**1**

Sort subarray 2

GAP	2



3	2	5	4	6	8	7	10	9	1

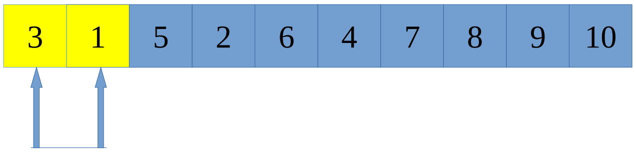
comparisons	**4**
displacements	**3**

3	1	5	2	6	4	7	8	9	10

Total Comparison with Gap Value 2	**13**
Total Displacement with Gap Value 2	**7**

Regular insertion sort

GAP	**1**

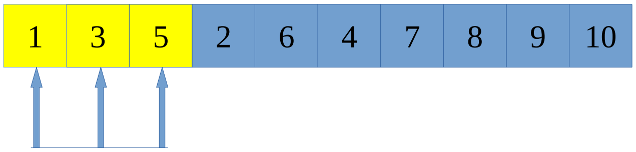


3	1	5	2	6	4	7	8	9	10

comparisons	**1**
displacements	**1**

Regular insertion sort

GAP	**1**

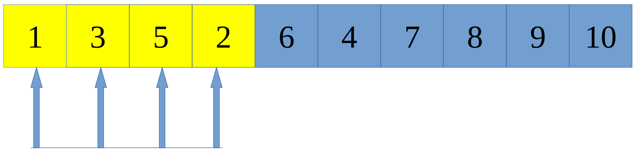


1	3	5	2	6	4	7	8	9	10

comparisons	**1**
displacements	**0**

Regular insertion sort

GAP	**1**

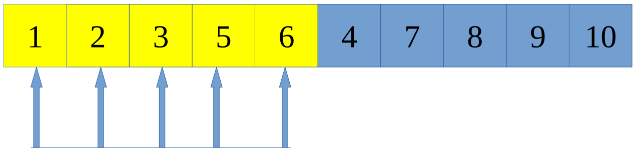


1	3	5	2	6	4	7	8	9	10

comparisons	**3**
displacements	**2**

GAP	**1**

Regular insertion sort

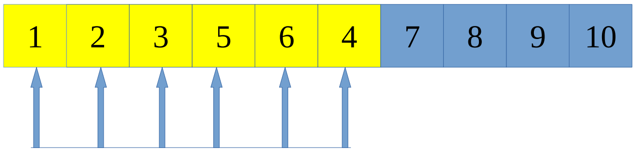


1	2	3	5	6	4	7	8	9	10

comparisons	**1**
displacements	**0**

GAP	**1**

Regular insertion sort

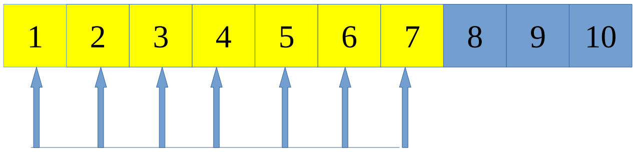


1	2	3	5	6	4	7	8	9	10

comparisons	**3**
displacements	**2**

GAP	**1**

Regular insertion sort

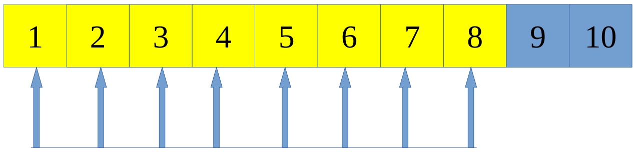


1	2	3	4	5	6	7	8	9	10

comparisons	**1**
displacements	**0**

Regular insertion sort

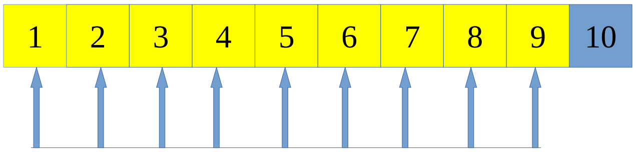
GAP	**1**



comparisons	**1**
displacements	**0**

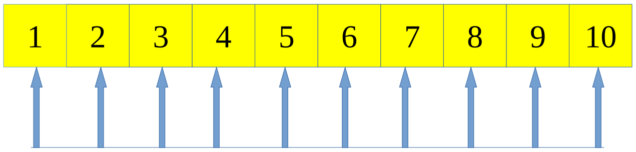
Regular insertion sort

GAP	**1**



comparisons	**1**
displacements	**0**

GAP	**1**



comparisons	**1**
displacements	**0**

Total Comparison with Gap Value 1	**13**
Total Displacement with Gap Value 1	**5**

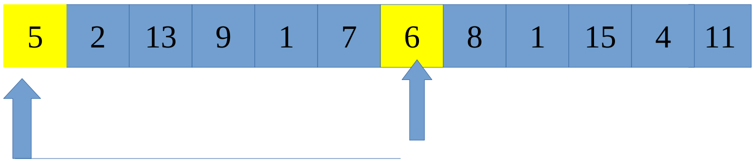
Total comparison for A is 31 and Total Displacement for A is 17.

* C = {5, 2, 13, 9, 1, 7, 6, 8, 1, 15, 4, 11}
let’s assume C as:

5	2	13	9	1	7	6	8	1	15	4	11

Sort subarray 1

GAP	**6**

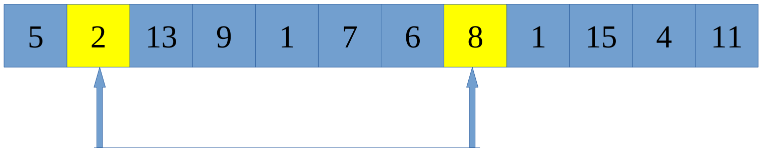


5	2	13	9	1	7	6	8	1	15	4	11

comparisons	**1**
displacements	**0**

Sort subarray 2

GAP	**6**

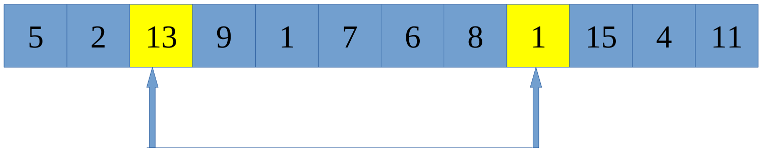


5	2	13	9	1	7	6	8	1	15	4	11

comparisons	**1**
displacements	**0**

Sort subarray 3

GAP	**6**

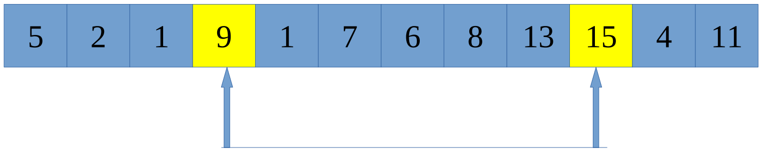


5	2	13	9	1	7	6	8	1	15	4	11

comparisons	**1**
displacements	**1**

Sort subarray 4

GAP	**6**

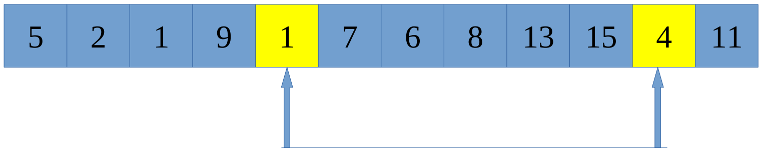


5	2	1	9	1	7	6	8	13	15	4	11

comparisons	**1**
displacements	**0**

Sort subarray 5

GAP	**6**

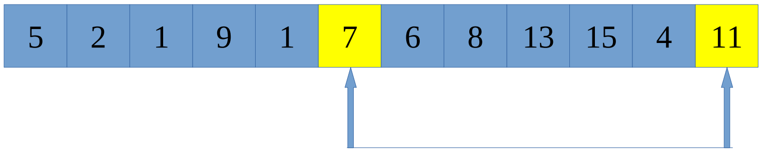


5	2	1	9	1	7	6	8	13	15	4	11

comparisons	**1**
displacements	**0**

Sort subarray 6

GAP	6



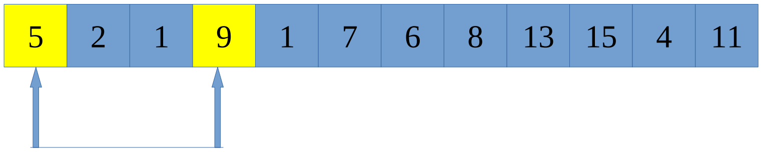
5	2	1	9	1	7	6	8	13	15	4	11

comparisons	1
displacements	0

Total Comparison with Gap Value 6	6
Total Displacement with Gap Value 6	1

Sort subarray 1

GAP	3

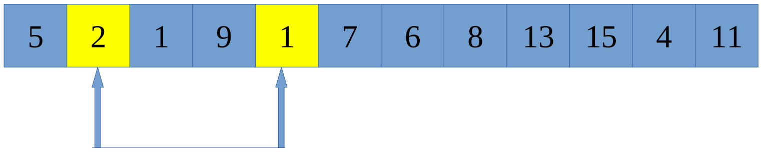


5	2	1	9	1	7	6	8	13	15	4	11

comparisons	**1**
displacements	**0**

Sort subarray 2

GAP	3

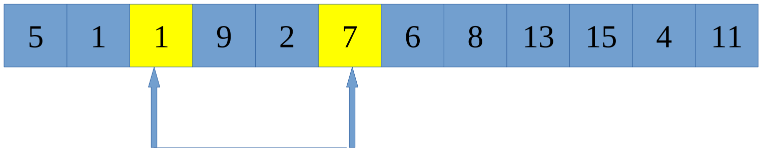


5	2	1	9	1	7	6	8	13	15	4	11

comparisons	**1**
displacements	**1**

Sort subarray 3

GAP	3

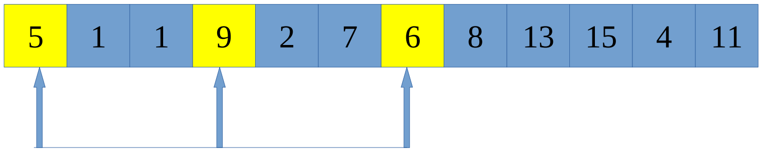


5	1	1	9	2	7	6	8	13	15	4	11

comparisons	**1**
displacements	**0**

Sort subarray 1

GAP	3

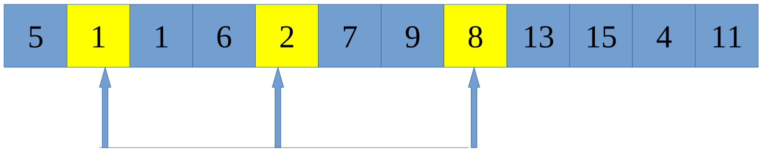


5	1	1	9	2	7	6	8	13	15	4	11

comparisons	2
displacements	1

Sort subarray 2

GAP	3

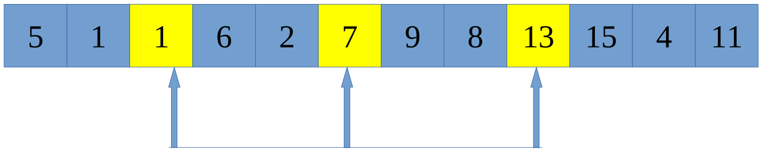


5	1	1	6	2	7	9	8	13	15	4	11

comparisons	1
displacements	0

Sort subarray 3

GAP	3

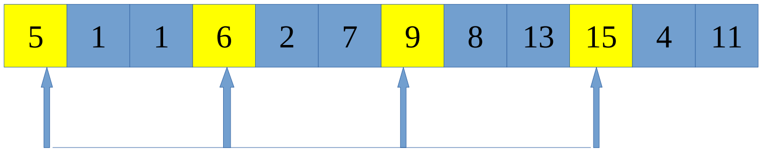


5	1	1	6	2	7	9	8	13	15	4	11

comparisons	1
displacements	0

Sort subarray 1

GAP	3

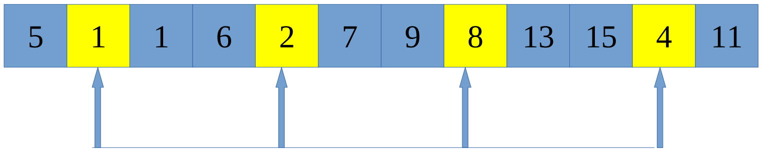


5	1	1	6	2	7	9	8	13	15	4	11

comparisons	1
displacements	0

Sort subarray 2

GAP	3

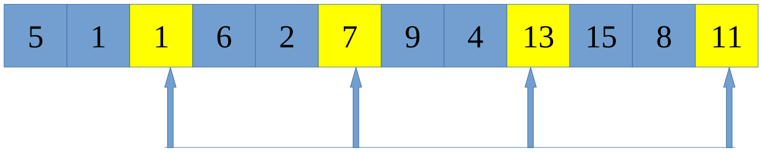


5	1	1	6	2	7	9	8	13	15	4	11

comparisons	2
displacements	1

Sort subarray 3

GAP	3



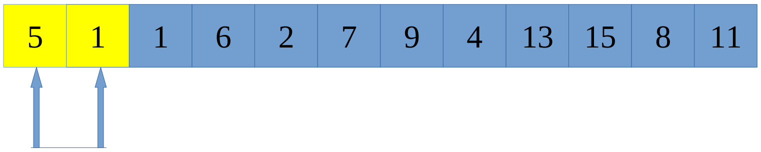
5	1	1	6	2	7	9	4	13	15	8	11

comparisons	2
displacements	1

Total Comparison with Gap Value 3	12
Total Displacement with Gap Value 3	3

Regular insertion sort

GAP	**1**

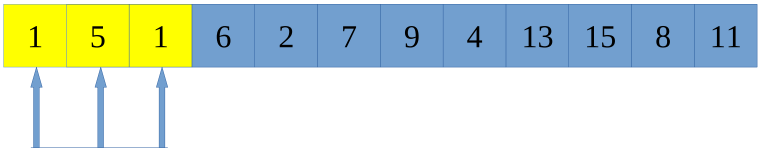


5	1	1	6	2	7	9	4	13	15	8	11

comparisons	**1**
displacements	**1**

Regular insertion sort

GAP	**1**

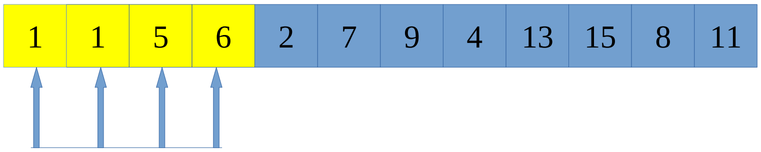


1	5	1	6	2	7	9	4	13	15	8	11

comparisons	**2**
displacements	**1**

Regular insertion sort

GAP	**1**

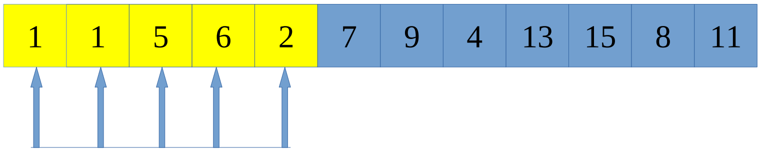


1	1	5	6	2	7	9	4	13	15	8	11

comparisons	**1**
displacements	**0**

Regular insertion sort

GAP	**1**

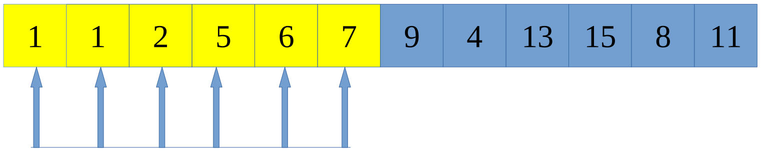


1	1	5	6	2	7	9	4	13	15	8	11

comparisons	**3**
displacements	**2**

Regular insertion sort

GAP	**1**

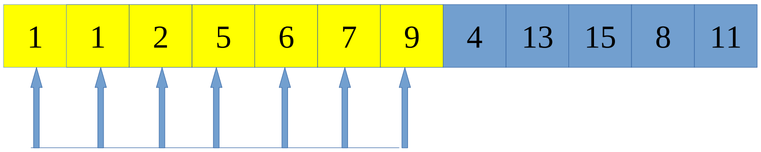


1	1	2	5	6	7	9	4	13	15	8	11

comparisons	**1**
displacements	**0**

Regular insertion sort

GAP	**1**

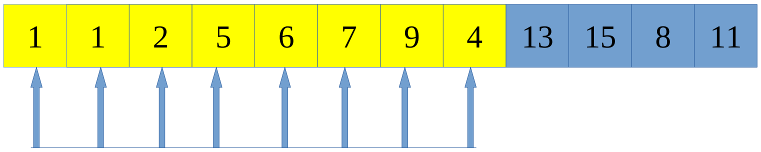


1	1	2	5	6	7	9	4	13	15	8	11

comparisons	**1**
displacements	**0**

Regular insertion sort

GAP	1

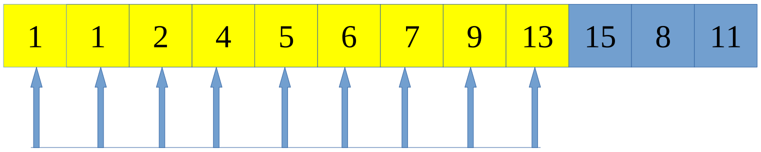


1	1	2	5	6	7	9	4	13	15	8	11

comparisons	5
displacements	4

Regular insertion sort

GAP	1

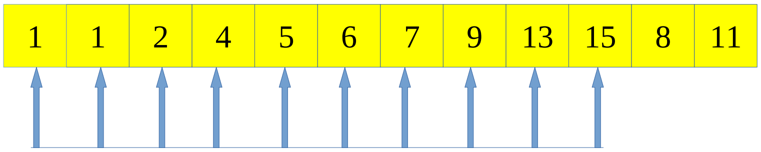


1	1	2	4	5	6	7	9	13	15	8	11

comparisons	1
displacements	0

GAP	1

Regular insertion sort

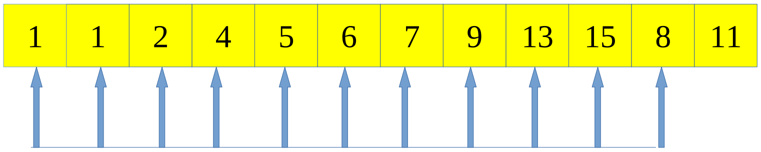


1	1	2	4	5	6	7	9	13	15	8	11

comparisons	1
displacements	0

GAP	1

Regular insertion sort

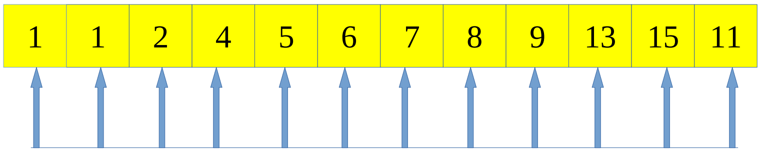


1	1	2	4	5	6	7	9	13	15	8	11

comparisons	**4**
displacements	**3**

GAP	1

Regular insertion sort



1	1	2	4	5	6	7	8	9	13	15	11

comparisons	**3**
displacements	**2**

1	1	2	4	5	6	7	8	9	11	13	15

Total Comparison with Gap Value 1	**23**
Total Displacement with Gap Value 1	**12**

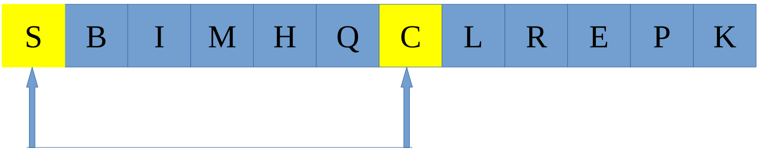
Total comparison for C is 41 and Total Displacement for C is 16.

* D = {‘S’, ‘B’, ‘I’, ‘M’, ‘H’, ‘Q’, ‘C’, ‘L’, ‘R’, ‘E’, ‘P’, ‘K’}
let’s assume D as:

S	B	I	M	H	Q	C	L	R	E	P	K

Sort subarray 1

GAP	6

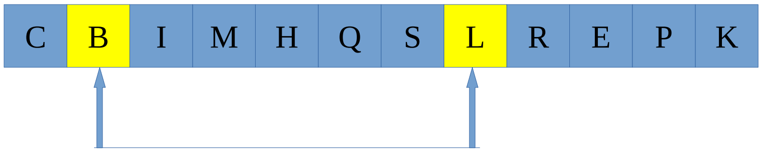


S	B	I	M	H	Q	C	L	R	E	P	K

comparisons	1
displacements	1

Sort subarray 2

GAP	6

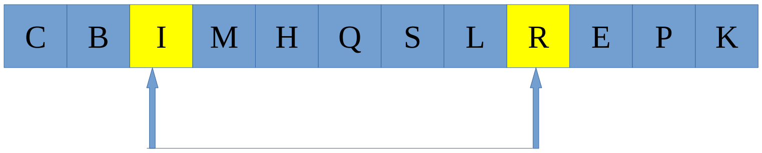


C	B	I	M	H	Q	S	L	R	E	P	K

comparisons	1
displacements	0

Sort subarray 3

GAP	6

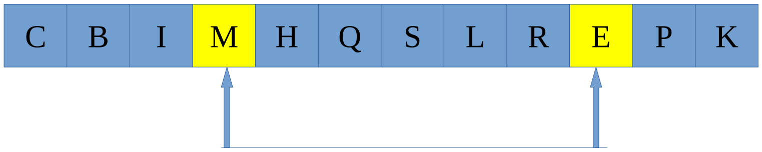


C	B	I	M	H	Q	S	L	R	E	P	K

comparisons	**1**
displacements	**0**

Sort subarray 4

GAP	**6**

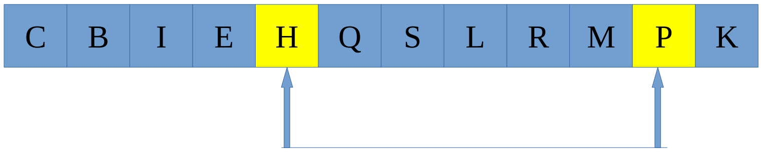


C	B	I	M	H	Q	S	L	R	E	P	K

comparisons	**1**
displacements	**1**

Sort subarray 5

GAP	**6**

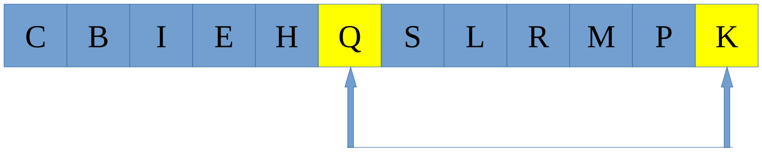


C	B	I	E	H	Q	S	L	R	M	P	K

comparisons	**1**
displacements	**0**

Sort subarray 6

GAP	**6**



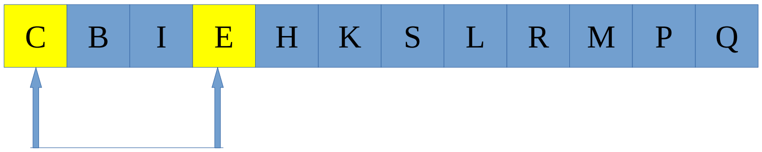
C	B	I	E	H	Q	S	L	R	M	P	K

comparisons	**1**
displacements	**1**

Total Comparison with Gap Value 6	**6**
Total Displacement with Gap Value 6	**3**

Sort subarray 1

GAP	**3**

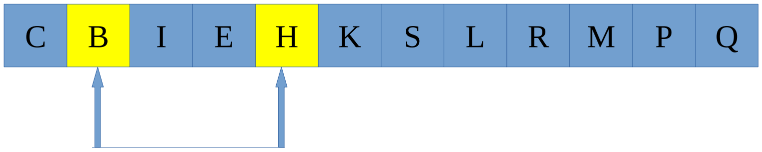


C	B	I	E	H	K	S	L	R	M	P	Q

comparisons	**1**
displacements	**0**

Sort subarray 2

GAP	**3**

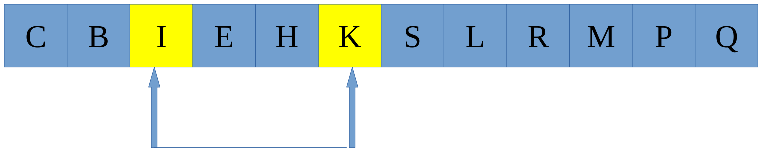


C	B	I	E	H	K	S	L	R	M	P	Q

comparisons	**1**
displacements	**0**

Sort subarray 3

GAP	**3**

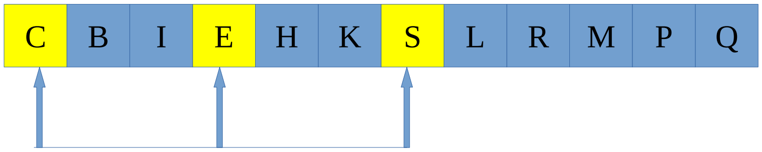


C	B	I	E	H	K	S	L	R	M	P	Q

comparisons	**1**
displacements	**0**

Sort subarray 1

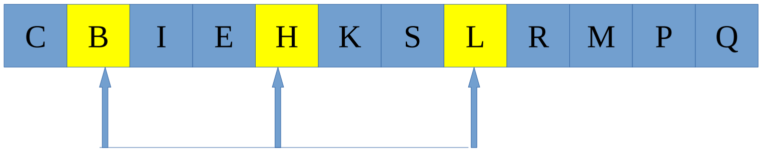
GAP	3



comparisons	1
displacements	0

Sort subarray 2

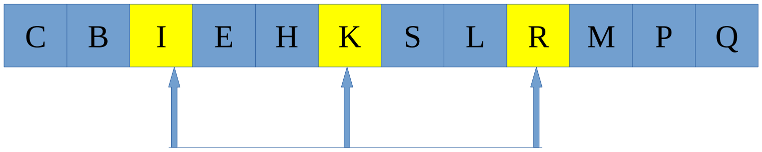
GAP	3



comparisons	1
displacements	0

Sort subarray 3

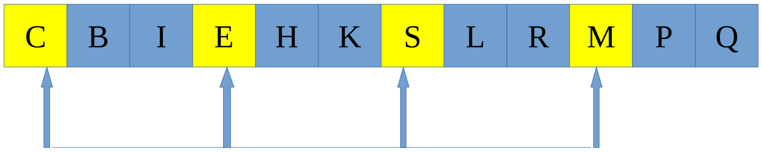
GAP	3



comparisons	1
displacements	0

Sort subarray 1

GAP	3

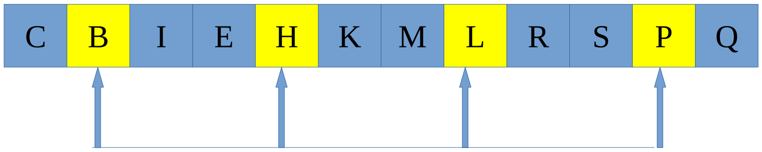


C	B	I	E	H	K	S	L	R	M	P	Q

comparisons	2
displacements	1

Sort subarray 2

GAP	3

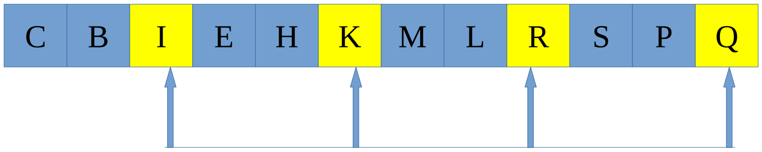


C	B	I	E	H	K	M	L	R	S	P	Q

comparisons	2
displacements	1

Sort subarray 3

GAP	3



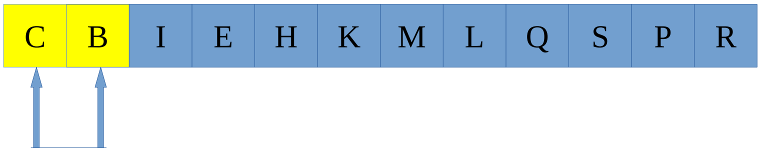
C	B	I	E	H	K	M	L	R	S	P	Q

comparisons	2
displacements	1

Total Comparison with Gap Value 3	**12**
Total Displacement with Gap Value 3	**3**

Regular insertion sort

GAP	**1**

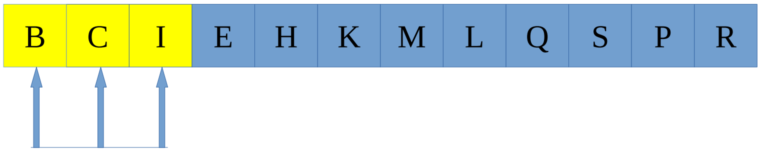


C	B	I	E	H	K	M	L	Q	S	P	R

comparisons	**1**
displacements	**1**

Regular insertion sort

GAP	**1**

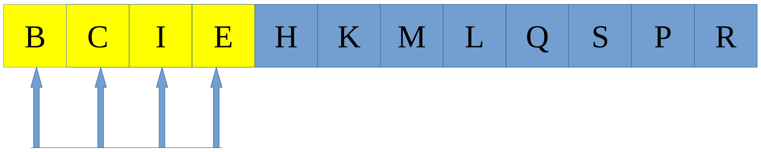


B	C	I	E	H	K	M	L	Q	S	P	R

comparisons	**1**
displacements	**0**

Regular insertion sort

GAP	**1**

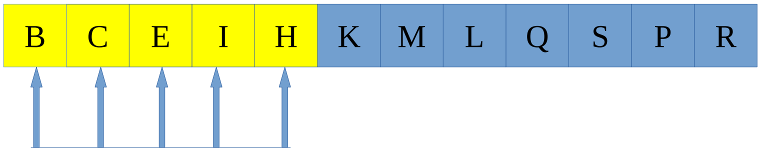


B	C	I	E	H	K	M	L	Q	S	P	R

comparisons	**2**
displacements	**1**

Regular insertion sort

GAP	**1**

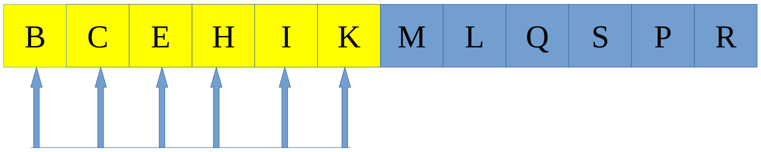


B	C	E	I	H	K	M	L	Q	S	P	R

comparisons	**2**
displacements	**1**

Regular insertion sort

GAP	**1**

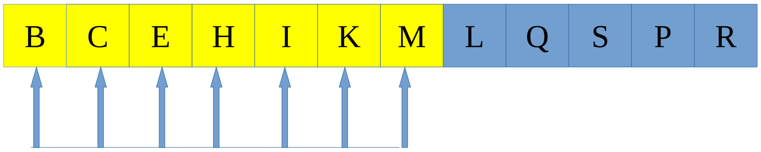


B	C	E	H	I	K	M	L	Q	S	P	R

comparisons	**1**
displacements	**0**

Regular insertion sort

GAP	**1**

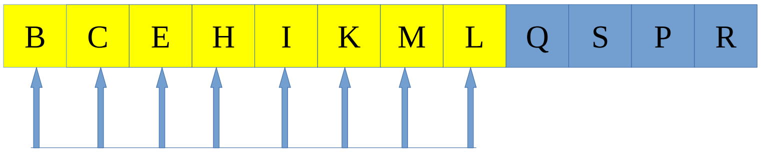


B	C	E	H	I	K	M	L	Q	S	P	R

comparisons	**1**
displacements	**0**

Regular insertion sort

GAP	**1**

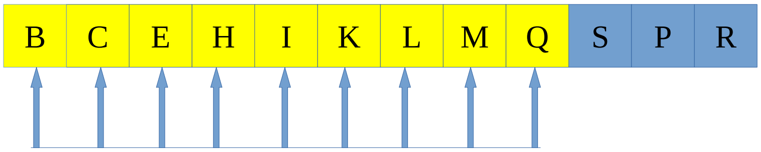


B	C	E	H	I	K	M	L	Q	S	P	R

comparisons	**2**
displacements	**1**

Regular insertion sort

GAP	**1**

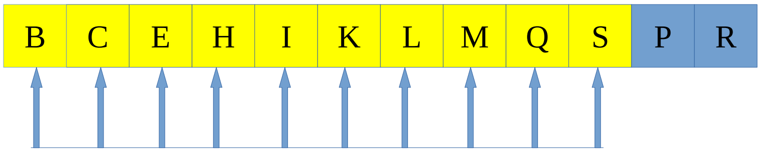


B	C	E	H	I	K	L	M	Q	S	P	R

comparisons	**1**
displacements	**0**

Regular insertion sort

GAP	**1**



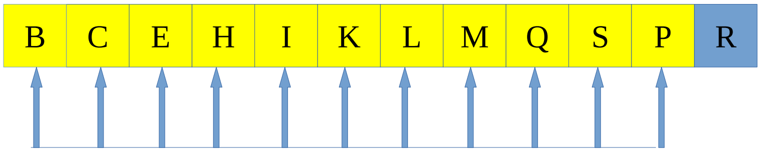
B	C	E	H	I	K	L	M	Q	S	P	R

comparisons	**1**

displacements	**0**

Regular insertion sort

GAP	**1**

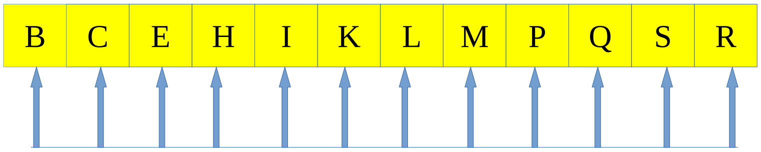


B	C	E	H	I	K	L	M	Q	S	P	R

comparisons	**3**
displacements	**2**

Regular insertion sort

GAP	**1**



B	C	E	H	I	K	L	M	P	Q	S	R

comparisons	**2**
displacements	**1**

B	C	E	H	I	K	L	M	P	Q	R	S

Total Comparison with Gap Value 1	**17**
Total Displacement with Gap Value 1	**7**

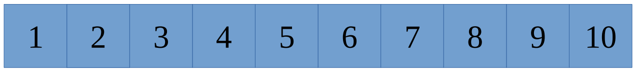
Total comparison for D is 35 and Total Displacement for D is 13.

* A is an ordered integer array with 10 elements from small to large

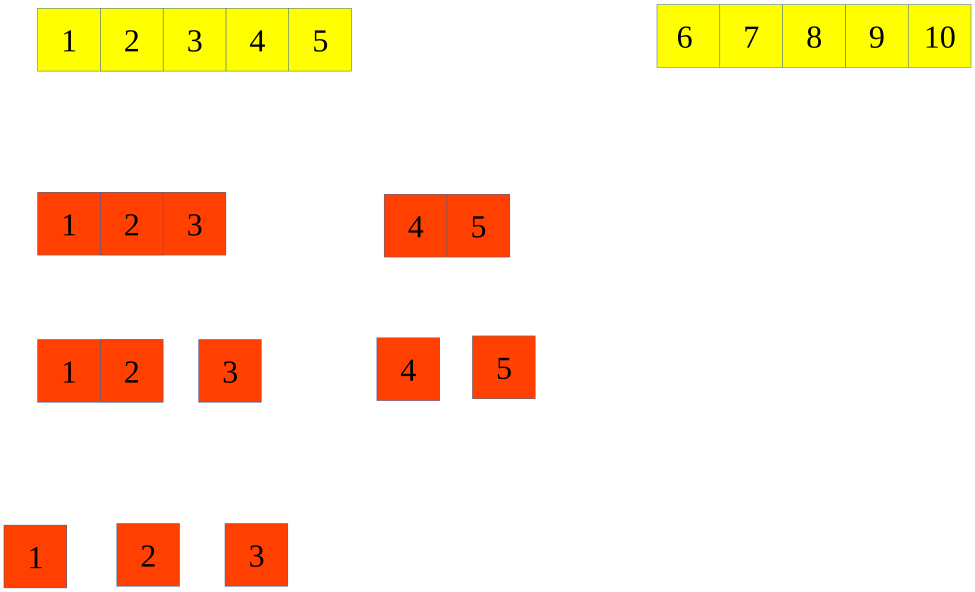
Merge Algorithm

```
1. Access the first item from both sequences.
2. while not finished with either sequence
3.        Compare the current items from the two sequences, copy the smaller
          current item to the output sequence, and access the next item from the
          input sequence whose item was copied.
4. Copy any remaining items from the first sequence to the output sequence.
5. Copy any remaining items from the second sequence to the output sequence.
```

Assume A is:

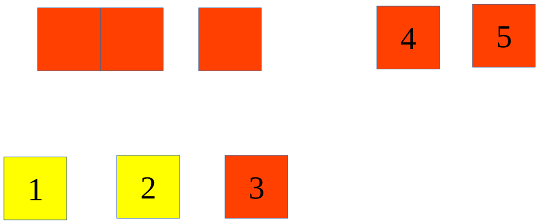


split array to two half recursively:



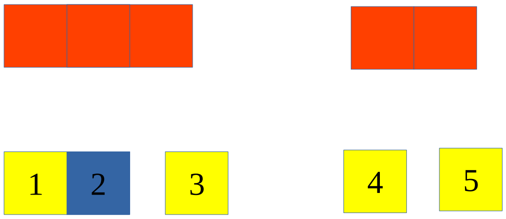
sorting left half of main array recursively…

comparison:
yellow boxes are indicated comparing two elements.

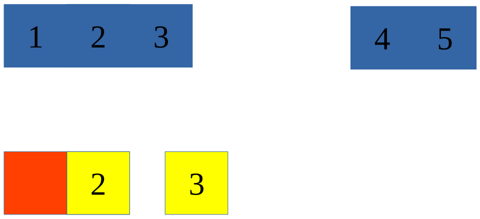


comparisons	**1**

1)



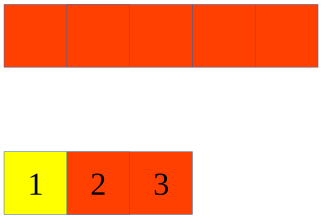
2)



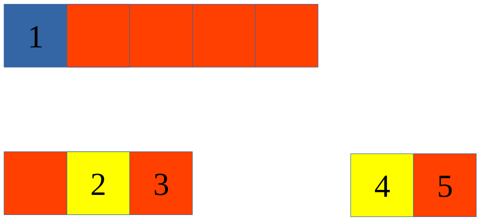
3)

comparisons	**3**

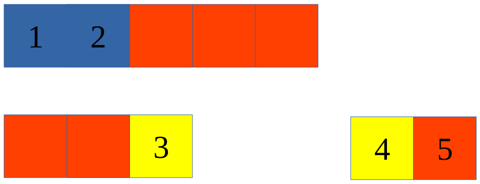
1)



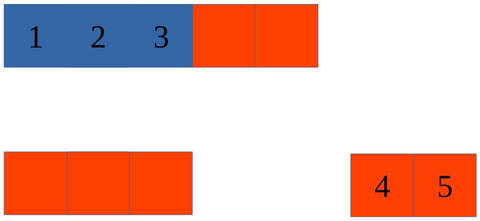
2)



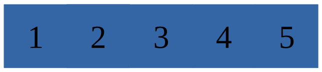
3)



4)

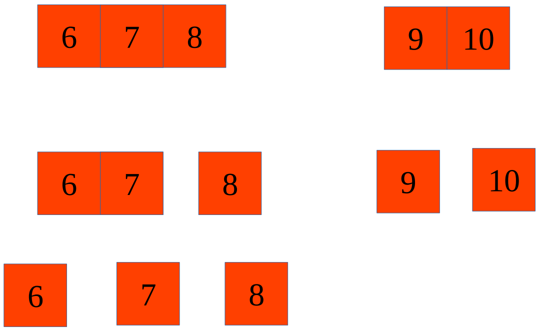


5)

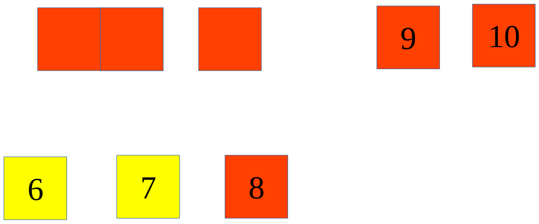


comparisons	**3**

right half of array :

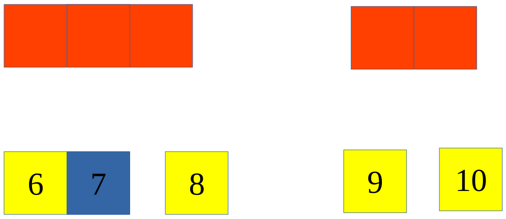


comparison:
yellow boxes are indicated comparing two elements.

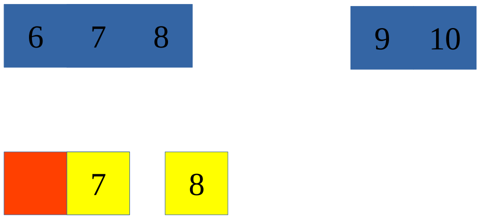


comparisons	**1**

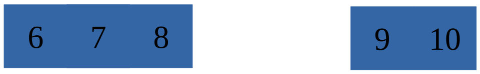
1)



2)

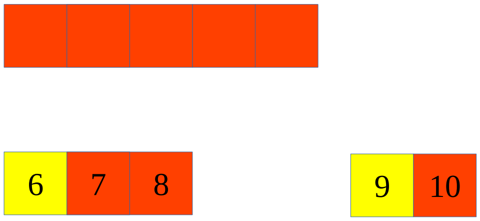


3)

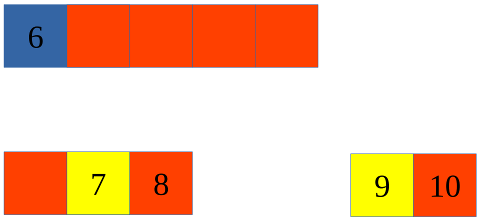


comparisons	**3**

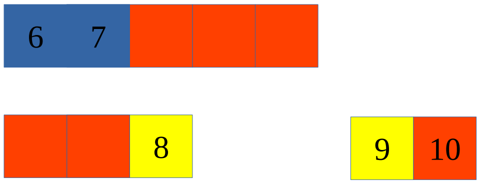
1)



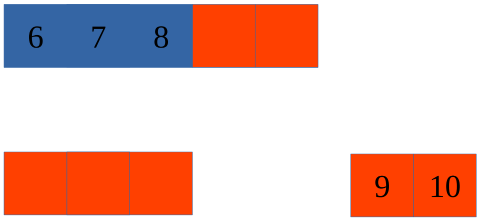
2)



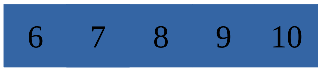
3)



4)

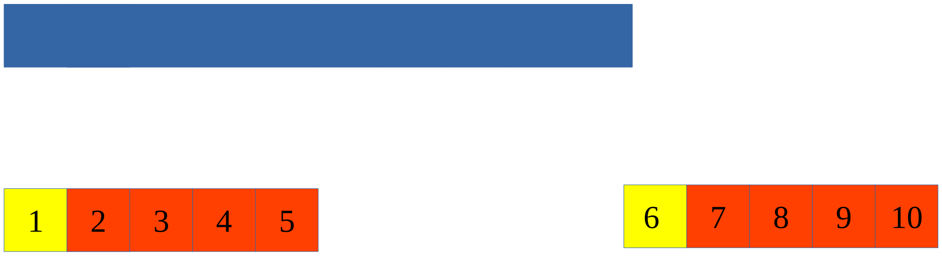


5)

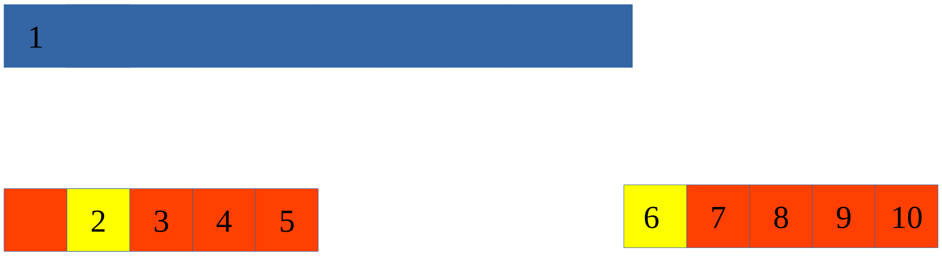


comparisons	**3**

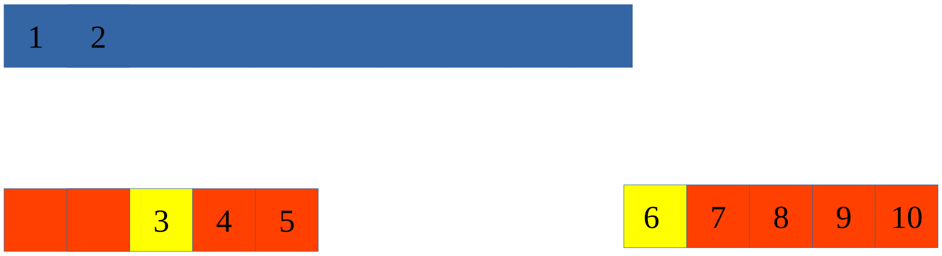
1)



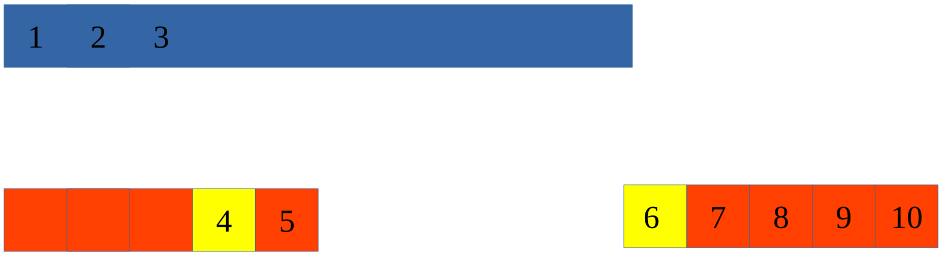
2)


1
2
3
4
5
6
7
8
9
10

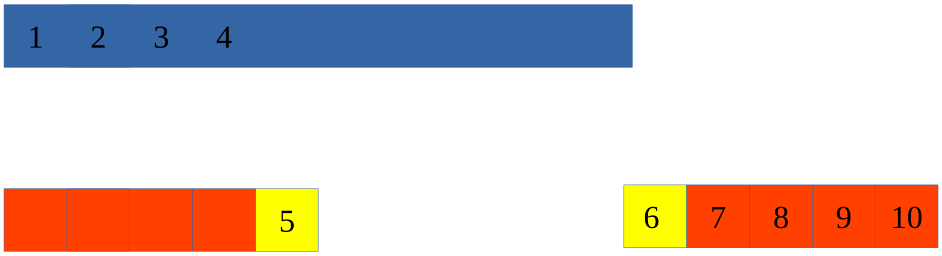
3)


1
2
3
4
5
6
7
8
9
10

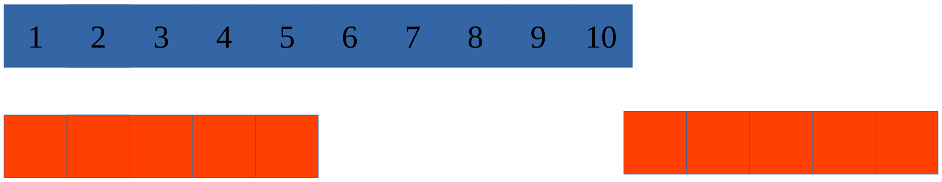
4)


1
2
3
4
5
6
7
8
9
10

5)


1
2
3
4
5
6
7
8
9
10

6)


1
2
3
4
5
6
7
8
9
10

comparisons	**5**

total compared cases are: 19 ormerge sort in array A

* B is an ordered integer array with 10 elements from large to small

Assume B is:

10	9	8	7	6	5	4	3	2	1

split array to two half recursively:

10	9	8	7	6

5	4	3	2	1

10	9	8

7	6

10	9

8

7

6

10

9

8

sorting left half of main array recursively…

comparison:
yellow boxes are indicated comparing two elements.

7

6

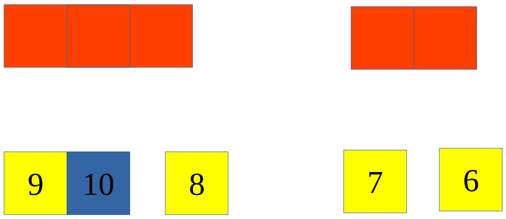
10

9

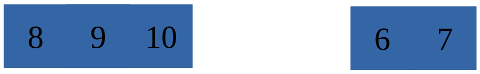
8

comparisons	**1**

1)

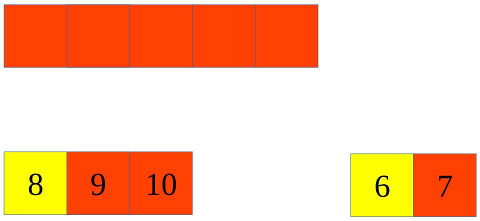


2)

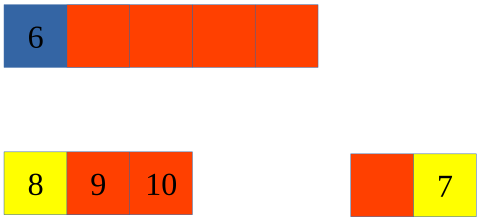


comparisons	**2**

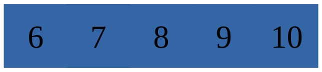
1)



2)

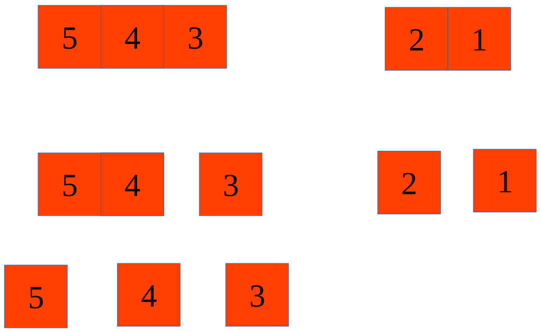


3)

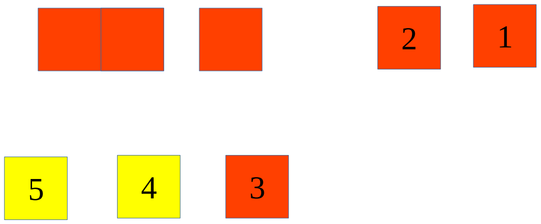


comparisons	**2**

right half of array :

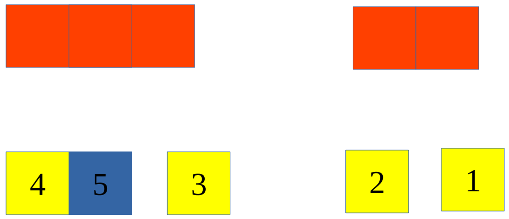


sorting right of main array recursively…
comparison:
yellow boxes are indicated comparing two elements.

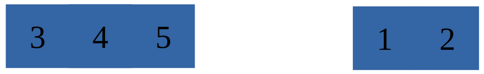


comparisons	1

1)

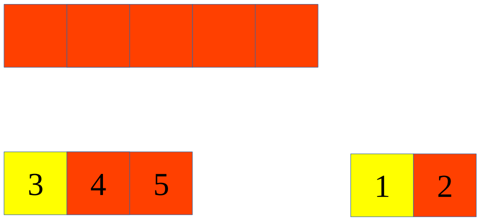


2)

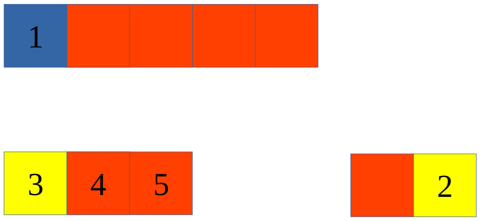


comparisons	2

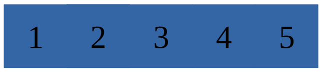
1)



2)

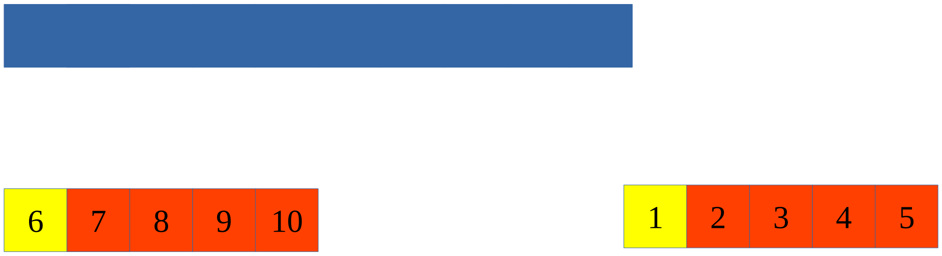


3)

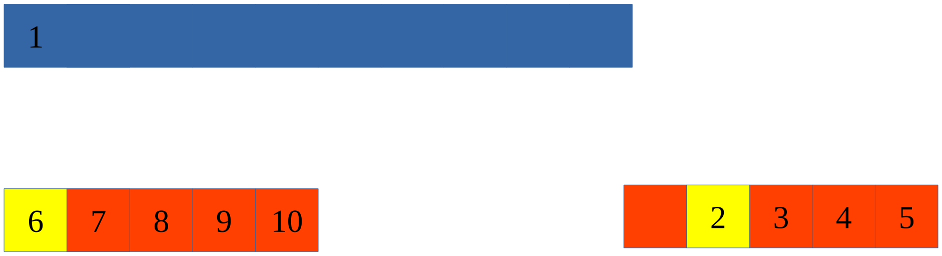


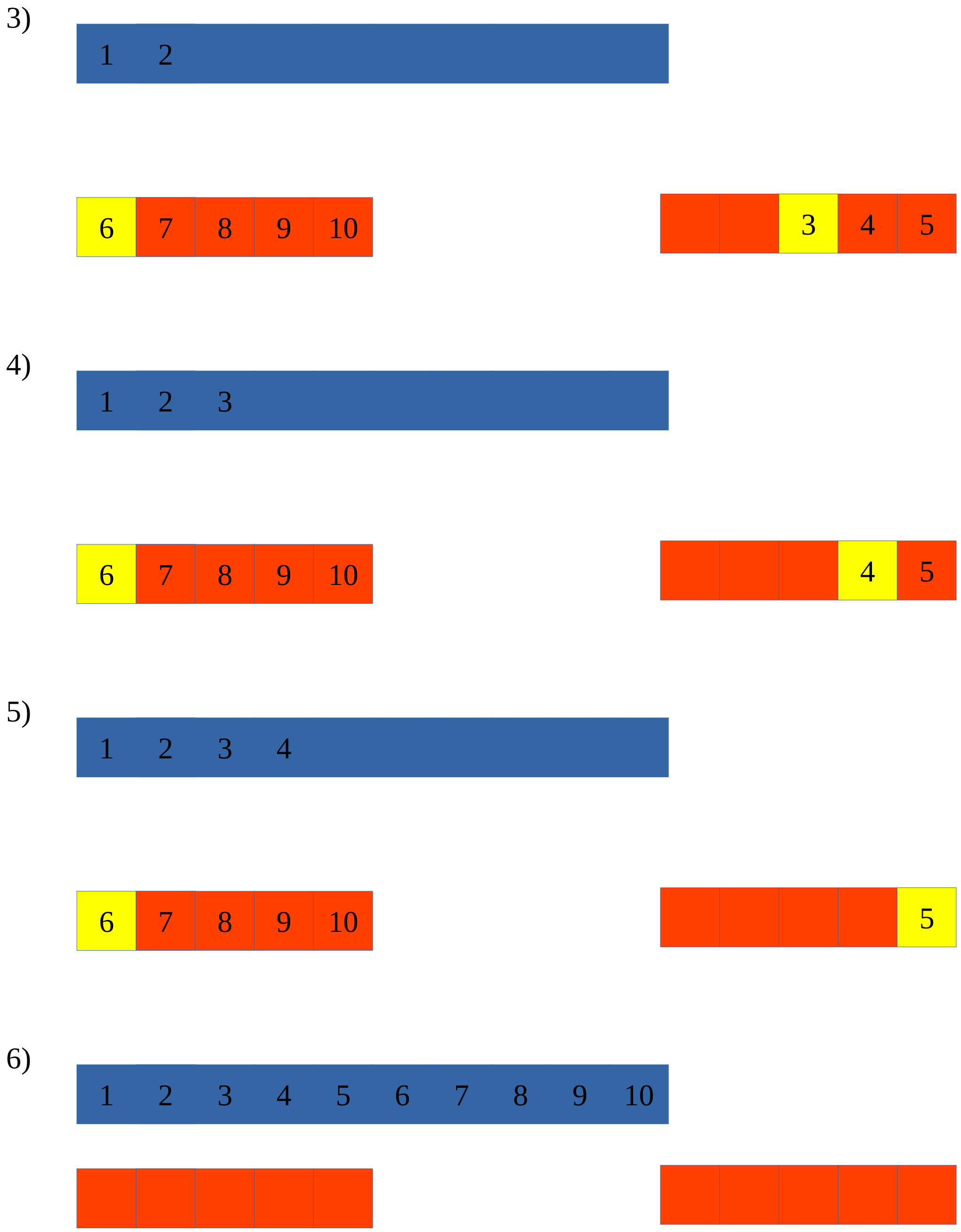
comparisons	**2**

1)



2)



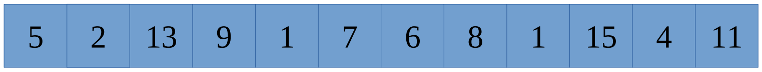


comparisons	**5**

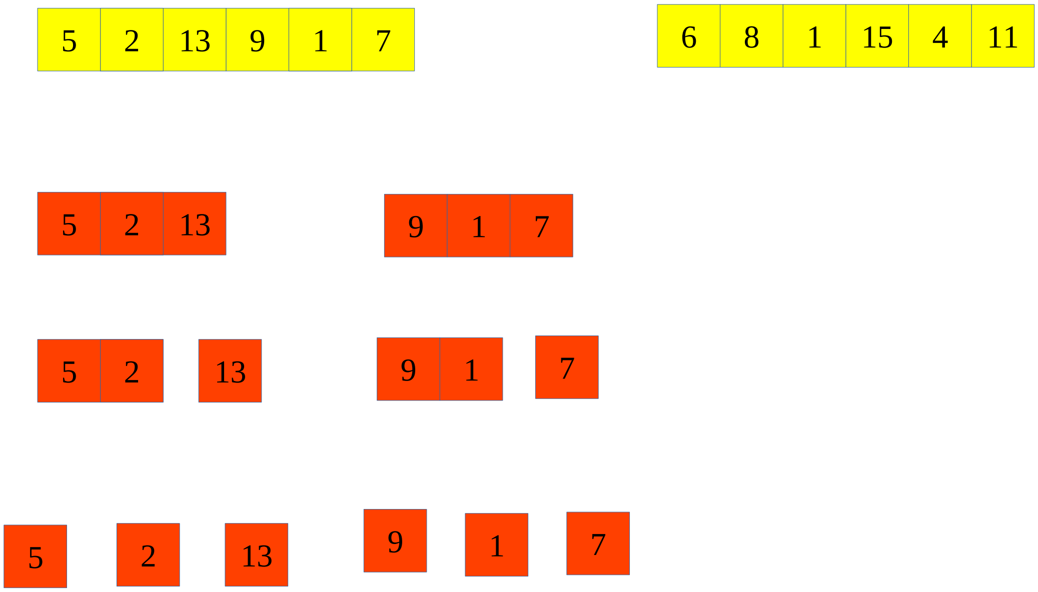
Total comparison is 15.

* C = {5, 2, 13, 9, 1, 7, 6, 8, 1, 15, 4, 11}

Assume C is:



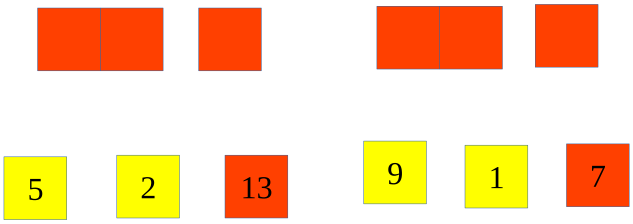
split array to two half recursively:



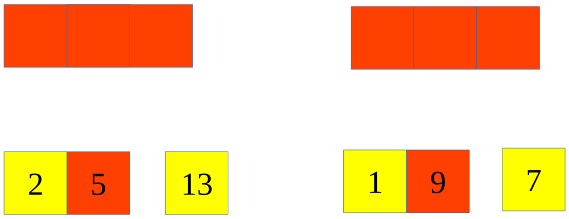
sorting left half of main array recursively…

comparison
yellow boxes are indicated comparing two elements.

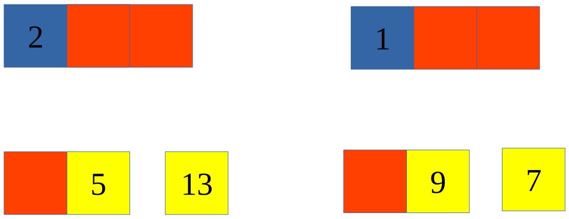
1)



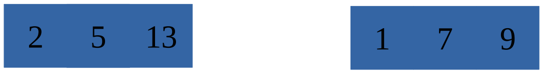
2)



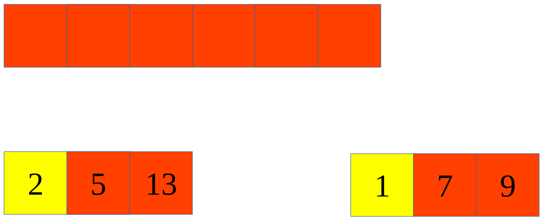
3)



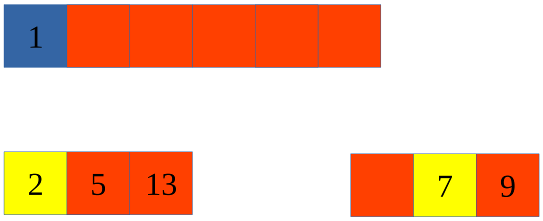
--->>>



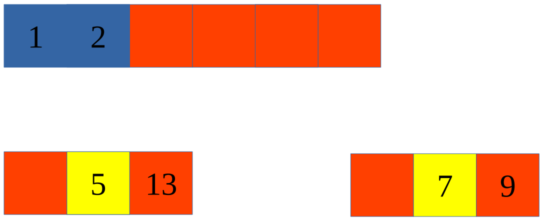
5)



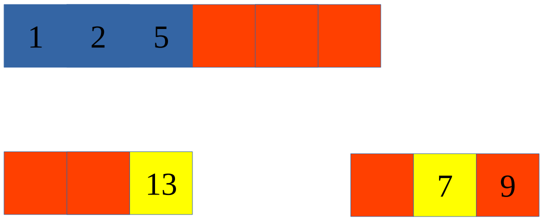
6)



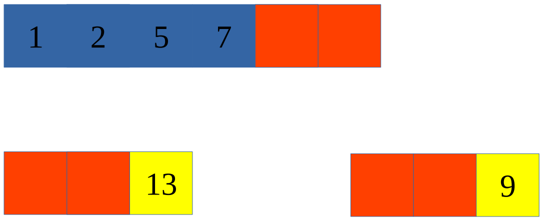
7)



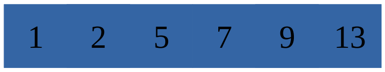
8)



9)

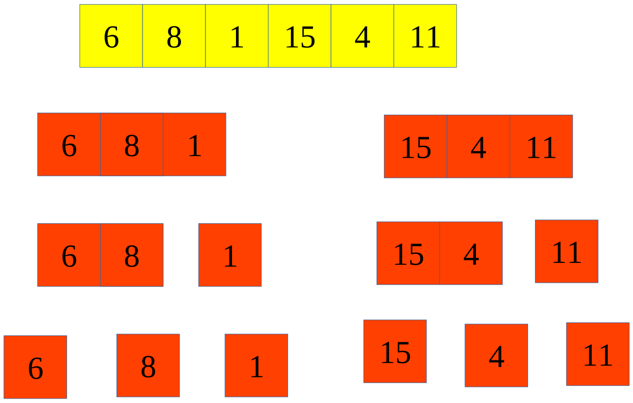


-->



Total compared	11

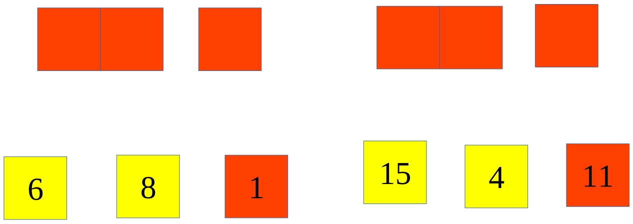
right side:



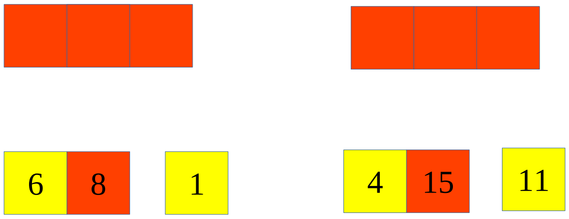
sorting left half of main array recursively…

comparison
yellow boxes are indicated comparing two elements.

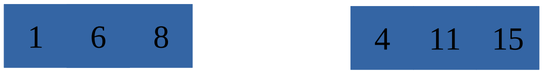
1)



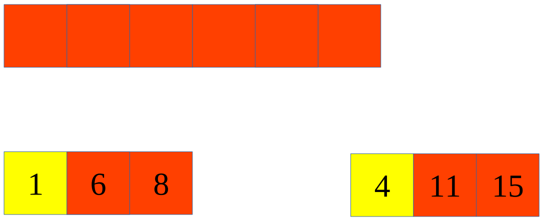
2)



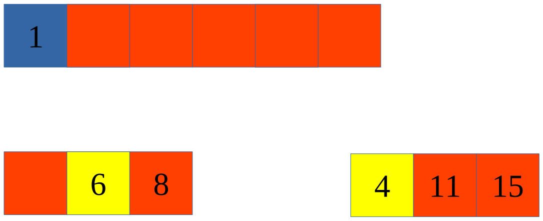
--->



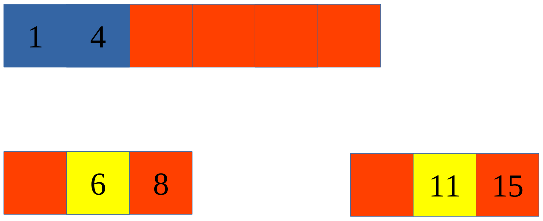
5)



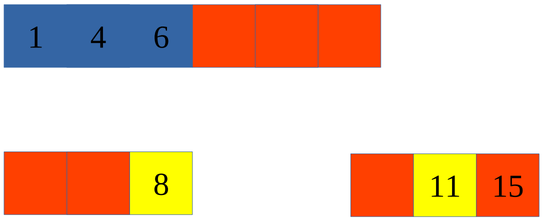
6)



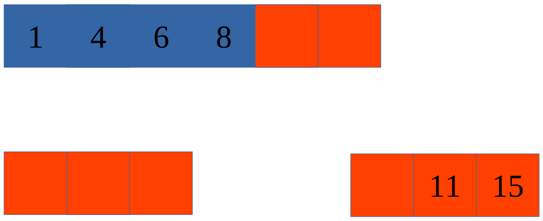
7)



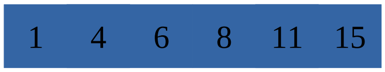
8)



9)

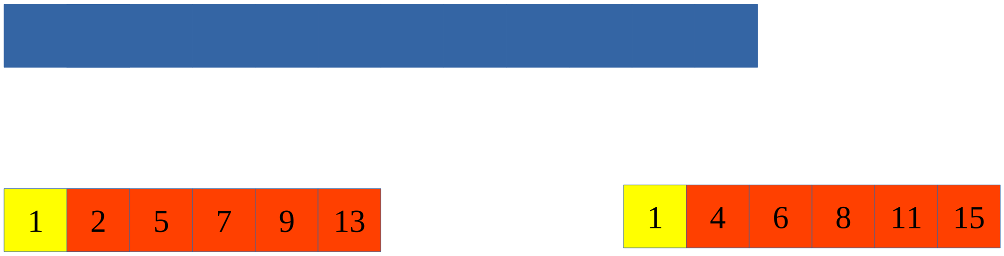


-->

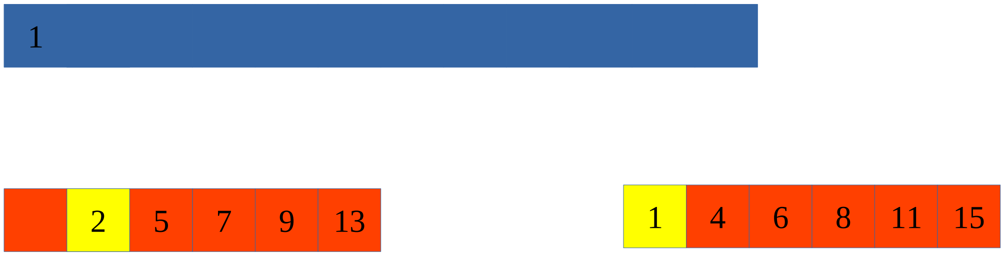


Total compared	8

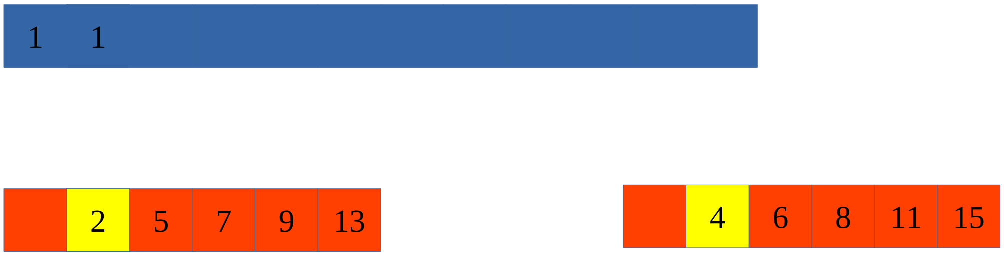
1)



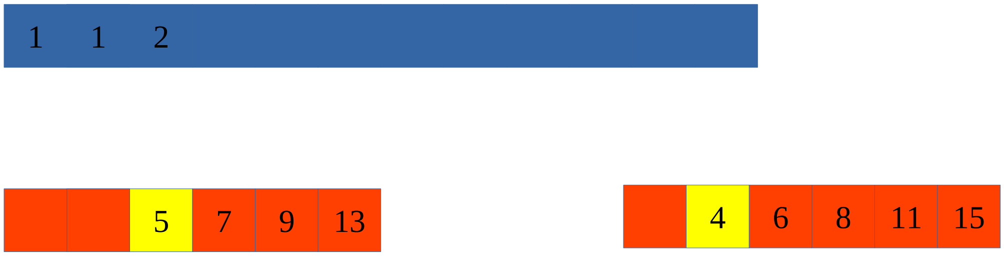
2)


1
2 5 7 9 13
1 4 6 8 11 15

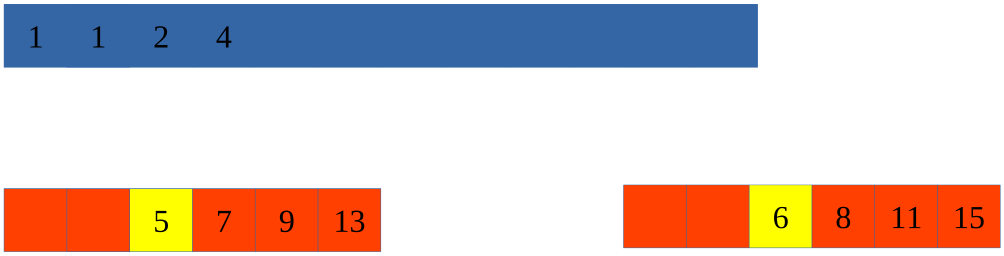
3)


1 1
2 5 7 9 13
4 6 8 11 15

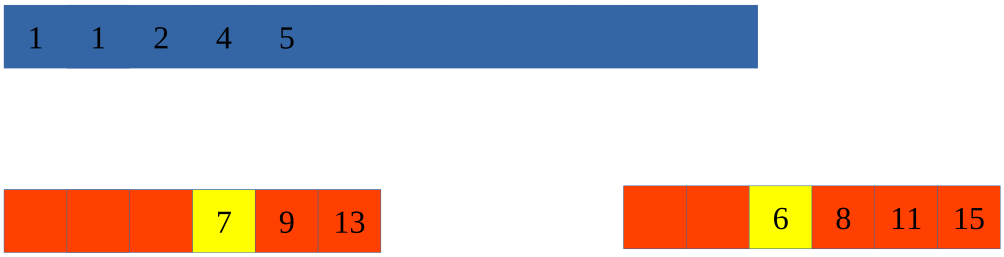
4)


1 1 2
5 7 9 13
4 6 8 11 15

5)


1 1 2 4
5 7 9 13
6 8 11 15

6)


1 1 2 4 5
7 9 13
6 8 11 15

7)

1	1	2	4	5	6						

			7	9	13

			8	11	15

8)

1	1	2	4	5	6	7					

				9	13

			8	11	15

8)

1	1	2	4	5	6	7	8				

				9	13

				11	15

9)

1	1	2	4	5	6	7	8	9			

					13

				11	15

10)

1	1	2	4	5	6	7	8	9	11		

					13

					15

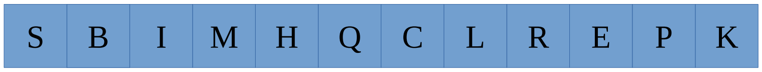
--->>>

1	1	2	4	5	6	7	8	9	11	13	15

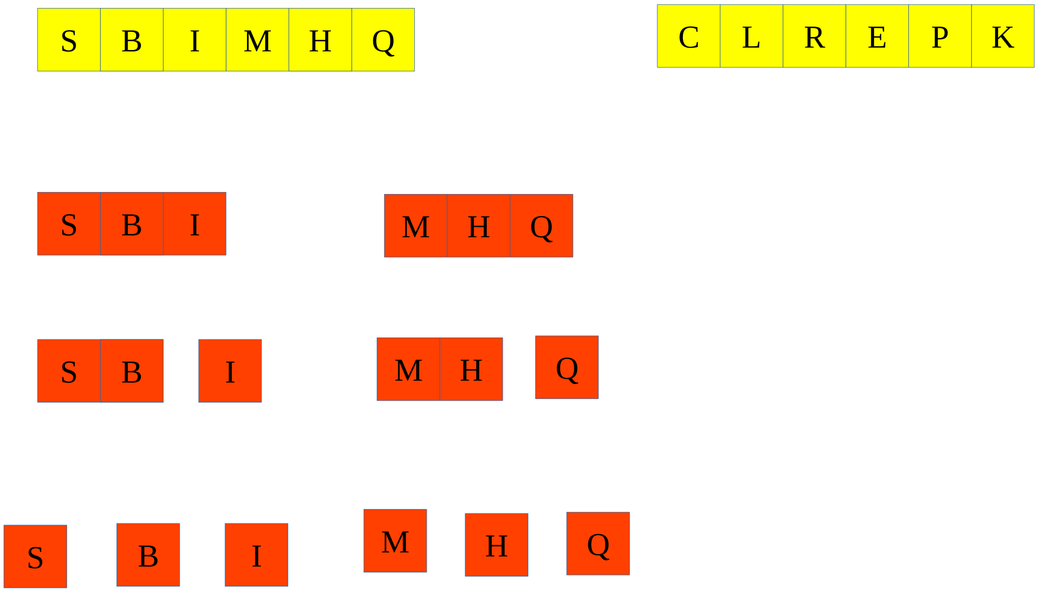
total comparisons : 30

* D = {‘S’, ‘B’, ‘I’, ‘M’, ‘H’, ‘Q’, ‘C’, ‘L’, ‘R’, ‘E’, ‘P’, ‘K’}

Assume D is:



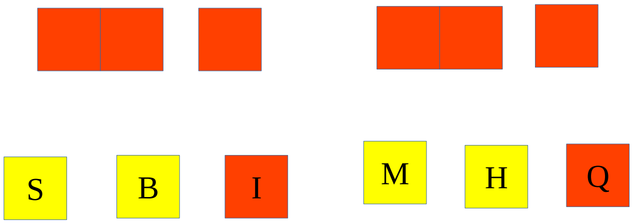
split array to two half recursively:



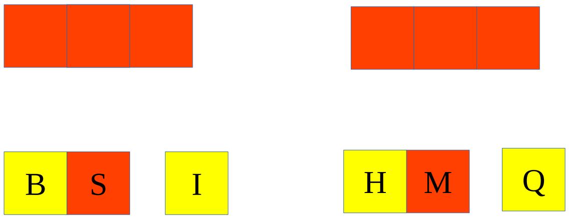
sorting left half of main array recursively…

comparison
yellow boxes are indicated comparing two elements.

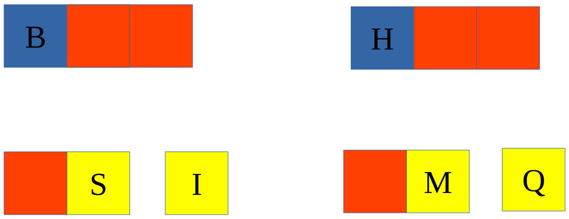
1)



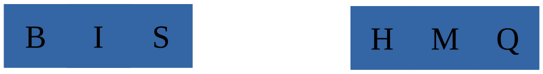
2)



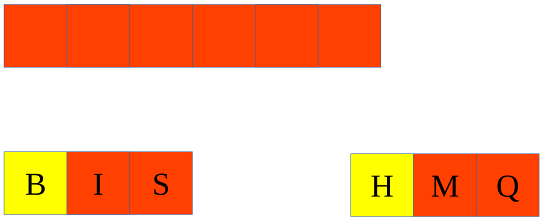
3)



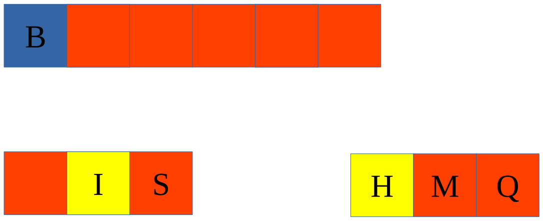
--->>>



5)



6)



7)

B H 🟥 🟥 🟥 🟥

🟥 I S   🟥 M Q

8)

B H I 🟥 🟥 🟥

🟥 🟥 S   🟥 M Q

9)

B H I M 🟥 🟥

🟥 🟥 S   🟥 🟥 Q

COMPARISON = 11
-->

B H I M Q S

right side:

C L R E P K

C L R   E P K

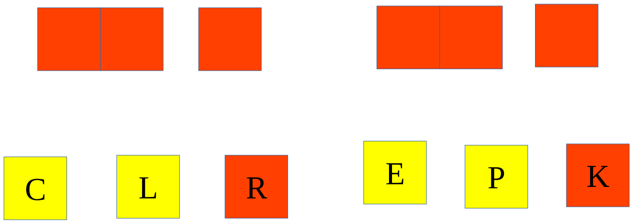
C L   R   E P   K

C   L   R   E   P   K

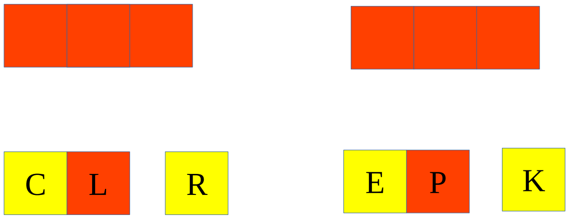
sorting RIGHT half of main array recursively…

comparison
yellow boxes are indicated comparing two elements.

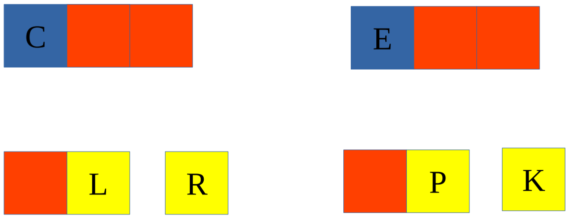
1)



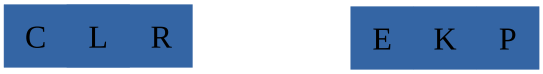
2)



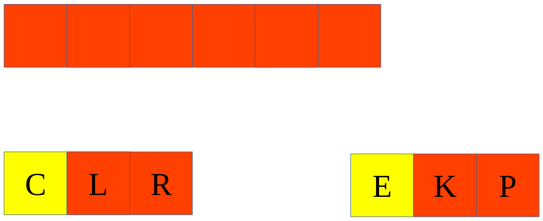
3)



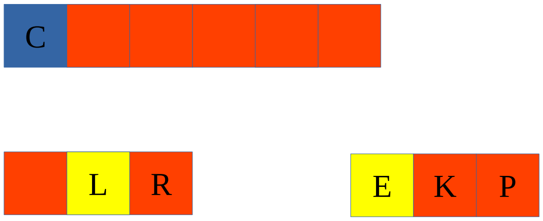
--->>>



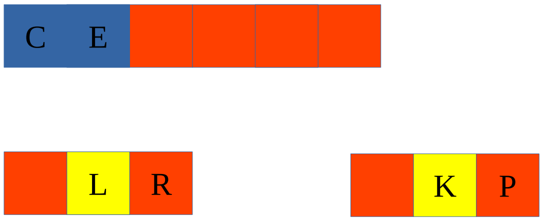
5)



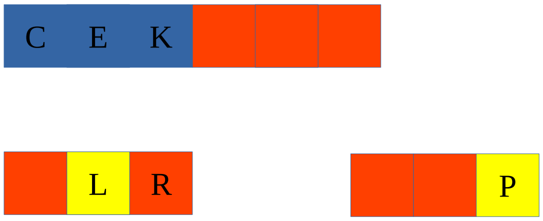
6)



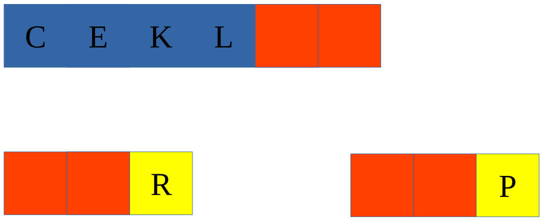
7)



8)

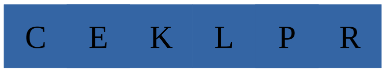


9)

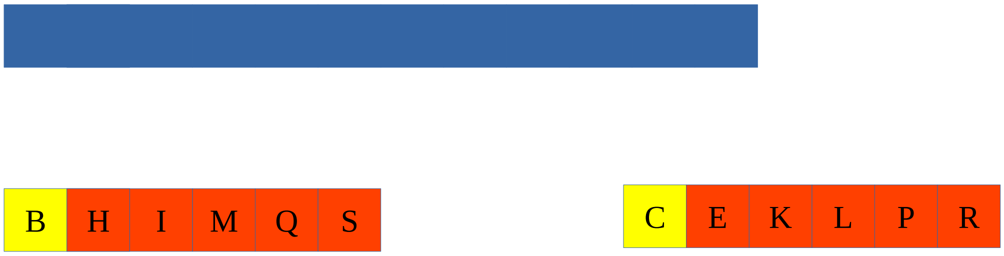


COMPARISON = 11

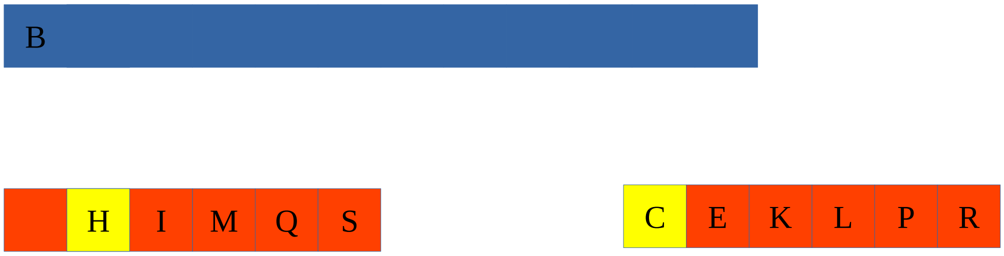
-->



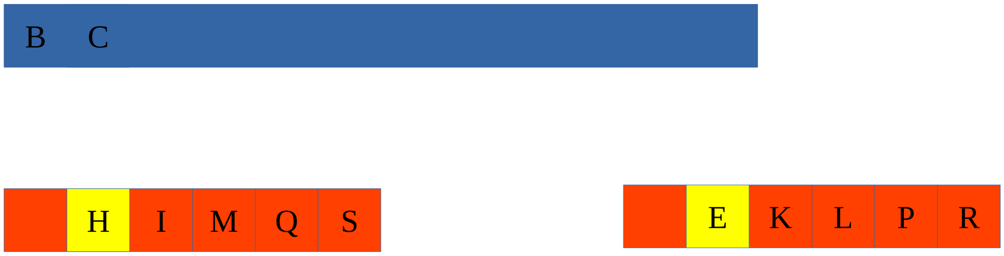
10)


B H I M Q S
C E K L P R

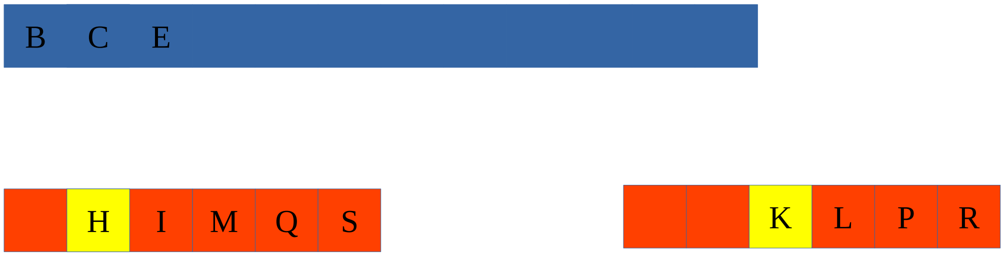
11)


B
H I M Q S
C E K L P R

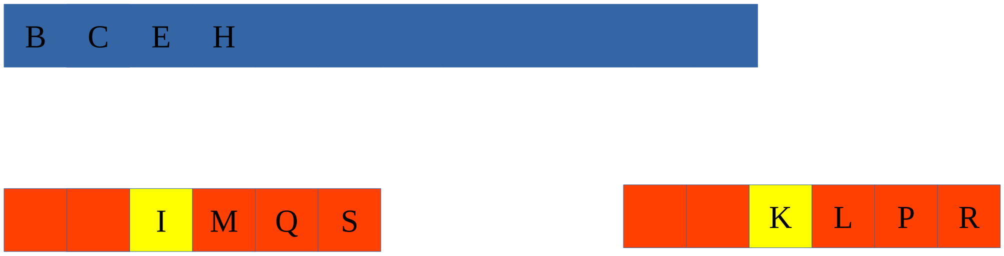
12)


B C
H I M Q S
E K L P R

13)


B C E
H I M Q S
K L P R

14)


B C E H
I M Q S
K L P R

15)

B C E H I

			M	Q	S

		K	L	P	R

16)

B C E H I K

			M	Q	S

			L	P	R

17)

B C E H I K L

			M	Q	S

				P	R

18)

B C E H I K L M

				Q	S

				P	R

19)

B C E H I K L M P

				Q	S

					R

19)

B	C	E	H	I	K	L	M	P	Q		

🟥	🟥	🟥	🟥	🟥	S

🟥	🟥	🟥	🟥	🟥	R

B	C	E	H	I	K	L	M	P	Q	R	S

TOTAL comparison is 33.

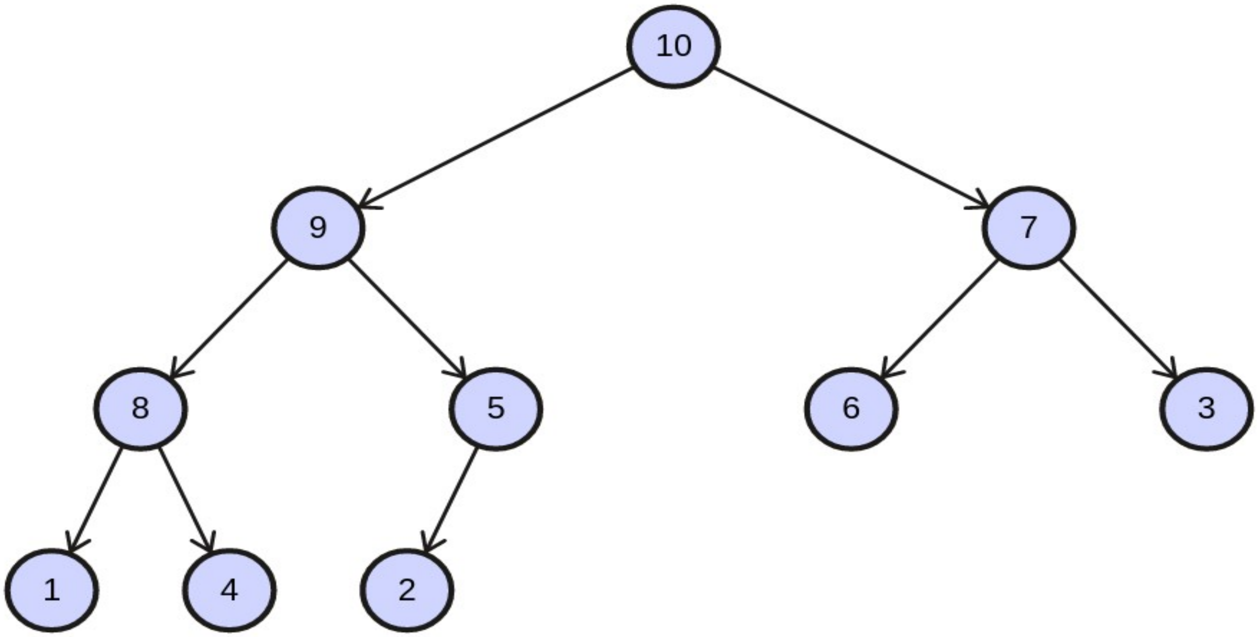
heap sort sort:

* A is an ordered integer array with 10 elements from small to large:

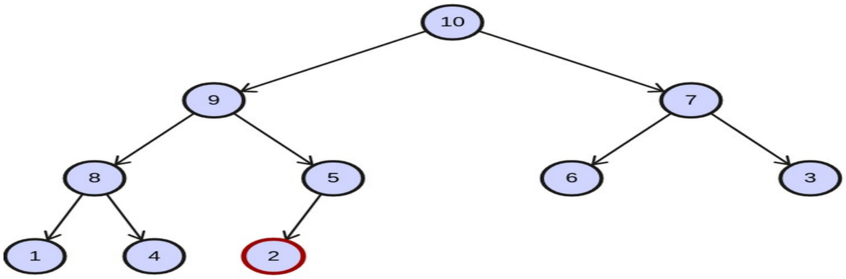
Algorithm for In-Place Heapsort

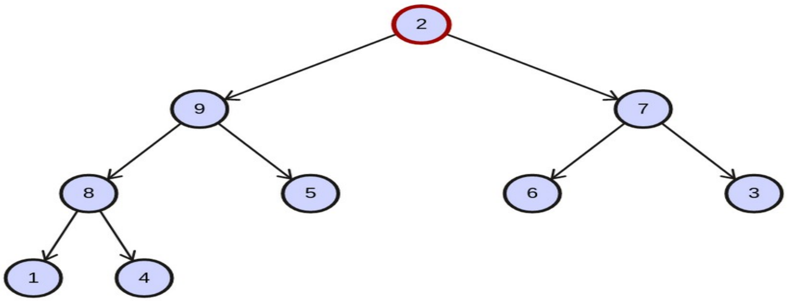
```
    1. Build a heap by rearranging the elements in an unsorted array
    2. while the heap is not empty
        3. Remove the first item from the heap by swapping it with the
        last item in the heap and restoring the heap property
```

so our heap will be like bellow:

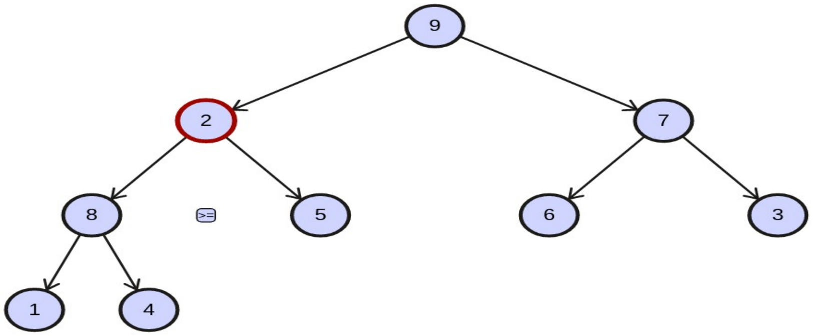


sorting steps:

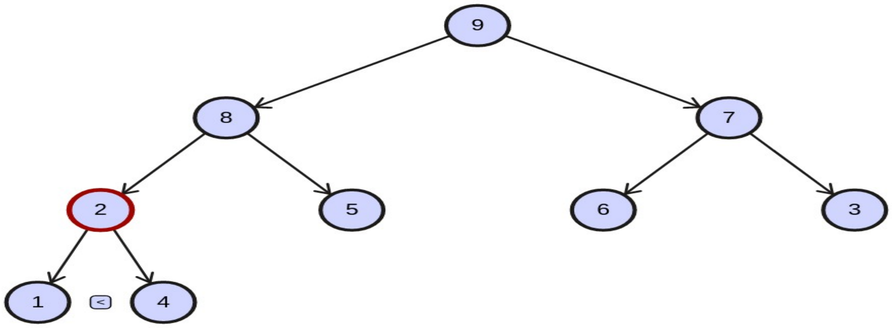


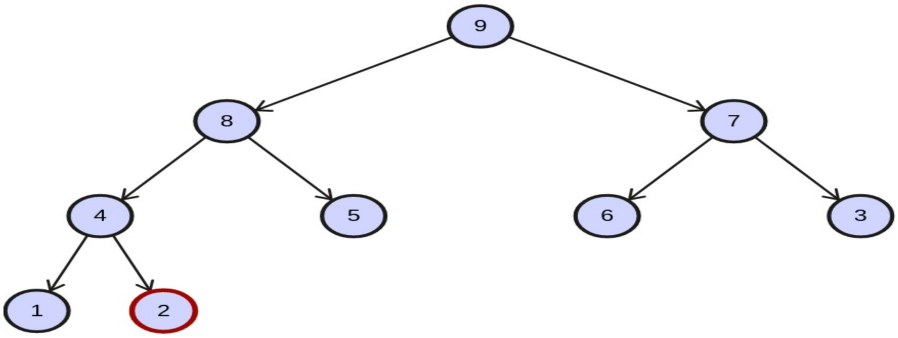


1	2	3	4	5	6	7	8	9	10
									10

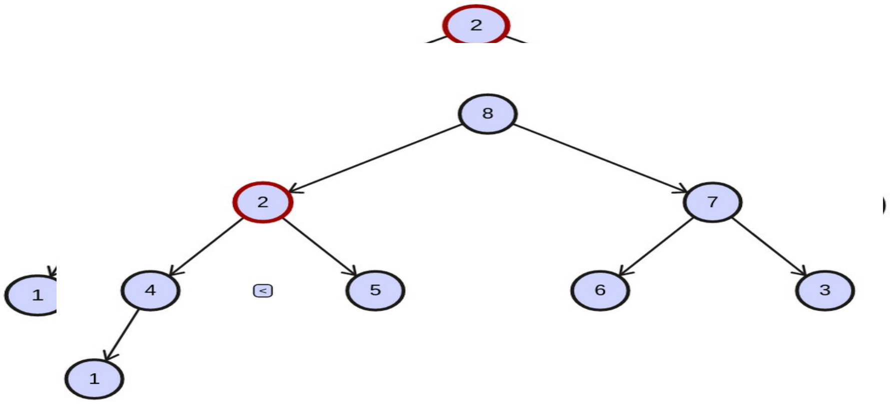


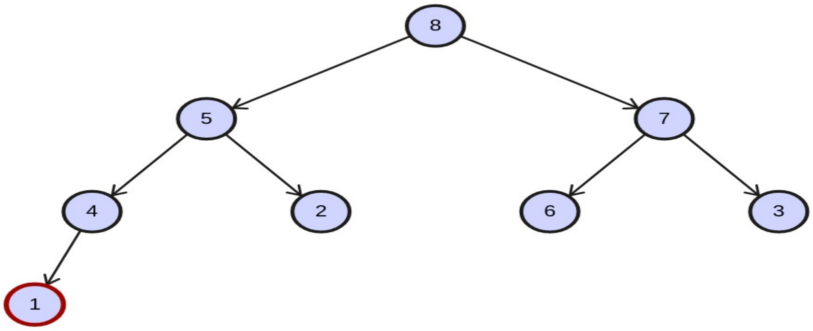
1	2	3	4	5	6	7	8	9	10
									10

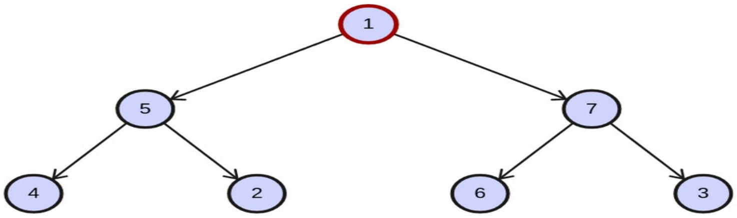




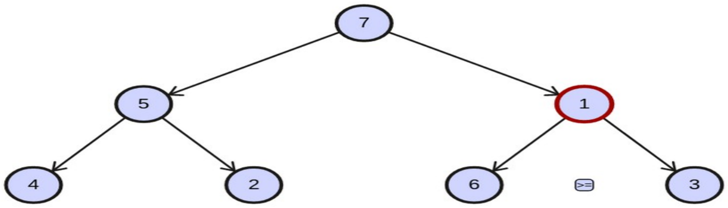
1	2	3	4	5	6	7	8	9	10
									10



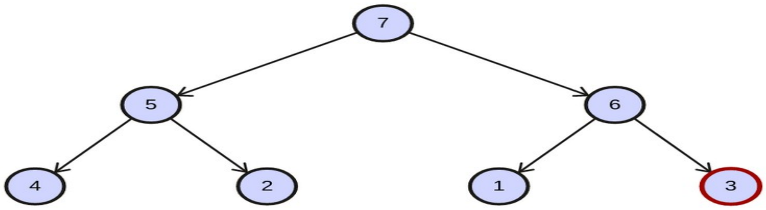


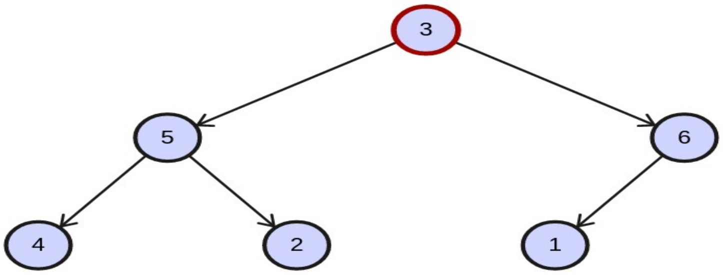


1	2	3	4	5	6	7	8	9	10
							8	9	10

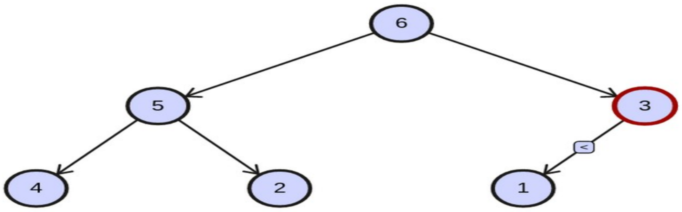


1	2	3	4	5	6	7	8	9	10
							8	9	10

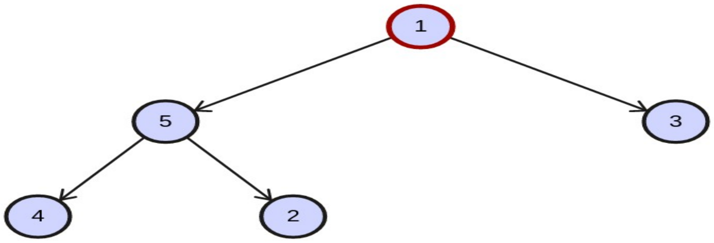


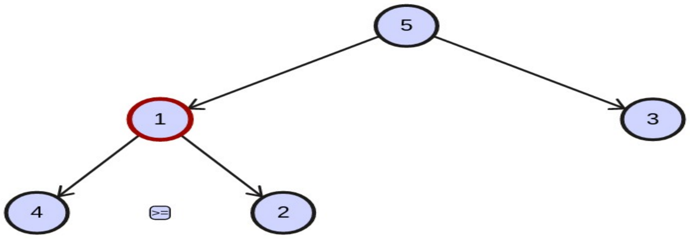


1	2	3	4	5	6	7	8	9	10
						7	8	9	10

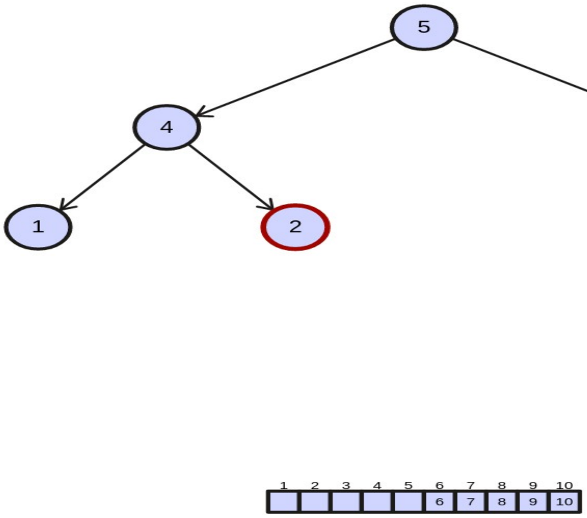


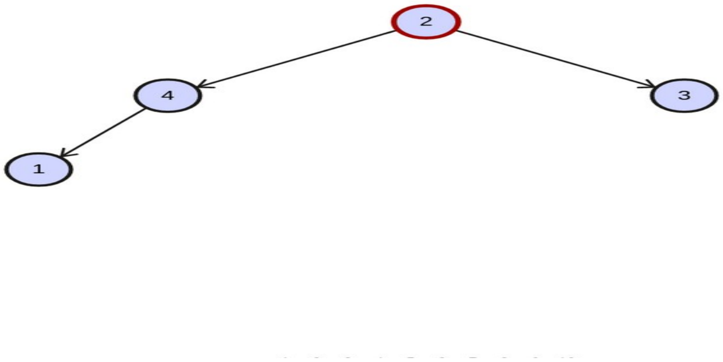
1	2	3	4	5	6	7	8	9	10
						7	8	9	10

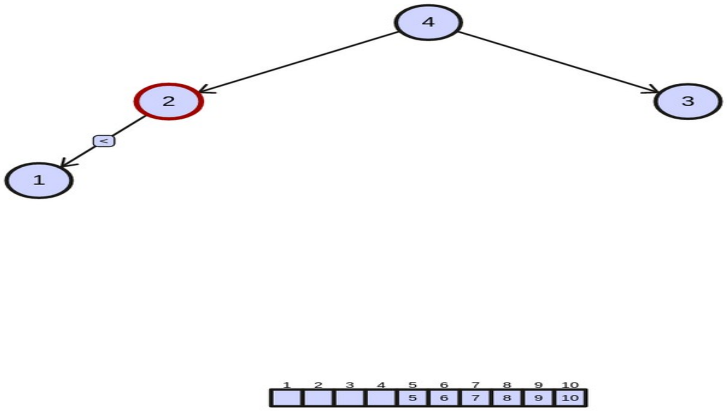


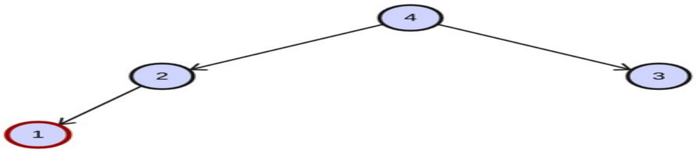


1	2	3	4	5	6	7	8	9	10
					6	7	8	9	10

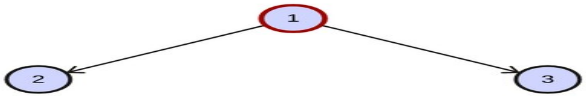

5
4
3
1
2
1 2 3 4 5 6 7 8 9 10
6 7 8 9 10


2
4
3
1

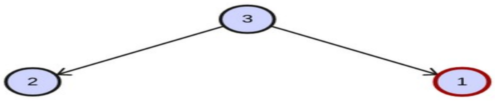

4
2
3
<
1
1 2 3 4 5 6 7 8 9 10
5 6 7 8 9 10



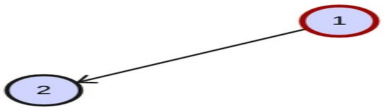
1	2	3	4	5	6	7	8	9	10
				5	6	7	8	9	10

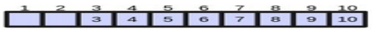


1	2	3	4	5	6	7	8	9	10
			4	5	6	7	8	9	10

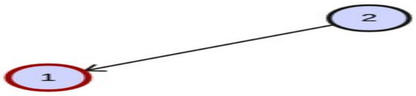


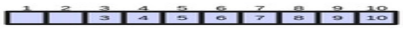
1	2	3	4	5	6	7	8	9	10
			4	5	6	7	8	9	10



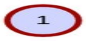


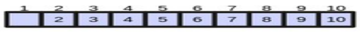
1	2	3	4	5	6	7	8	9	10
		3	4	5	6	7	8	9	10





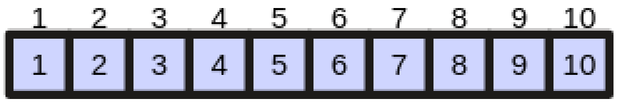
1	2	3	4	5	6	7	8	9	10
		3	4	5	6	7	8	9	10





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

sorted array:

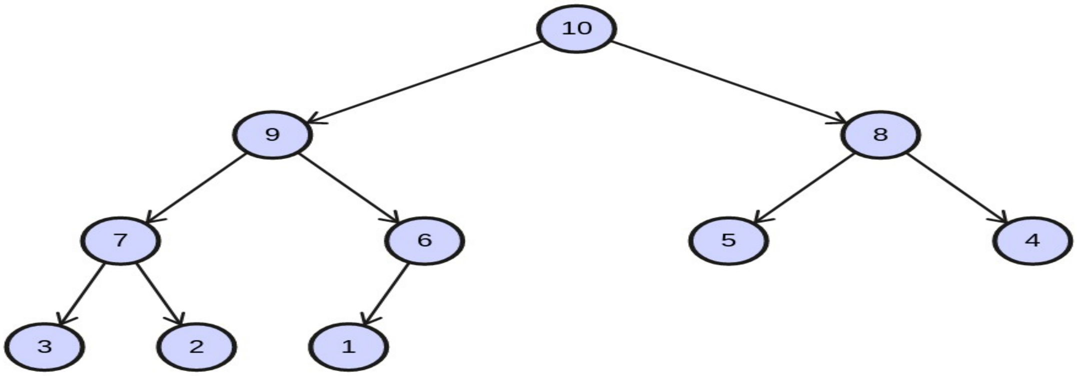


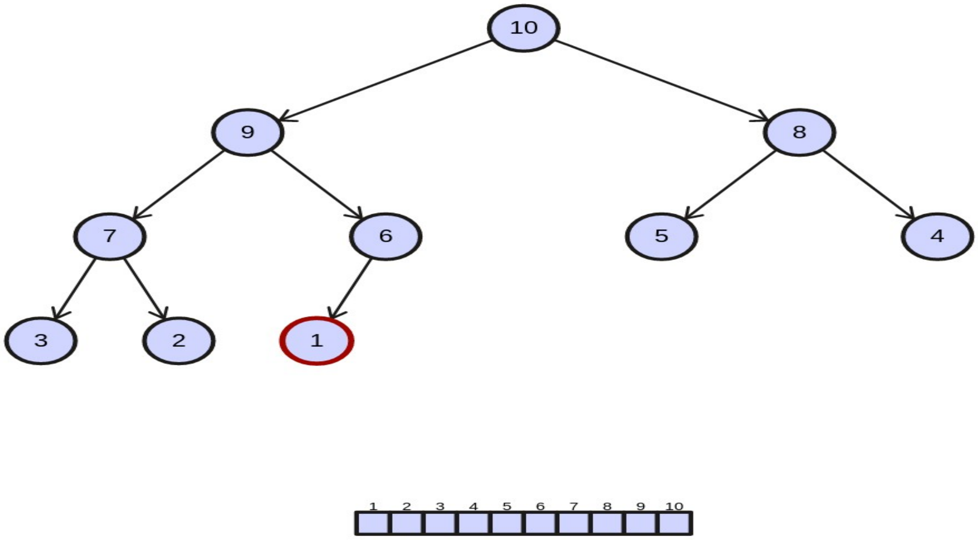
1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10

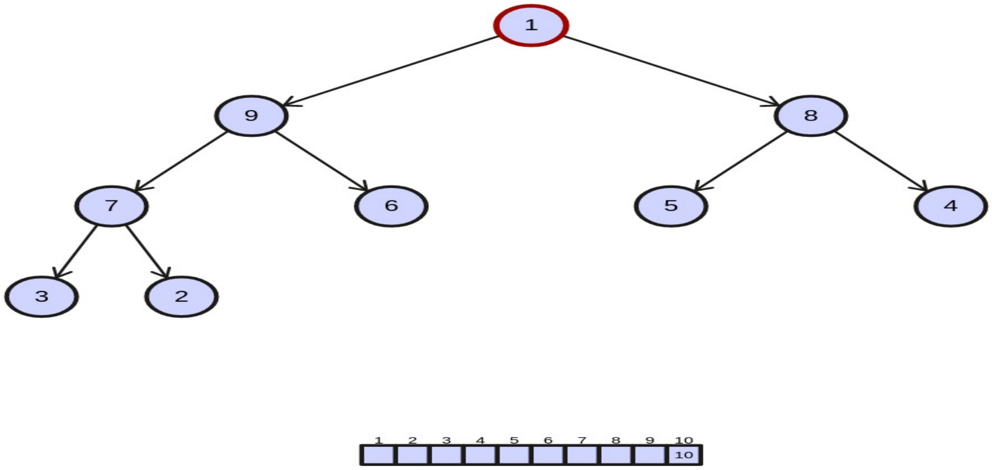
total displacements : 24

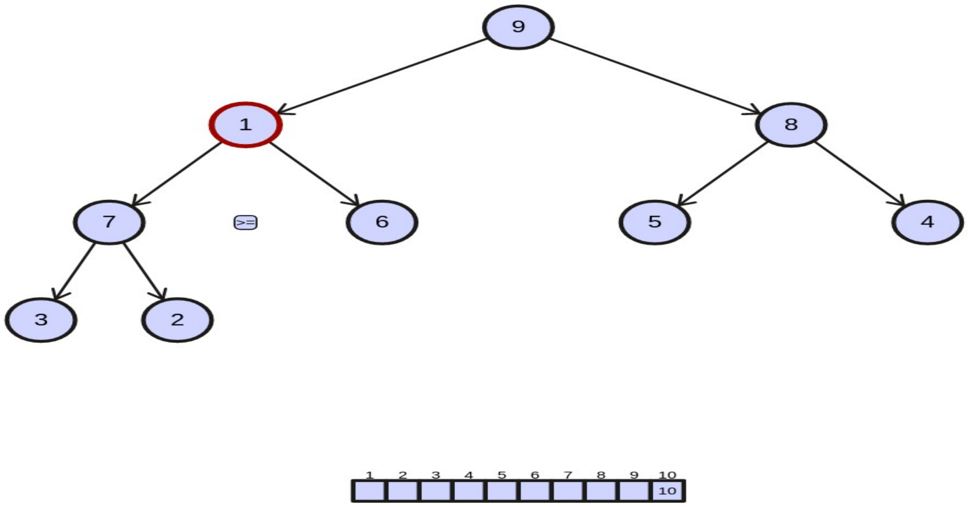
* B is an ordered integer array with 10 elements from large to small:

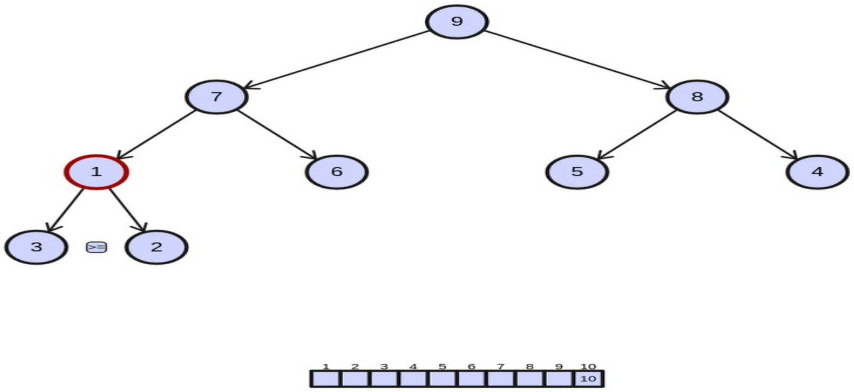
max heap will be :

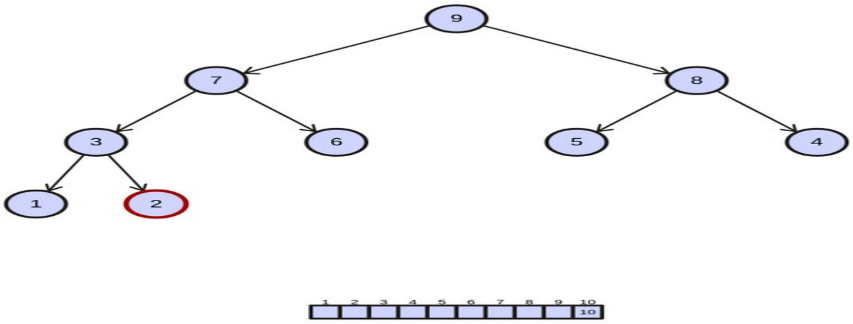

10
9
8
7
6
5
4
3
2
1

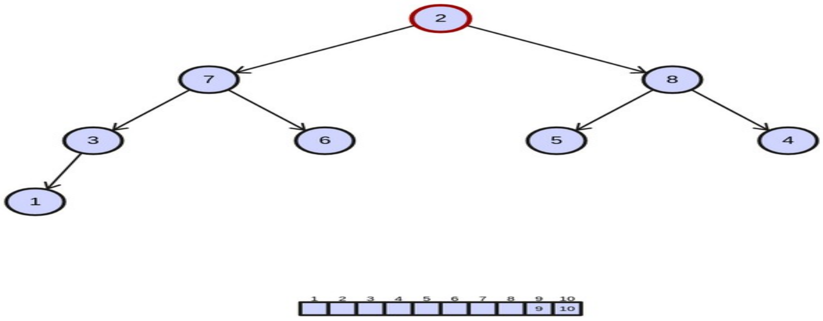

10
9
8
7
6
5
4
3
2
1
1 2 3 4 5 6 7 8 9 10

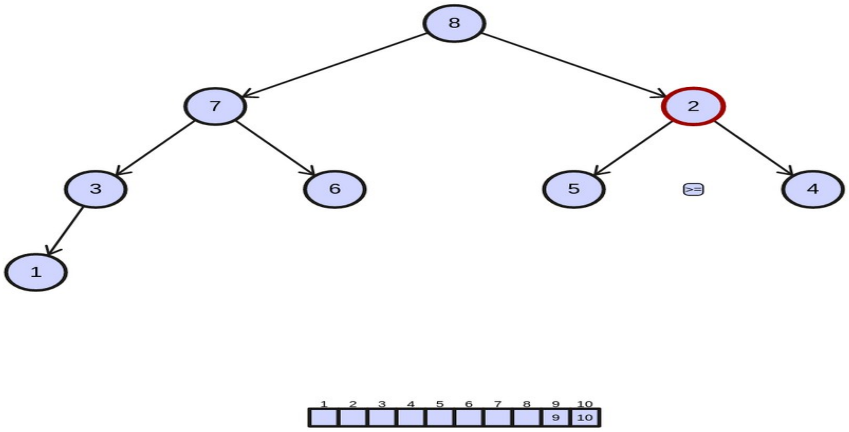

1
9
8
7
6
5
4
3
2
1 2 3 4 5 6 7 8 9 10
10

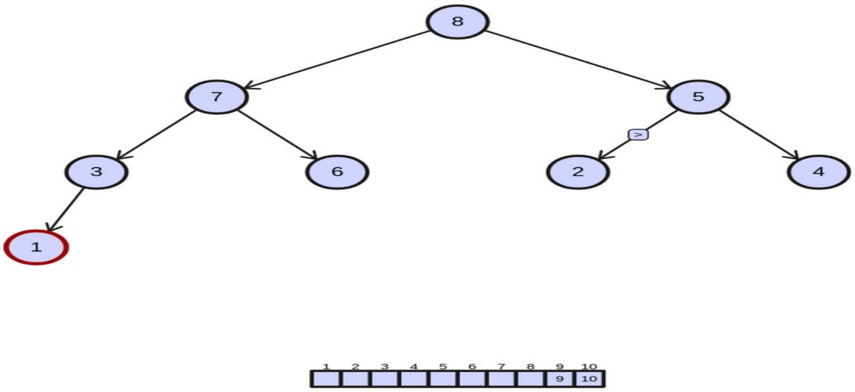

9
1
8
7
>=
6
5
4
3
2
1 2 3 4 5 6 7 8 9 10
10

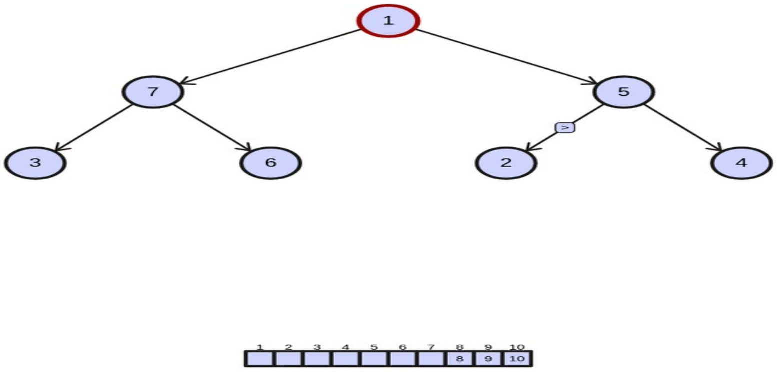

9
7
8
1
6
5
4
3
>=
2
1 2 3 4 5 6 7 8 9 10
10

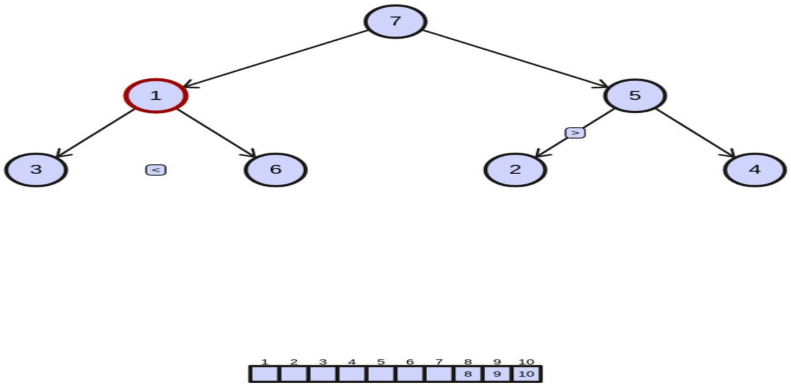

9
7
8
3
6
5
4
1
2
1 2 3 4 5 6 7 8 9 10
10

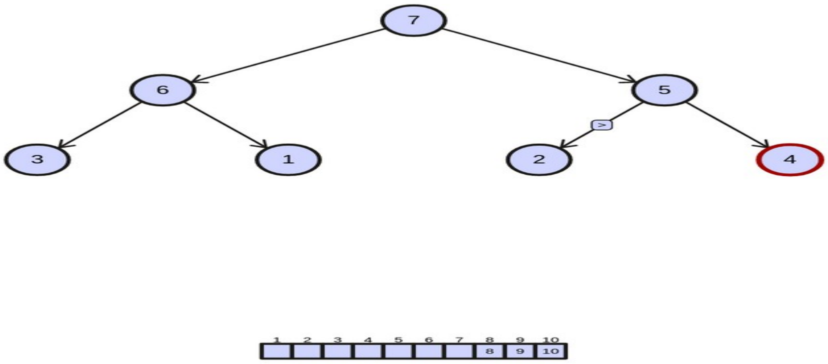

2
7
8
3
6
5
4
1
1 2 3 4 5 6 7 8 9 10
9
10

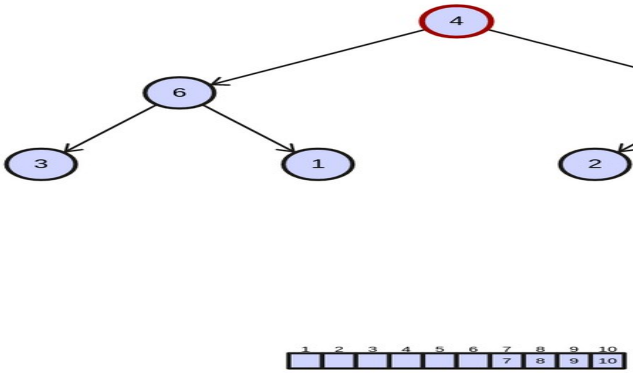

8
7
2
3
6
5
>=
4
1
1 2 3 4 5 6 7 8 9 10
9 10

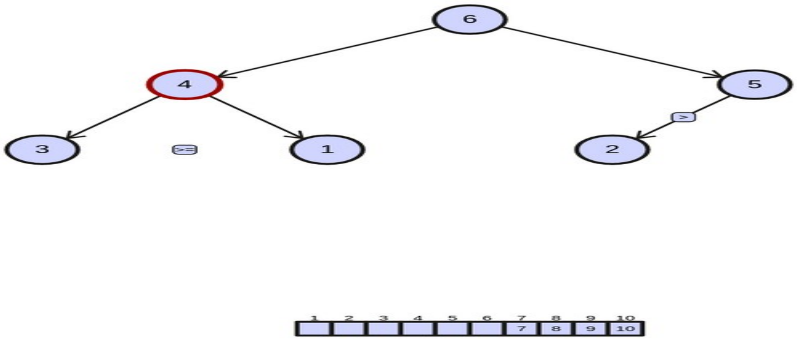

8
7
5
>
3
6
2
4
1
1 2 3 4 5 6 7 8 9 10
9 10

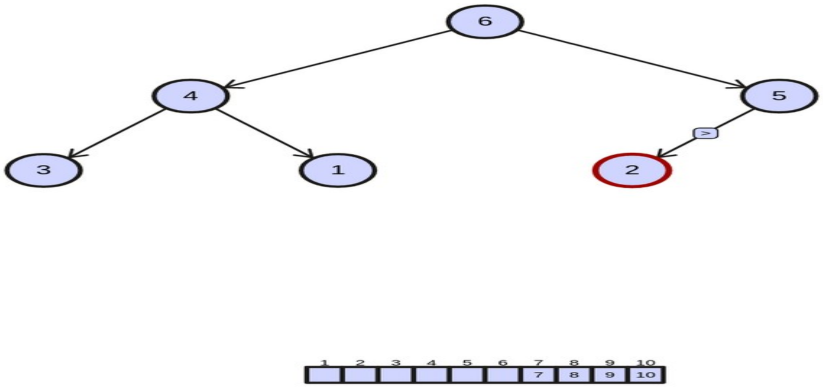

1
7
5
>
3
6
2
4
1 2 3 4 5 6 7 8 9 10
8 9 10

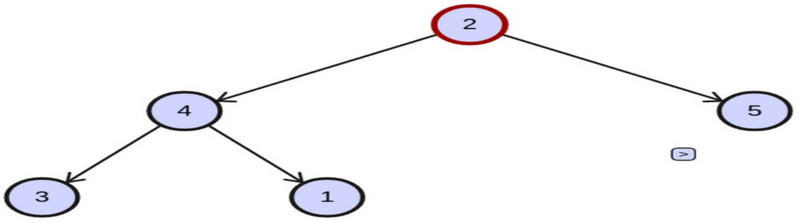

7
1
5
>
3
<
6
2
4
1 2 3 4 5 6 7 8 9 10
8 9 10


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

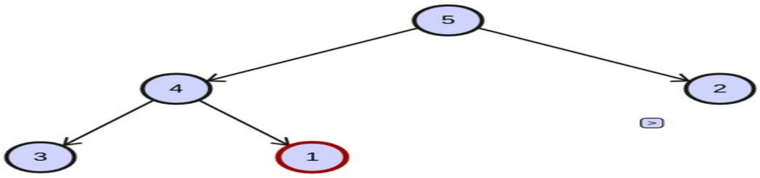

4
6
5
>
3
1
2
1 2 3 4 5 6 7 8 9 10
7 8 9 10


6
4
5
>
3
>=
1
2
1 2 3 4 5 6 7 8 9 10
7 8 9 10

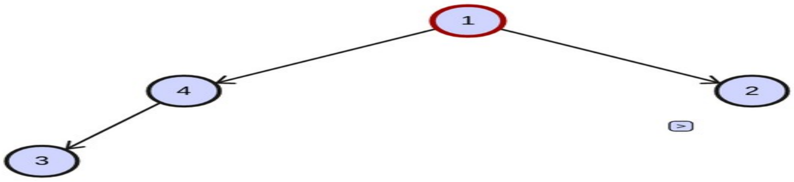

6
4
5
>
3
1
2
1 2 3 4 5 6 7 8 9 10
7 8 9 10



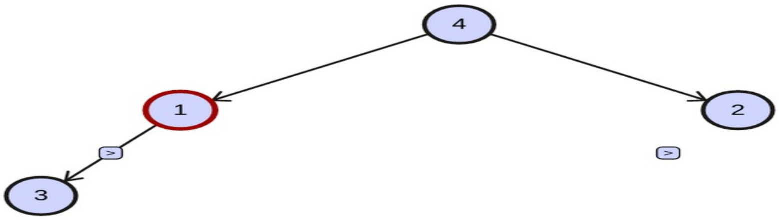
1	2	3	4	5	6	7	8	9	10
					6	7	8	9	10



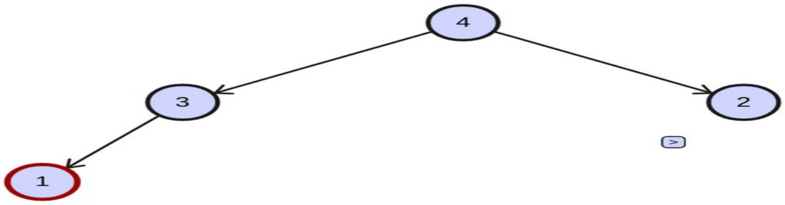
1	2	3	4	5	6	7	8	9	10
					6	7	8	9	10



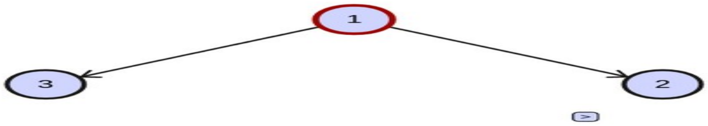
1	2	3	4	5	6	7	8	9	10
				5	6	7	8	9	10



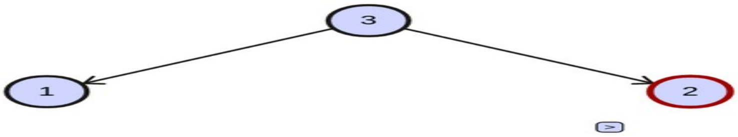
1	2	3	4	5	6	7	8	9	10
				5	6	7	8	9	10



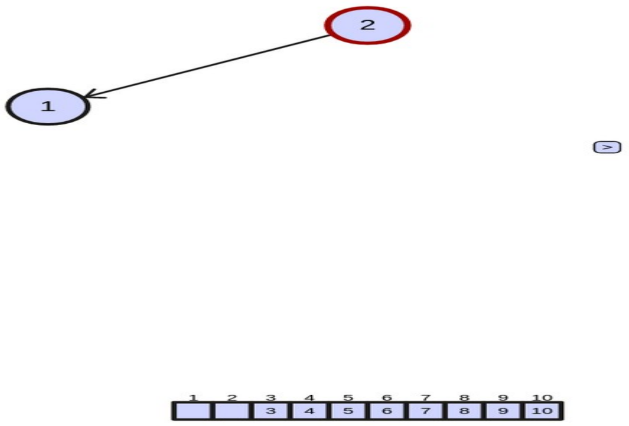
1	2	3	4	5	6	7	8	9	10
				5	6	7	8	9	10

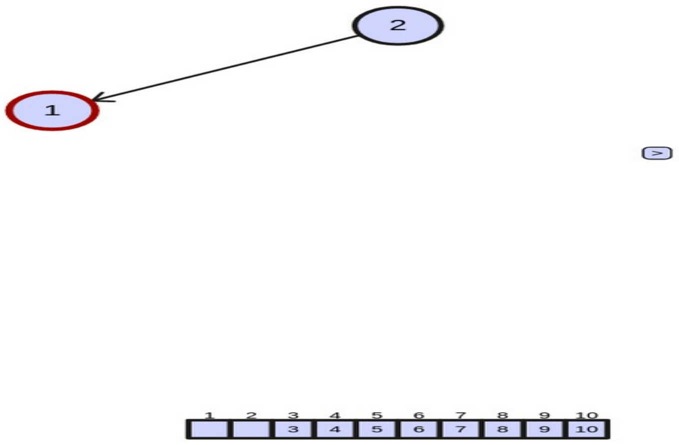


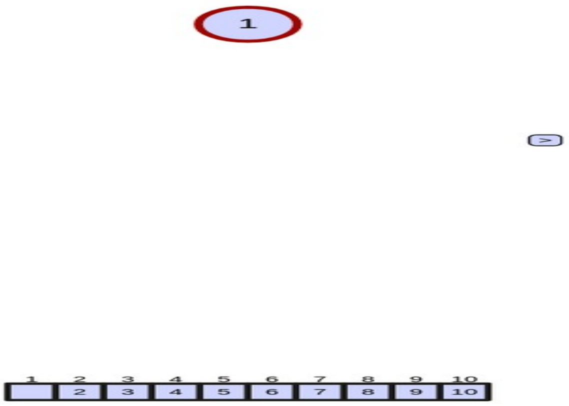
1	2	3	4	5	6	7	8	9	10
			4	5	6	7	8	9	10



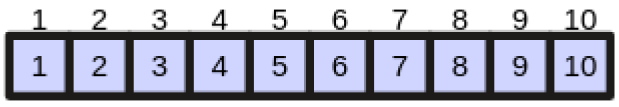
1	2	3	4	5	6	7	8	9	10
			4	5	6	7	8	9	10







we have sorted heap as:

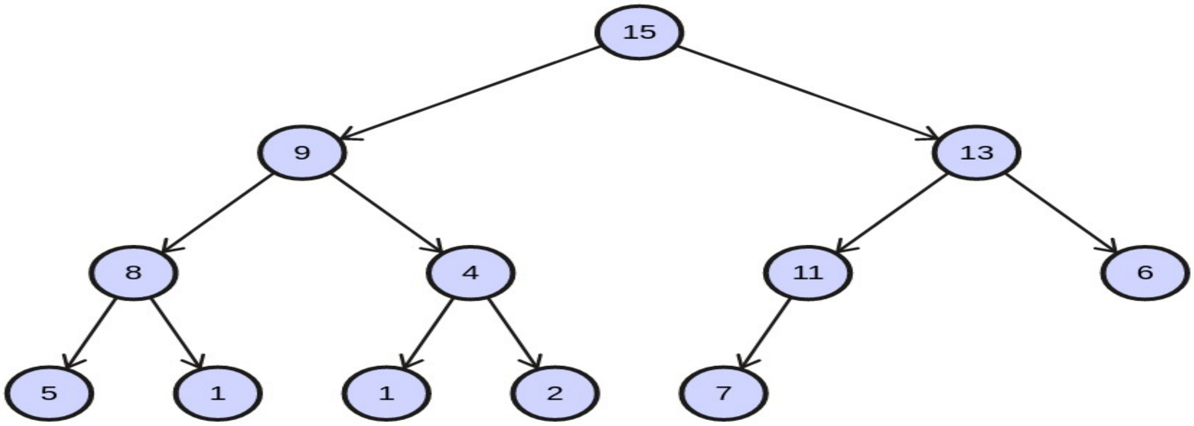


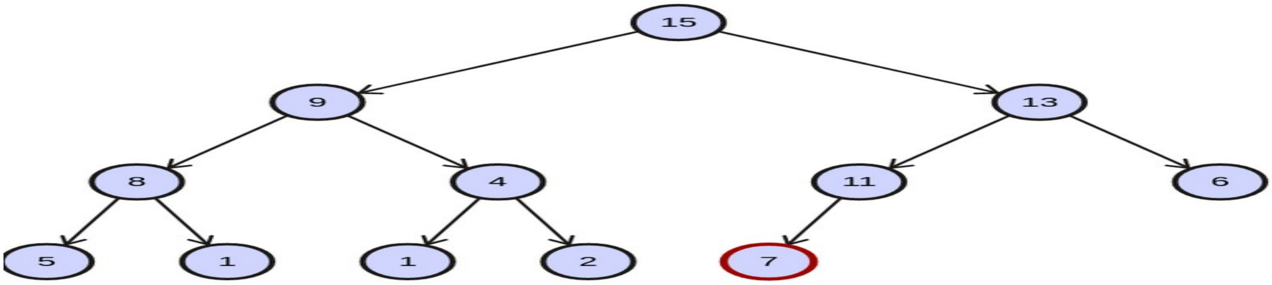
1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10

total displacement: 24

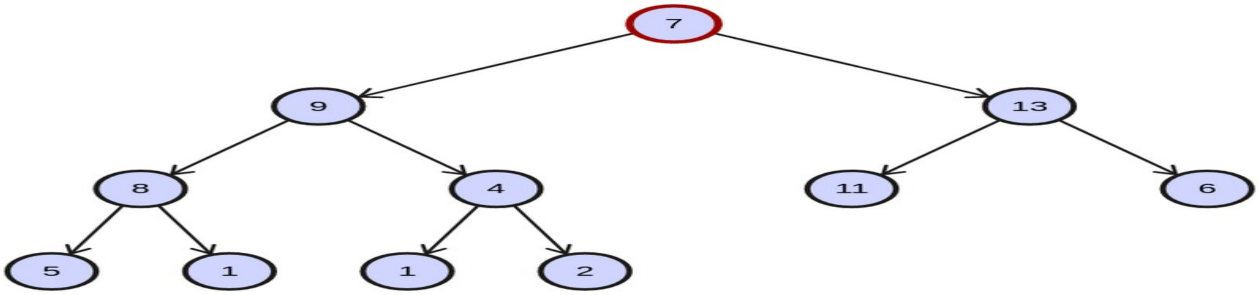
* C = {5, 2, 13, 9, 1, 7, 6, 8, 1, 15, 4, 11}

we have max heap as:

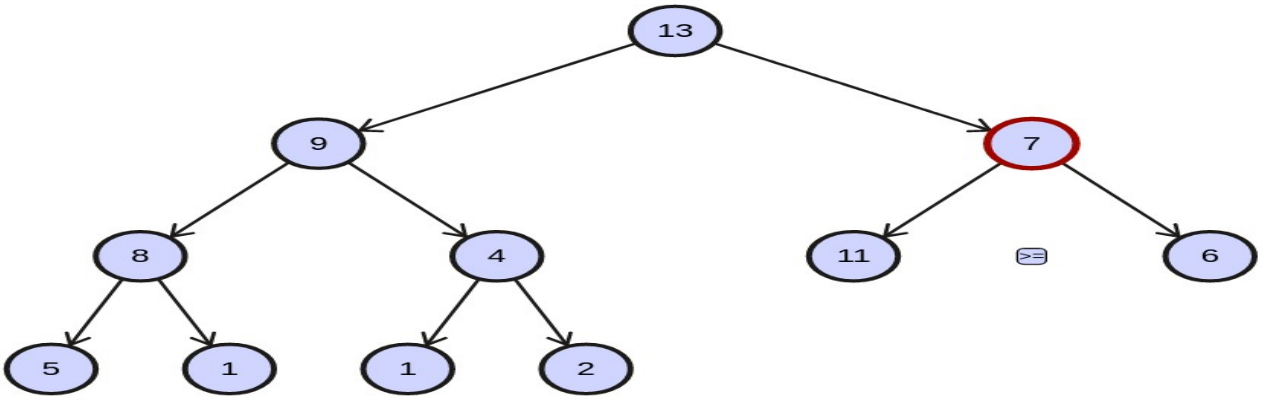




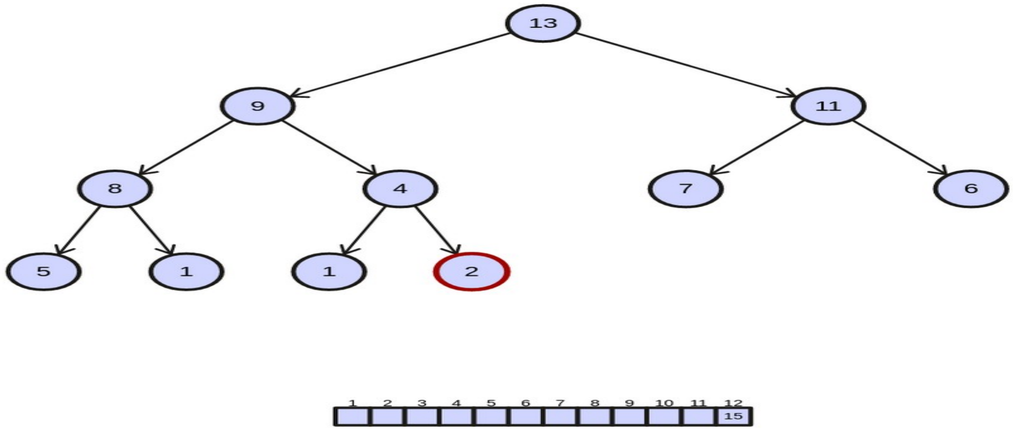
1	2	3	4	5	6	7	8	9	10	11	12

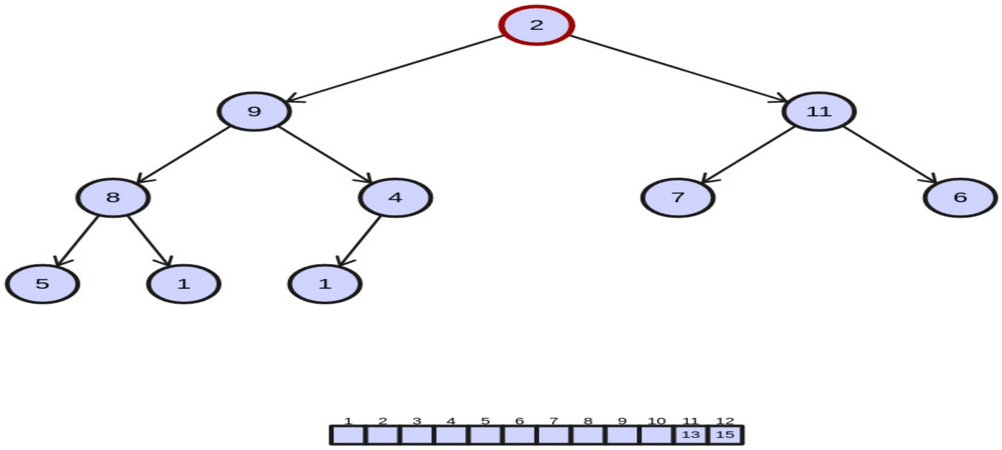


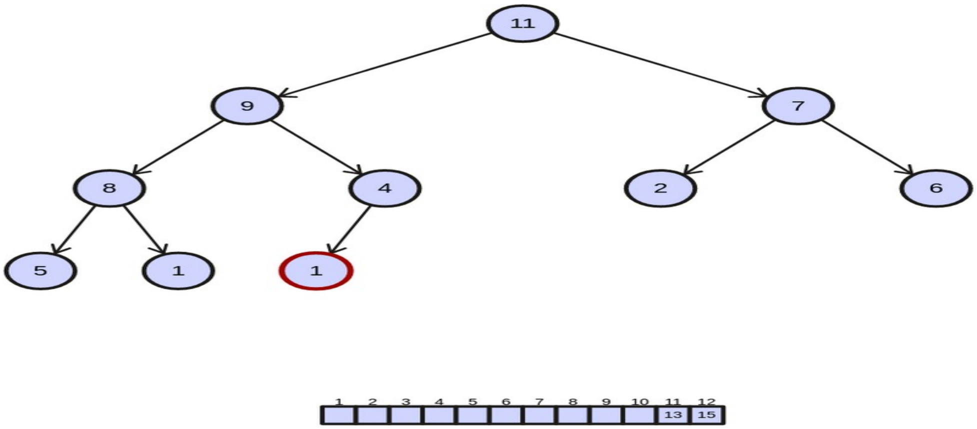
1	2	3	4	5	6	7	8	9	10	11	12
											15

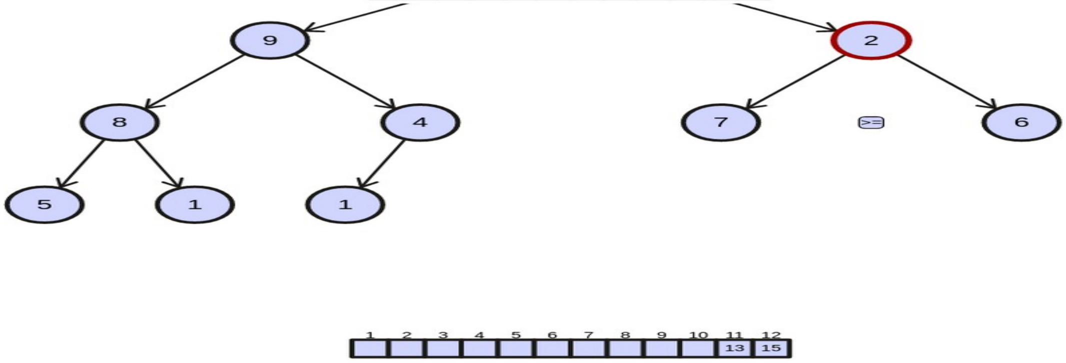


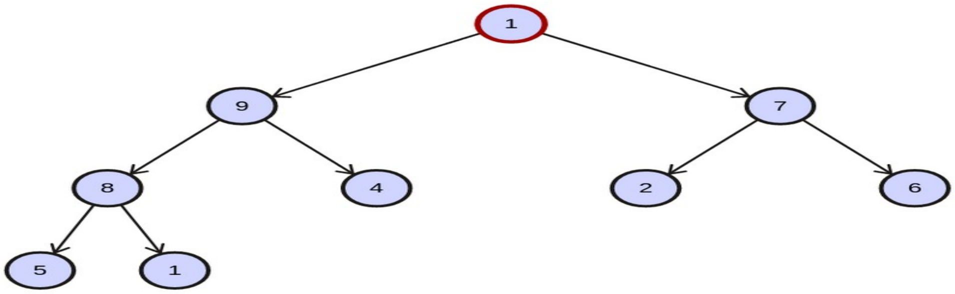
1	2	3	4	5	6	7	8	9	10	11	12
											15


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

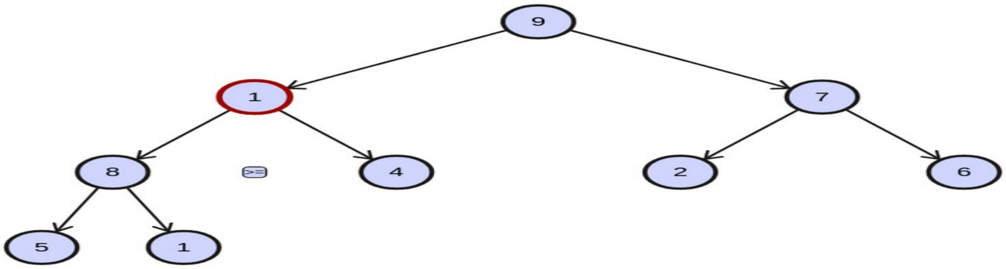

2
9
11
8
4
7
6
5
1
1
1 2 3 4 5 6 7 8 9 10 11 12
13 15


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

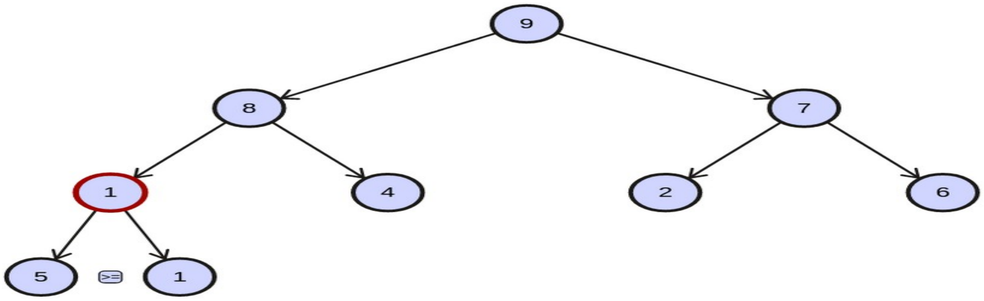

9
2
8
4
7
>=
6
5
1
1
1 2 3 4 5 6 7 8 9 10 11 12
13 15



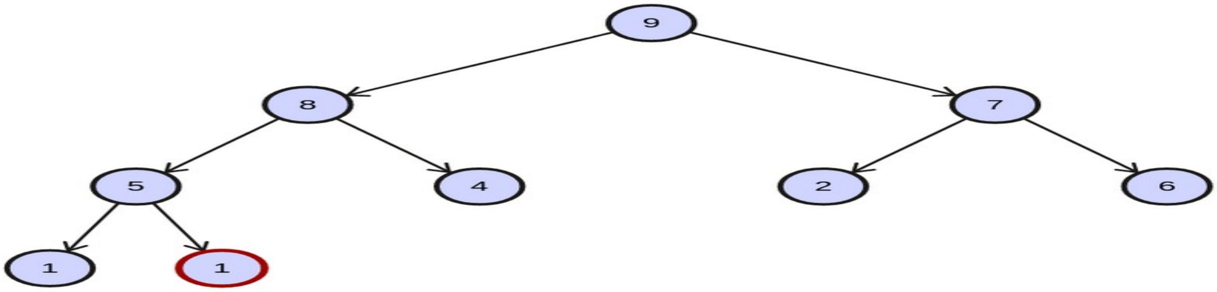
1	2	3	4	5	6	7	8	9	10	11	12
									11	13	15



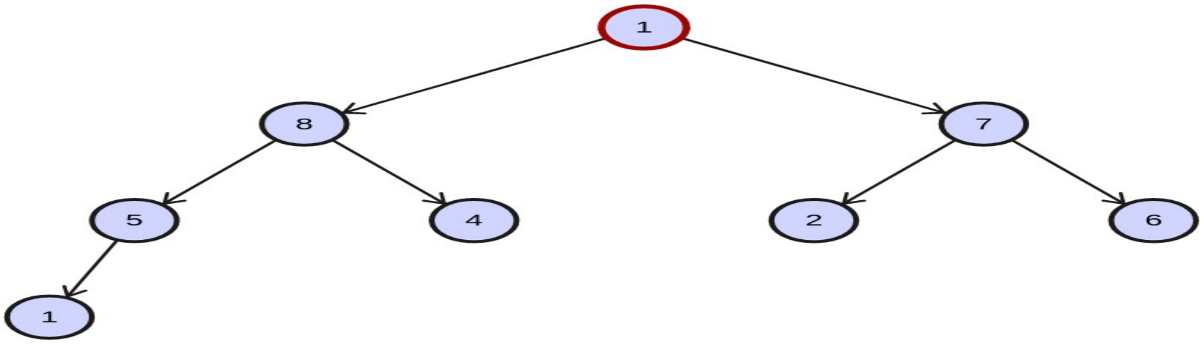
1	2	3	4	5	6	7	8	9	10	11	12
									11	13	15



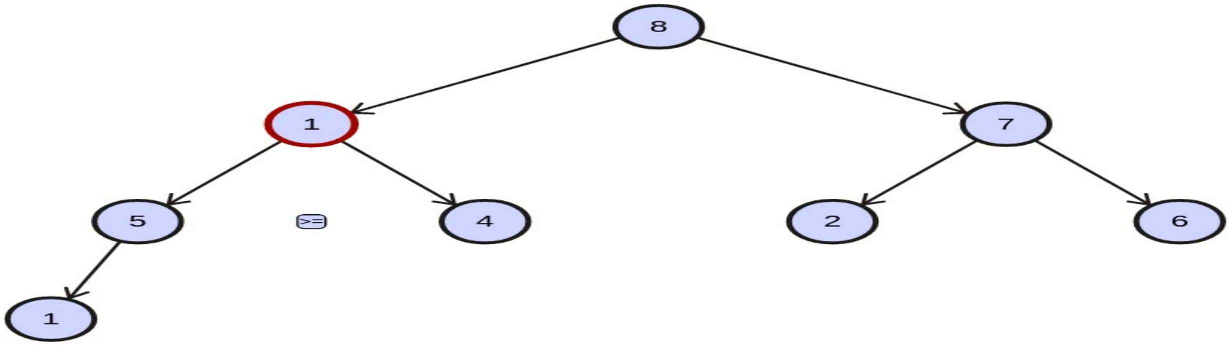
1	2	3	4	5	6	7	8	9	10	11	12
									11	13	15



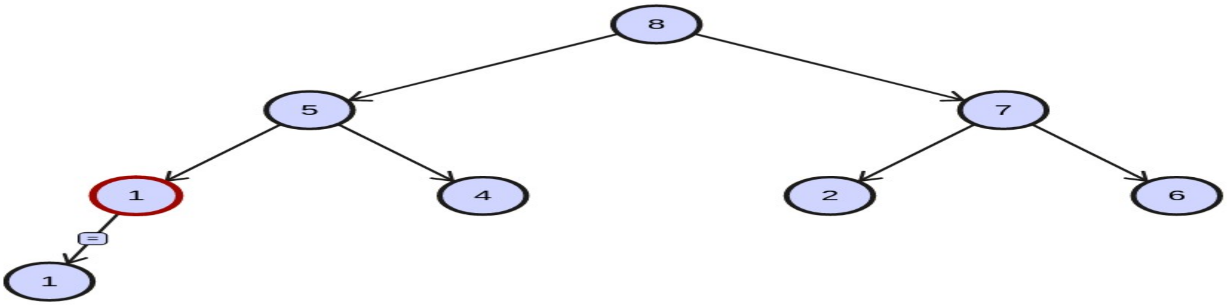
1	2	3	4	5	6	7	8	9	10	11	12
									11	13	15



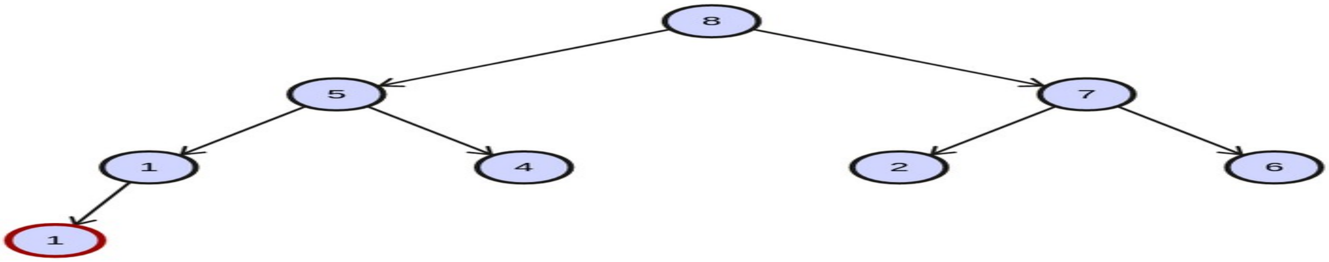
1	2	3	4	5	6	7	8	9	10	11	12
								9	11	13	15



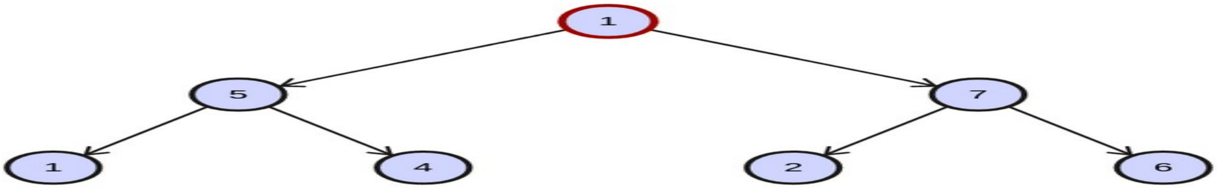
1	2	3	4	5	6	7	8	9	10	11	12
								9	11	13	15



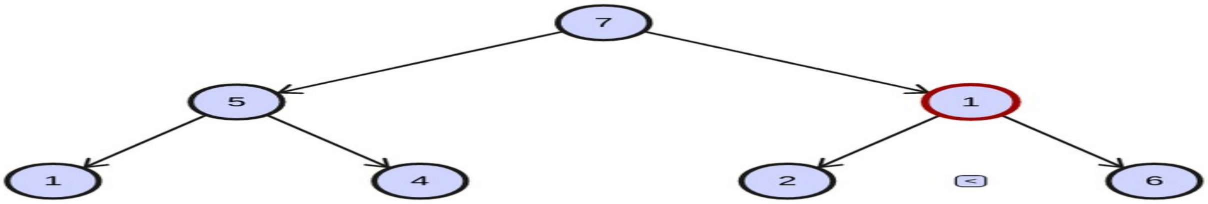
1	2	3	4	5	6	7	8	9	10	11	12
								9	11	13	15



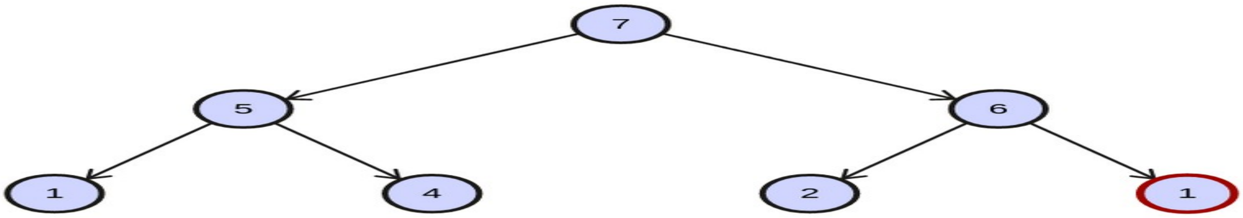
1	2	3	4	5	6	7	8	9	10	11	12
								9	11	13	15



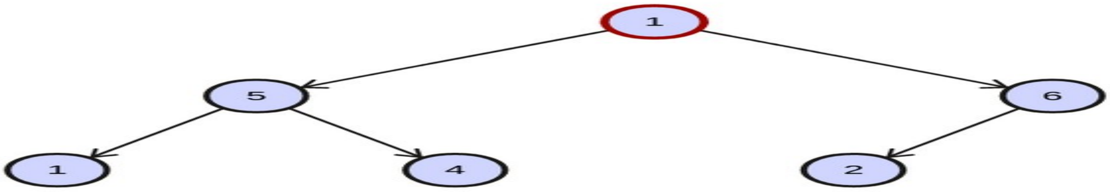
1	2	3	4	5	6	7	8	9	10	11	12
							8	9	11	13	15



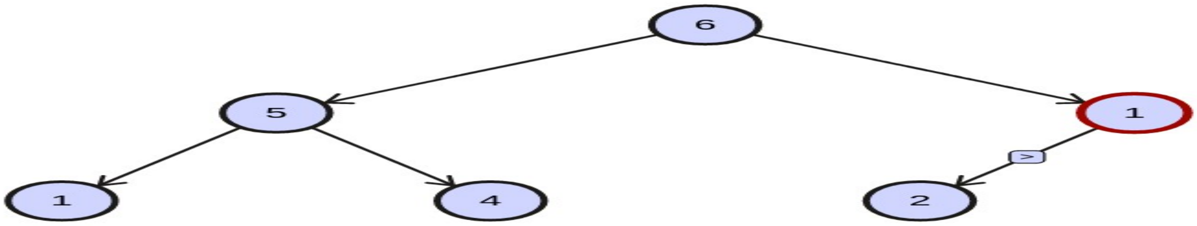
1	2	3	4	5	6	7	8	9	10	11	12
							8	9	11	13	15



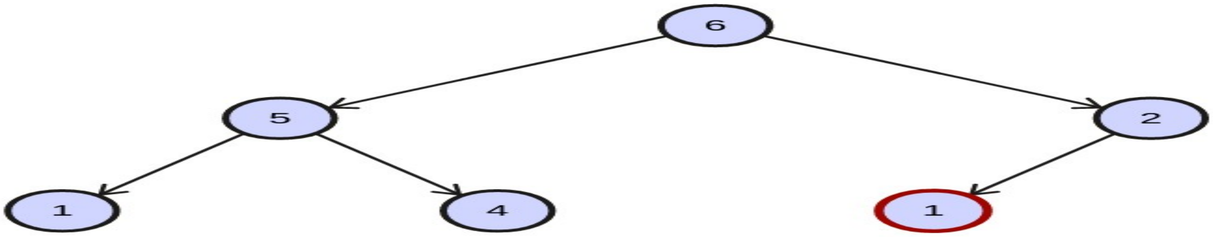
1	2	3	4	5	6	7	8	9	10	11	12
							8	9	11	13	15



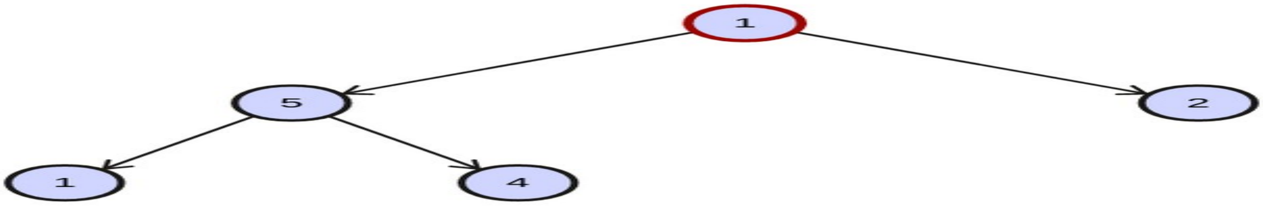
1	2	3	4	5	6	7	8	9	10	11	12
						7	8	9	11	13	15



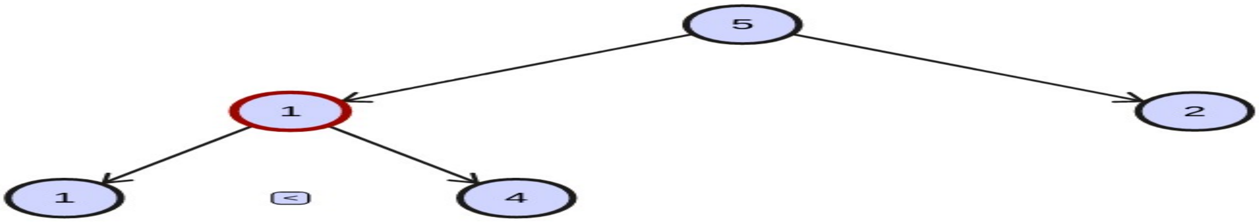
1	2	3	4	5	6	7	8	9	10	11	12
						7	8	9	11	13	15



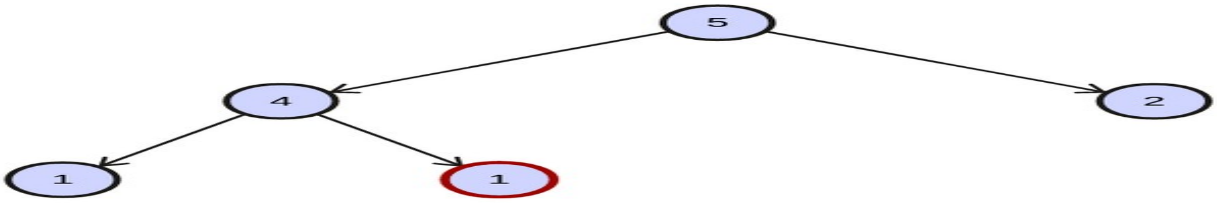
1	2	3	4	5	6	7	8	9	10	11	12
						7	8	9	11	13	15



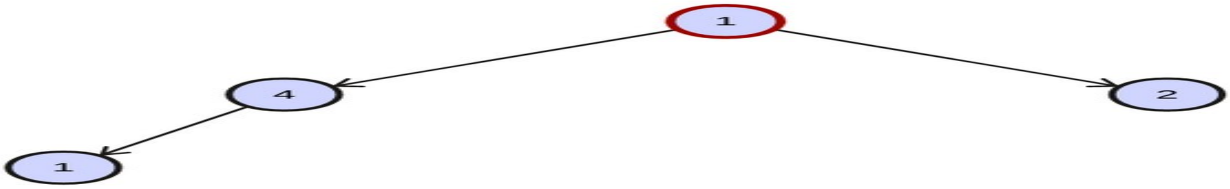
1	2	3	4	5	6	7	8	9	10	11	12
					6	7	8	9	11	13	15



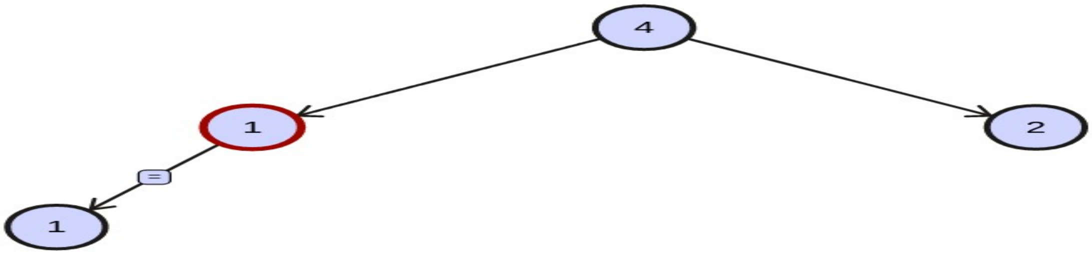
1	2	3	4	5	6	7	8	9	10	11	12
					6	7	8	9	11	13	15



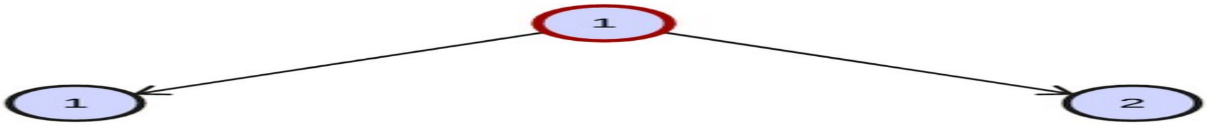
1	2	3	4	5	6	7	8	9	10	11	12
					6	7	8	9	11	13	15



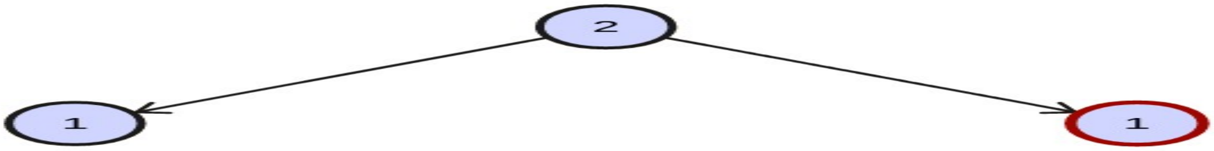
1	2	3	4	5	6	7	8	9	10	11	12
				5	6	7	8	9	11	13	15



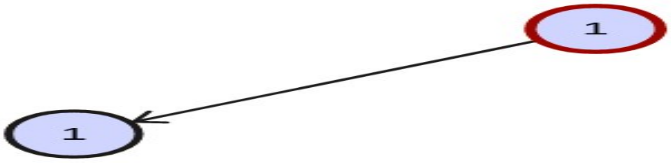
1	2	3	4	5	6	7	8	9	10	11	12
				5	6	7	8	9	11	13	15

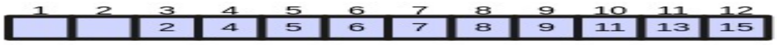


1	2	3	4	5	6	7	8	9	10	11	12
			4	5	6	7	8	9	11	13	15

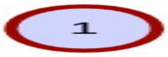


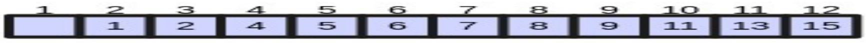
1	2	3	4	5	6	7	8	9	10	11	12
			4	5	6	7	8	9	11	13	15





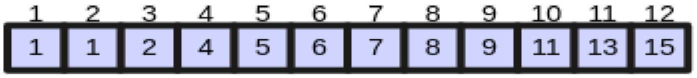
1	2	3	4	5	6	7	8	9	10	11	12
		2	4	5	6	7	8	9	11	13	15





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

sorted array: 30

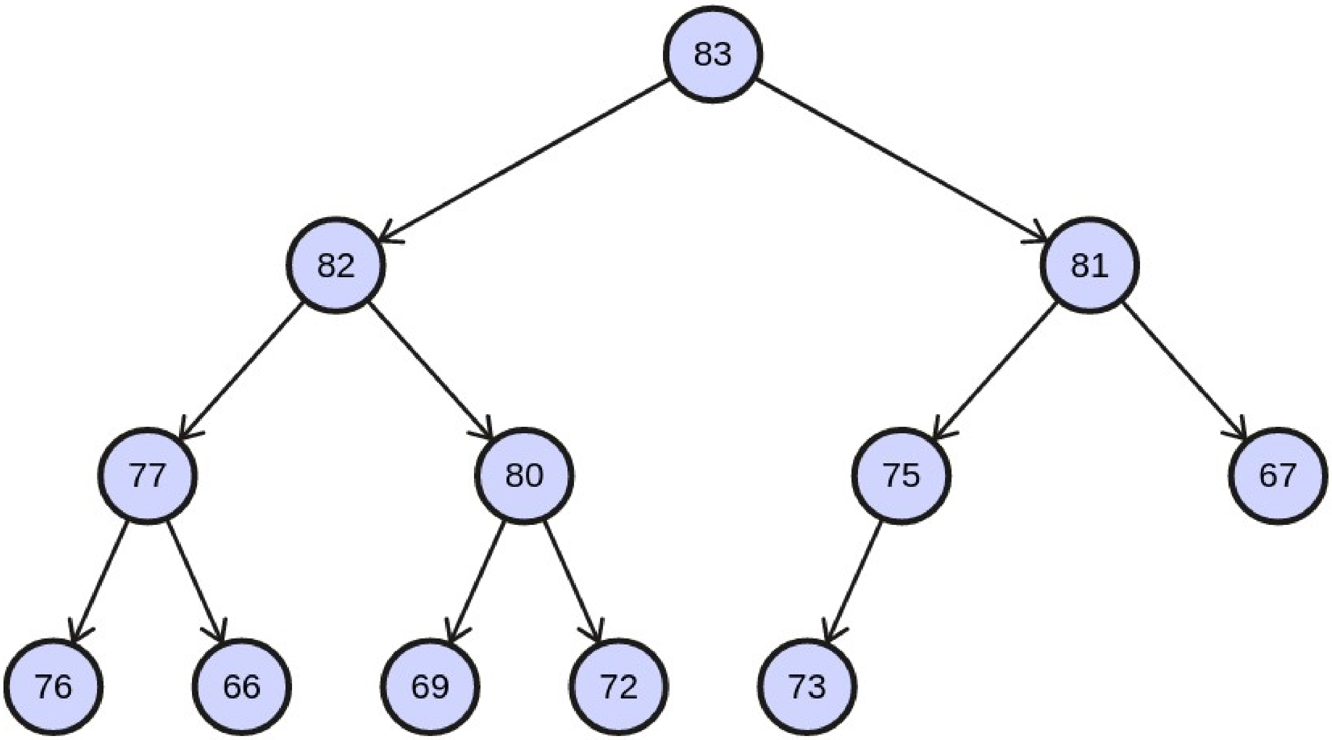


1	2	3	4	5	6	7	8	9	10	11	12
1	1	2	4	5	6	7	8	9	11	13	15

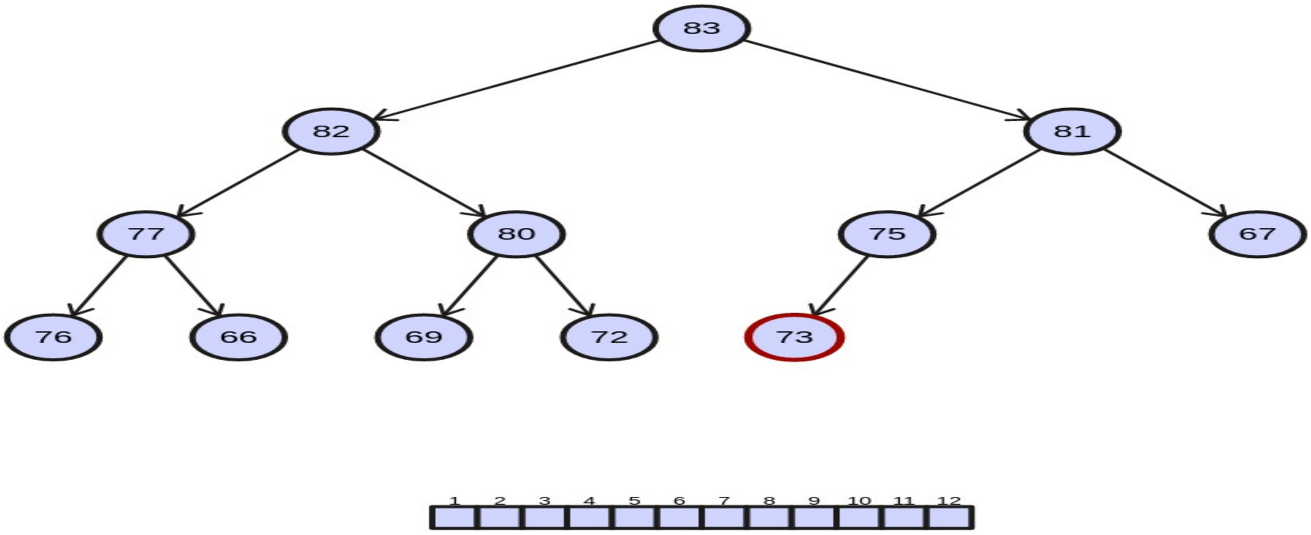
total displacement: 30

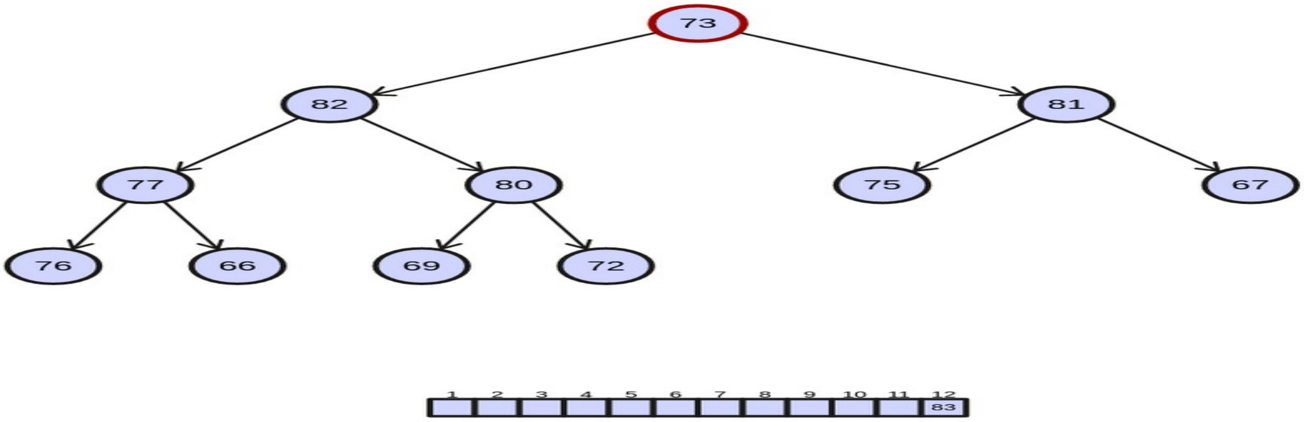
* D = {‘S’, ‘B’, ‘I’, ‘M’, ‘H’, ‘Q’, ‘C’, ‘L’, ‘R’, ‘E’, ‘P’, ‘K’}
USING ascii values of above letter we have heap max as :

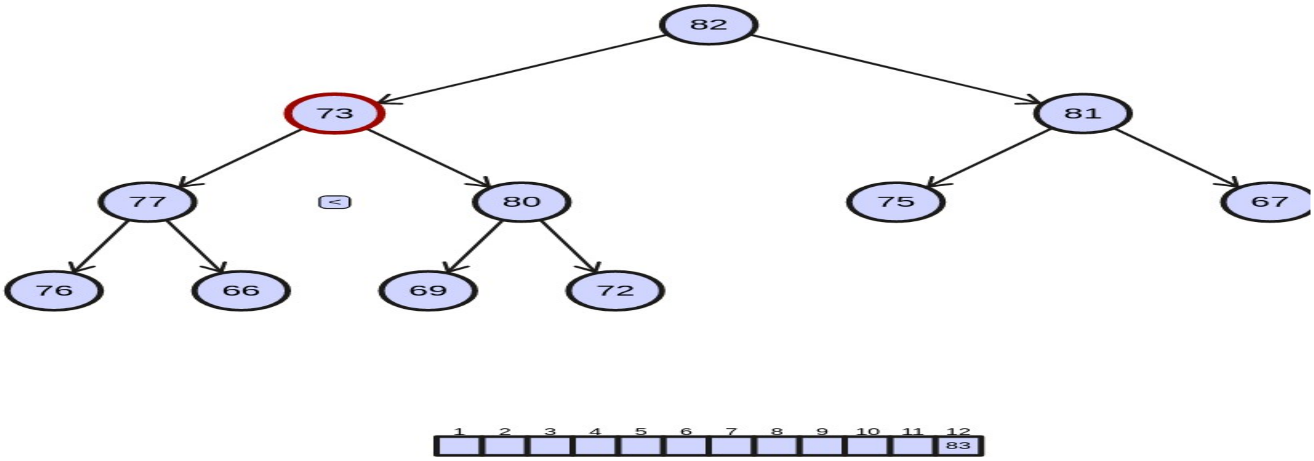
S = 83 , B = 66 , I = 73 , M = 77 , H = 72 , Q = 81 , C = 67 , L = 76 , R = 83 , E = 69 , P = 80 , K= 75

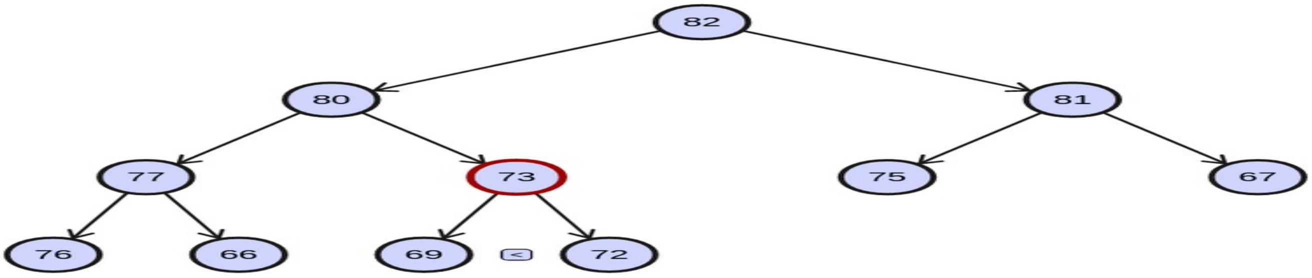


steps:

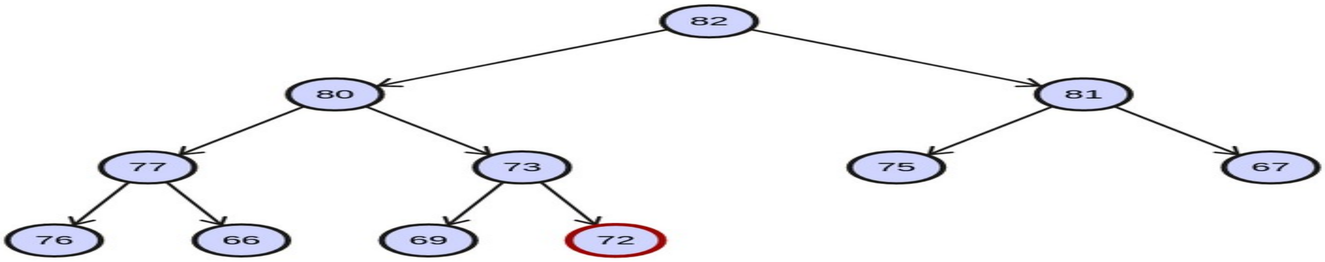

83
82
81
77
80
75
67
76
66
69
72
73
1 2 3 4 5 6 7 8 9 10 11 12


73
82
81
77
80
75
67
76
66
69
72
1 2 3 4 5 6 7 8 9 10 11 12
83

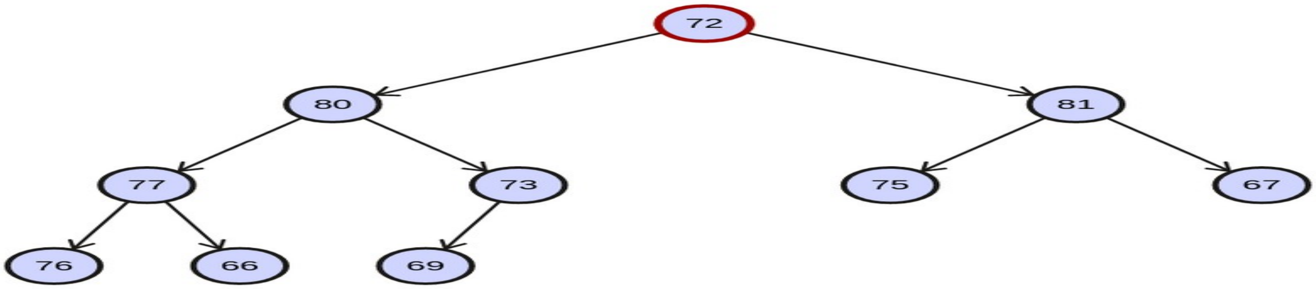

82
73
81
77
<
80
75
67
76
66
69
72
1 2 3 4 5 6 7 8 9 10 11 12
83



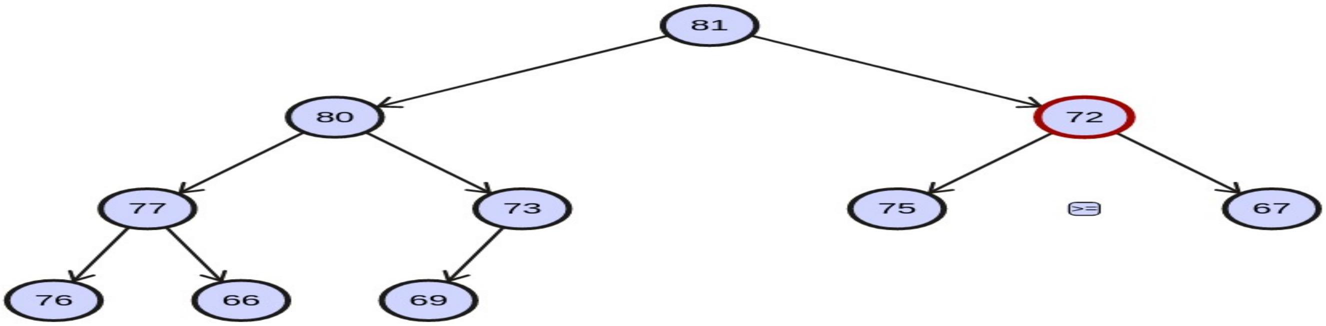
1	2	3	4	5	6	7	8	9	10	11	12
											83



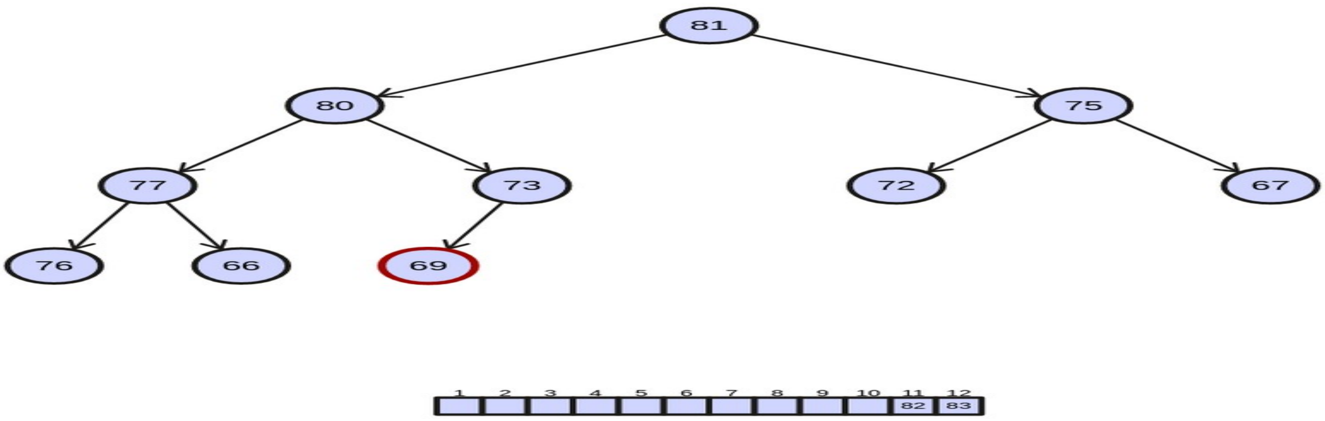
1	2	3	4	5	6	7	8	9	10	11	12
											83

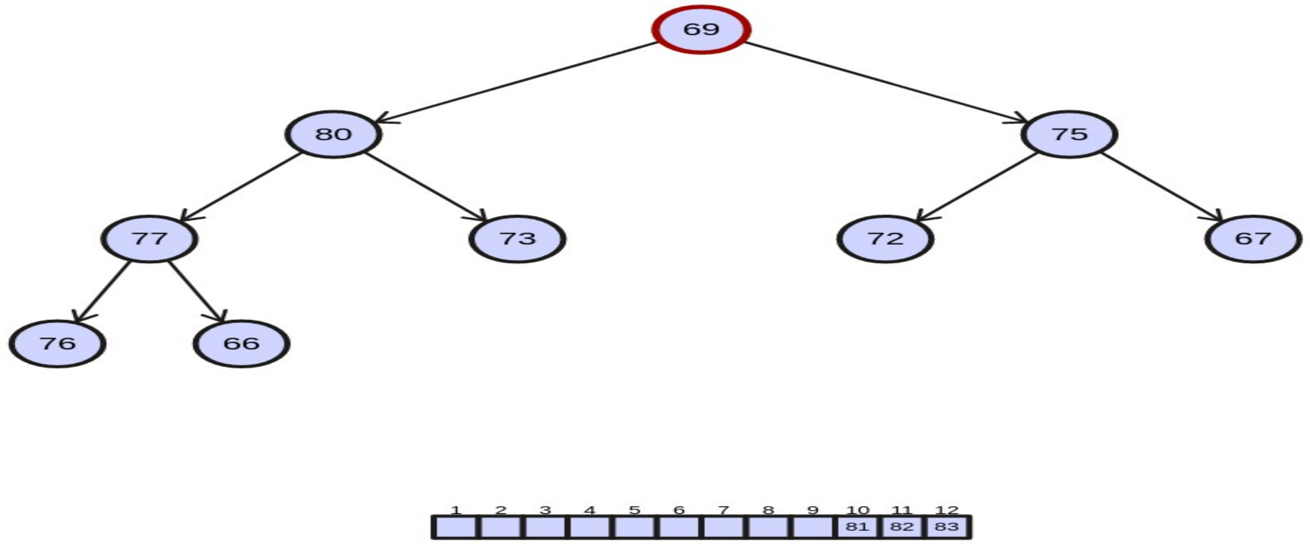


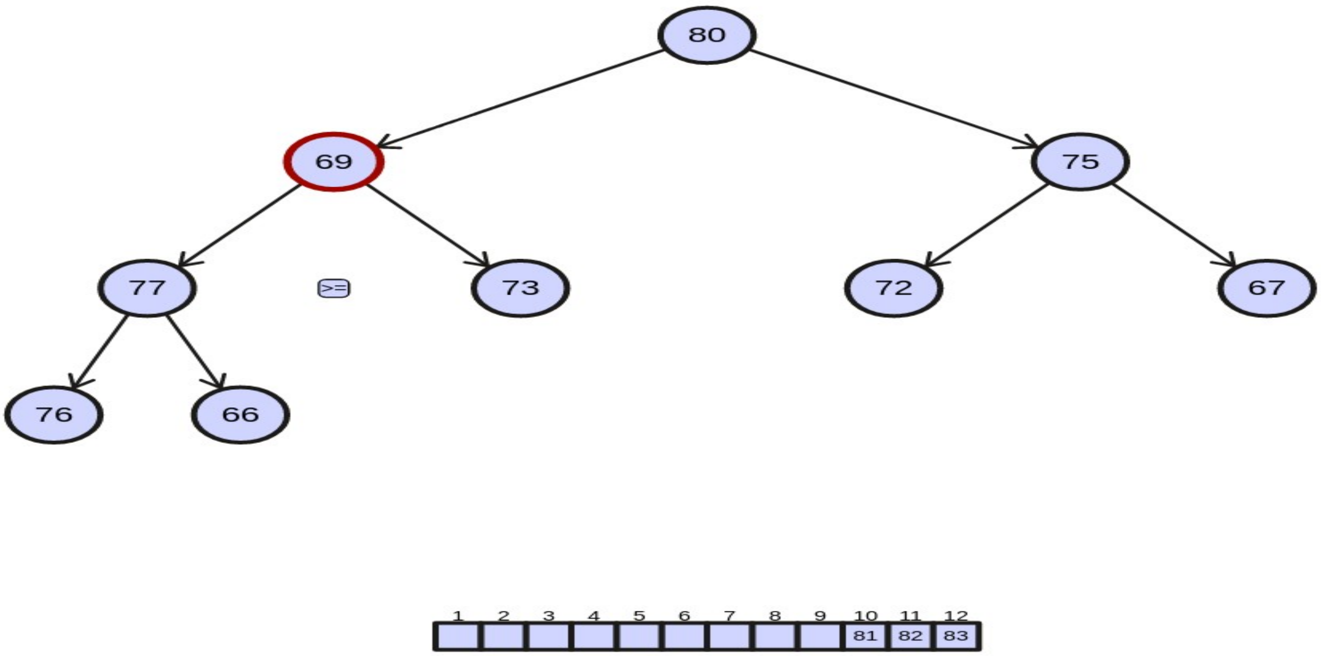
1	2	3	4	5	6	7	8	9	10	11	12
										82	83

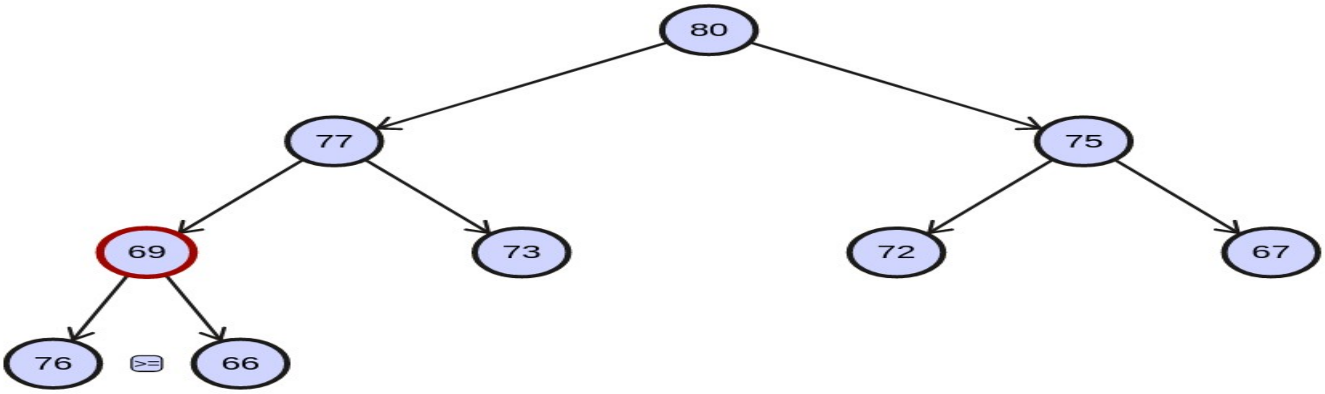


1	2	3	4	5	6	7	8	9	10	11	12
										82	83

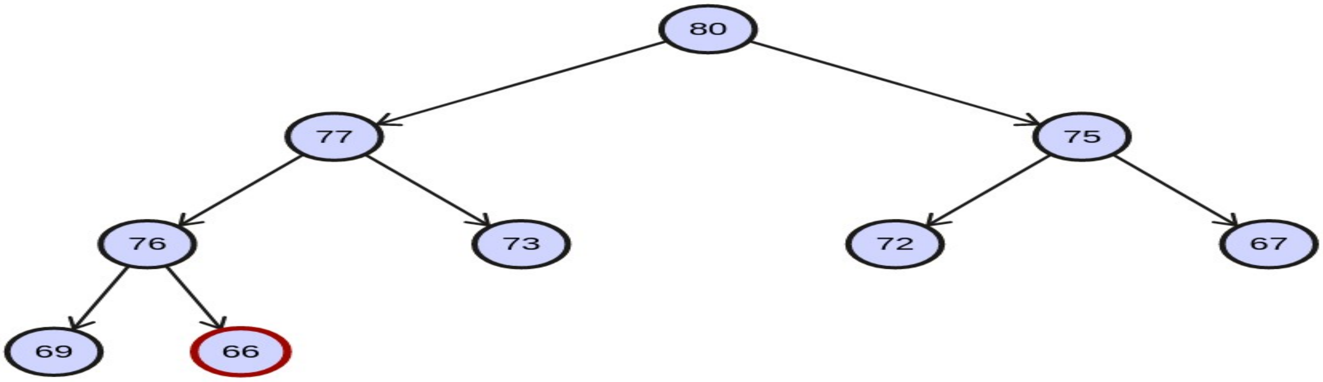

81
80
75
77
73
72
67
76
66
69
1 2 3 4 5 6 7 8 9 10 11 12
82 83


69
80
75
77
73
72
67
76
66
1 2 3 4 5 6 7 8 9 10 11 12
81 82 83

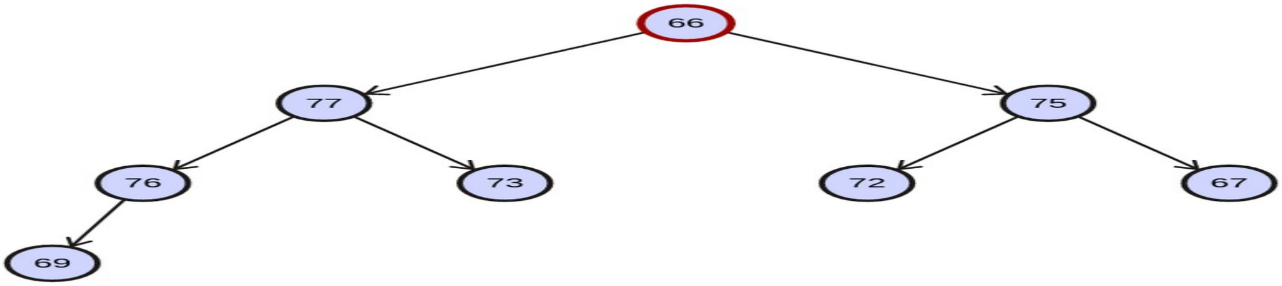

80
69
75
77
>=
73
72
67
76
66
1 2 3 4 5 6 7 8 9 10 11 12
81 82 83



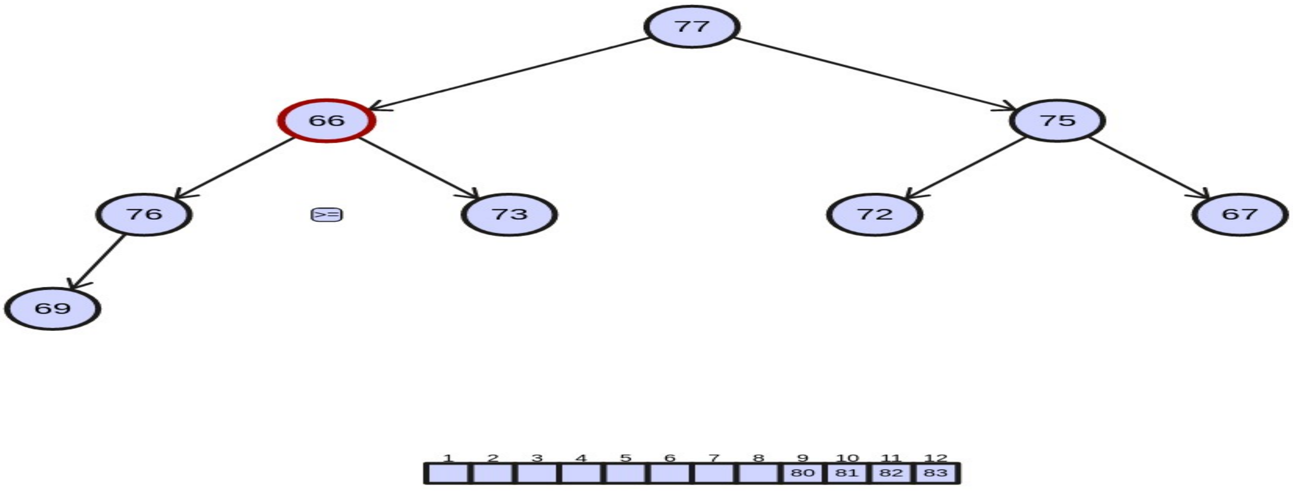
1	2	3	4	5	6	7	8	9	10	11	12
									81	82	83

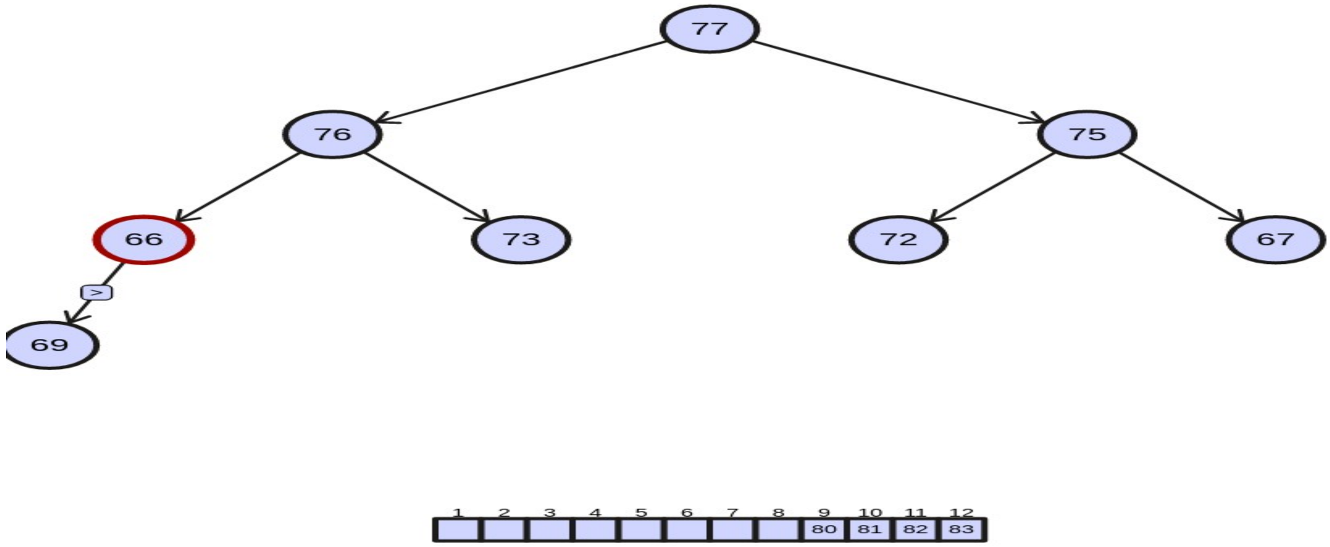


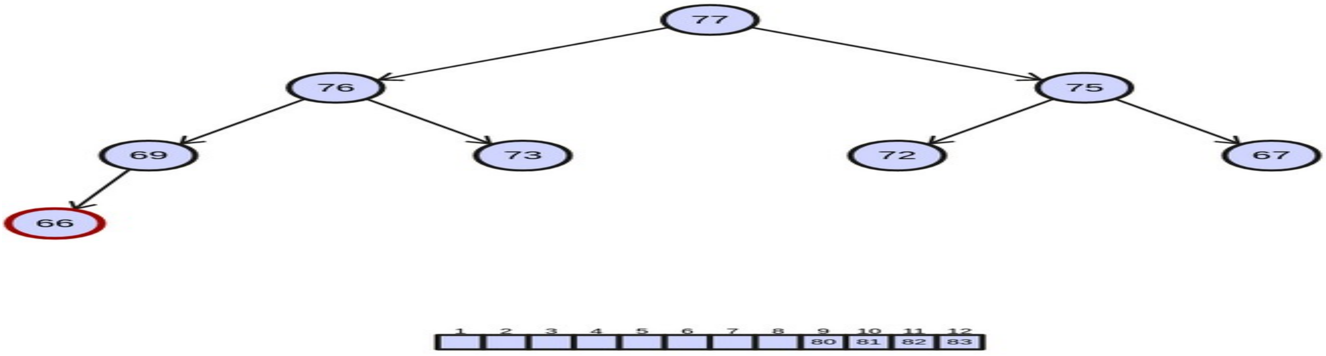
1	2	3	4	5	6	7	8	9	10	11	12
									81	82	83

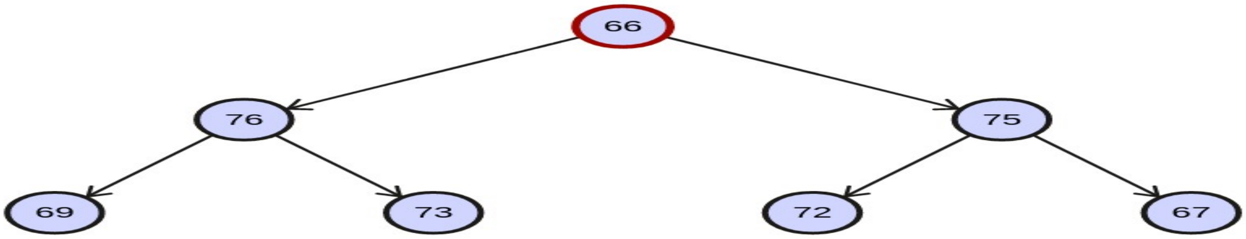


1	2	3	4	5	6	7	8	9	10	11	12
								80	81	82	83

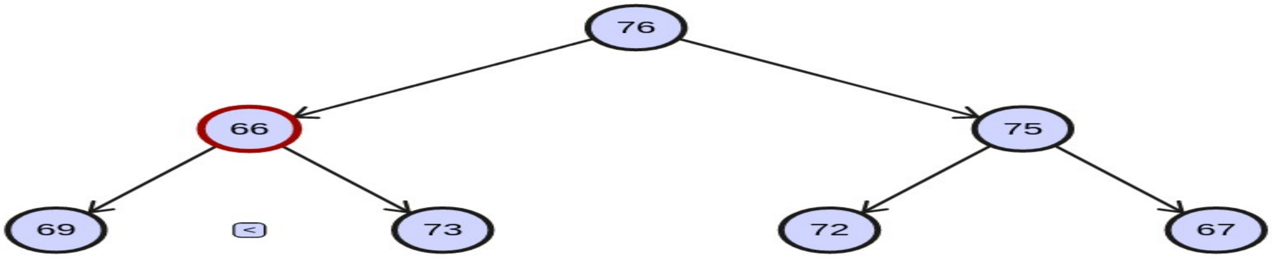

77
66
75
76
>=
73
72
67
69
1 2 3 4 5 6 7 8 9 10 11 12
80 81 82 83


77
76
75
66
73
72
67
>
69
1 2 3 4 5 6 7 8 9 10 11 12
80 81 82 83

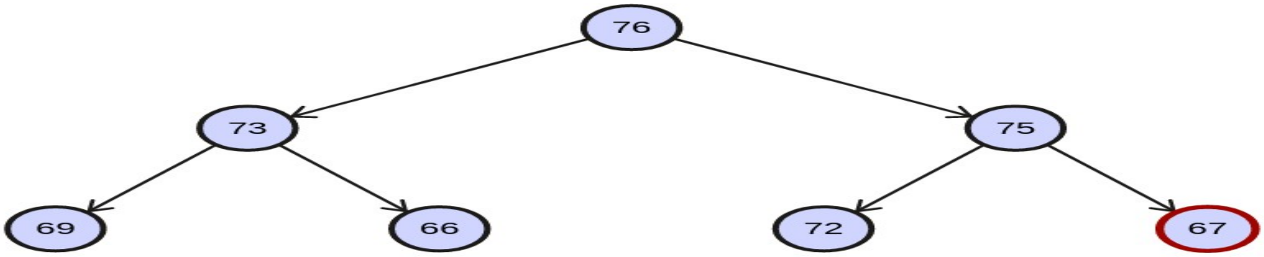

77
76
75
69
73
72
67
66
1 2 3 4 5 6 7 8 9 10 11 12
80 81 82 83



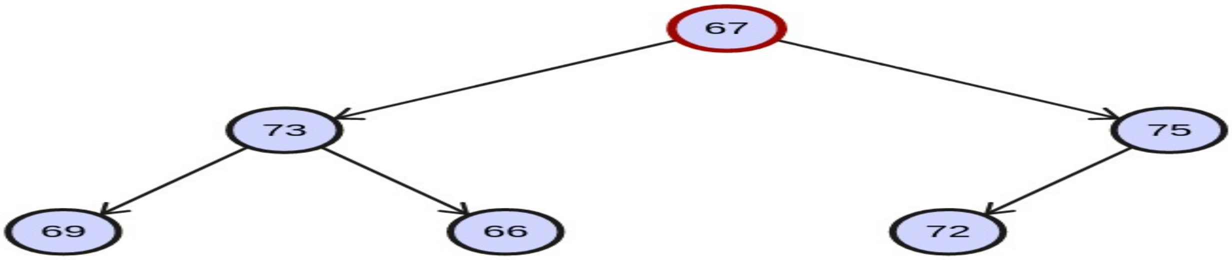
1	2	3	4	5	6	7	8	9	10	11	12
							77	80	81	82	83



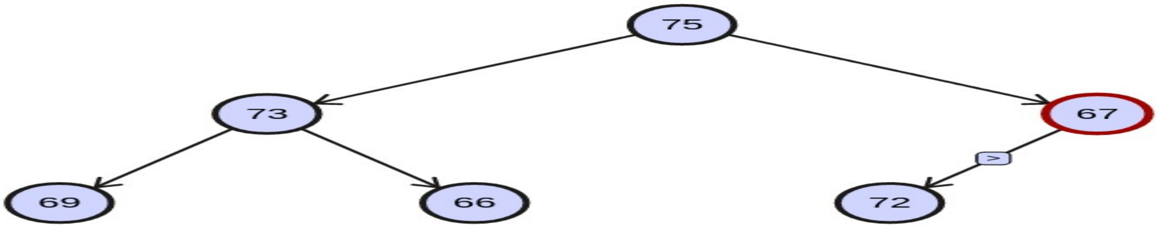
1	2	3	4	5	6	7	8	9	10	11	12
							77	80	81	82	83



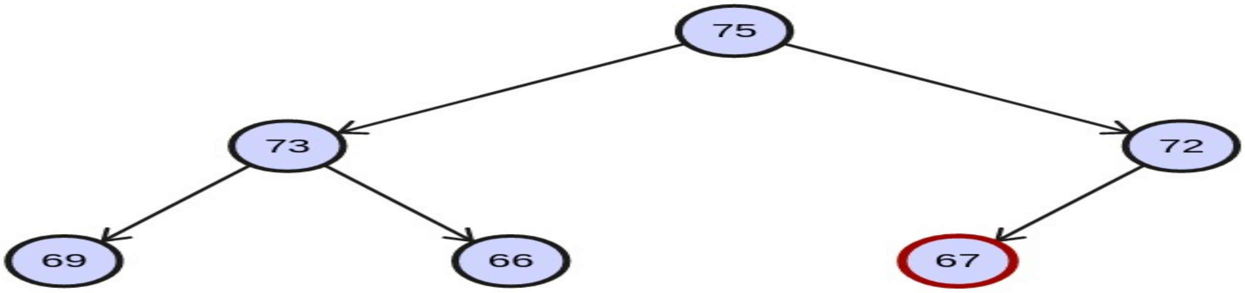
1	2	3	4	5	6	7	8	9	10	11	12
							77	80	81	82	83



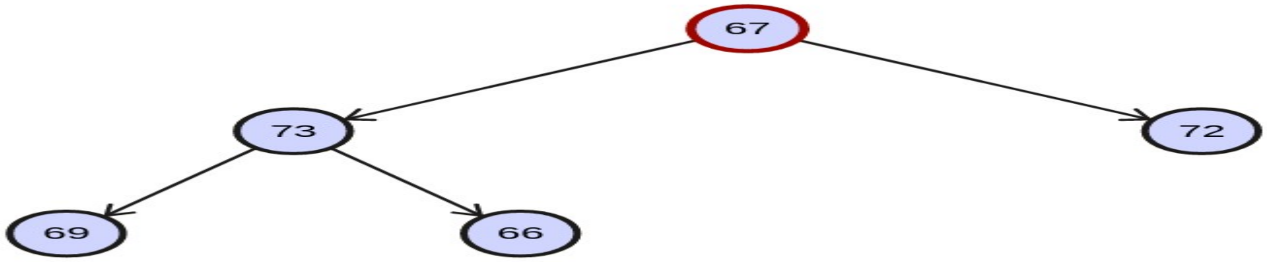
1	2	3	4	5	6	7	8	9	10	11	12
						76	77	80	81	82	83



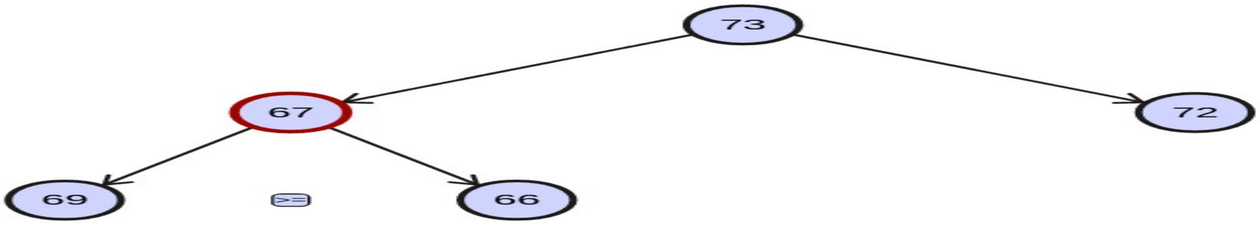
1	2	3	4	5	6	7	8	9	10	11	12
						76	77	80	81	82	83



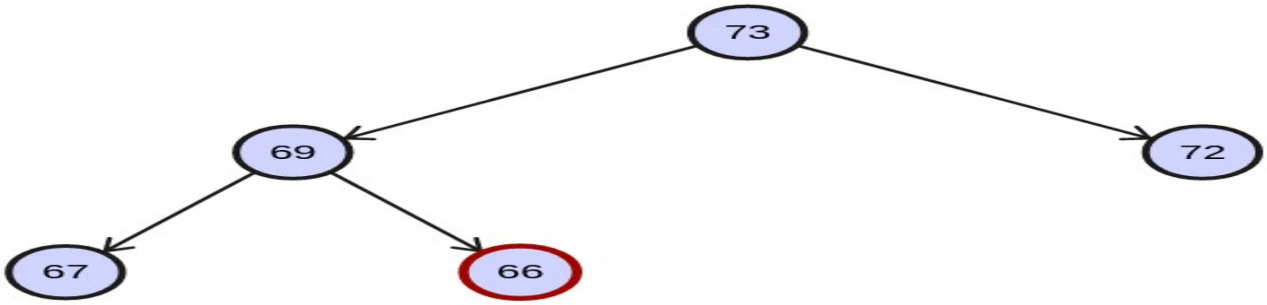
1	2	3	4	5	6	7	8	9	10	11	12
						76	77	80	81	82	83



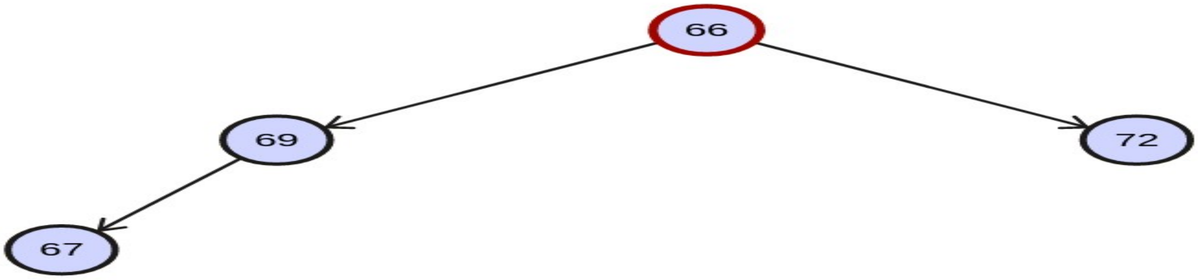
1	2	3	4	5	6	7	8	9	10	11	12
					75	76	77	80	81	82	83



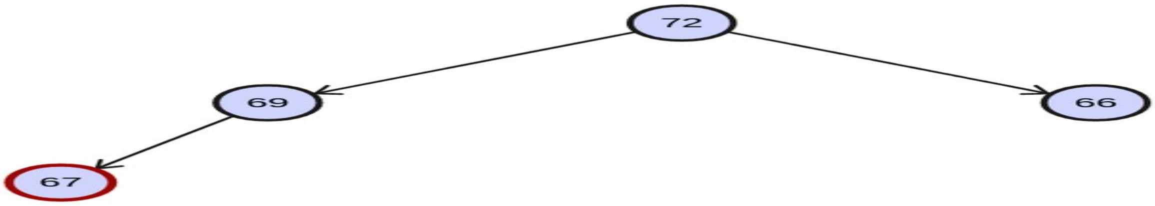
1	2	3	4	5	6	7	8	9	10	11	12
					75	76	77	80	81	82	83



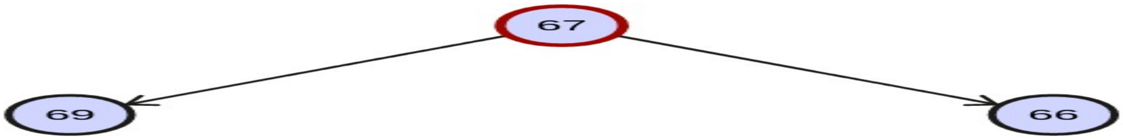
1	2	3	4	5	6	7	8	9	10	11	12
					75	76	77	80	81	82	83



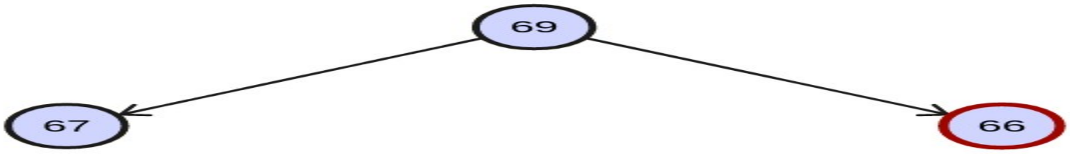
1	2	3	4	5	6	7	8	9	10	11	12
				73	75	76	77	80	81	82	83



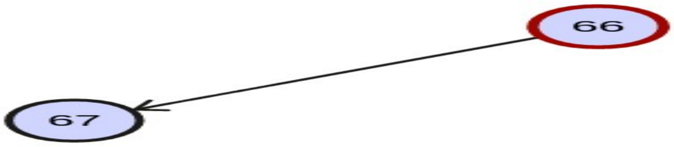
1	2	3	4	5	6	7	8	9	10	11	12
				73	75	76	77	80	81	82	83



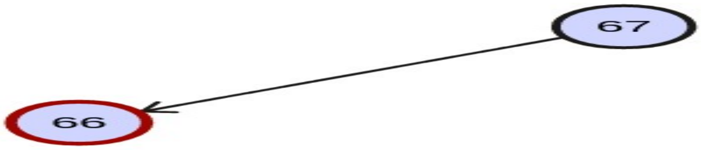
1	2	3	4	5	6	7	8	9	10	11	12
			72	73	75	76	77	80	81	82	83



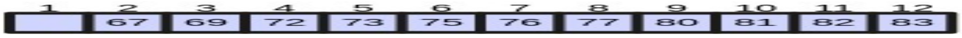
1	2	3	4	5	6	7	8	9	10	11	12
			72	73	75	76	77	80	81	82	83

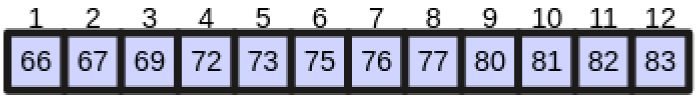


1	2	3	4	5	6	7	8	9	10	11	12
		69	72	73	75	76	77	80	81	82	83



1	2	3	4	5	6	7	8	9	10	11	12
		69	72	73	75	76	77	80	81	82	83





so we have sorted array as:

B	C	E	H	I	K	L	M	P	Q	R	S

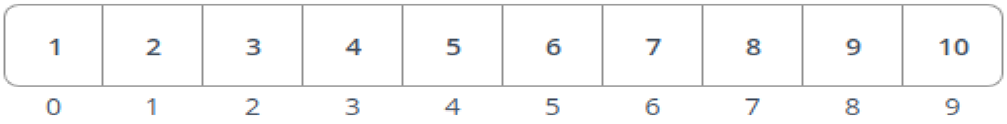
total displacement: 32

- Quick Sort

* A is an ordered integer array with 10 elements from small to large:

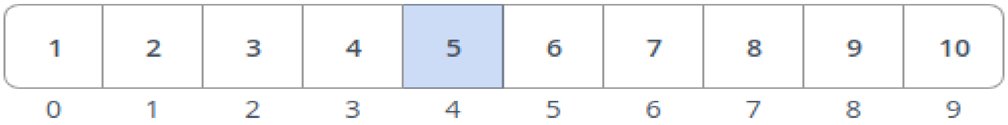
1	2	3	4	5	6	7	8	9	10

Partition the elements so that
all values less than or equal
to the pivot are to the left,
and all values greater than
the pivot are to the right



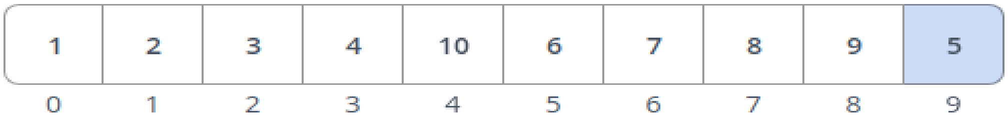
1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

> **Steps:**
> Here is a visualization of the entire Quicksort process.



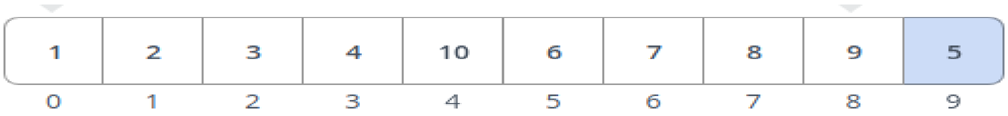
1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

> **Steps:**
> Select the pivot.



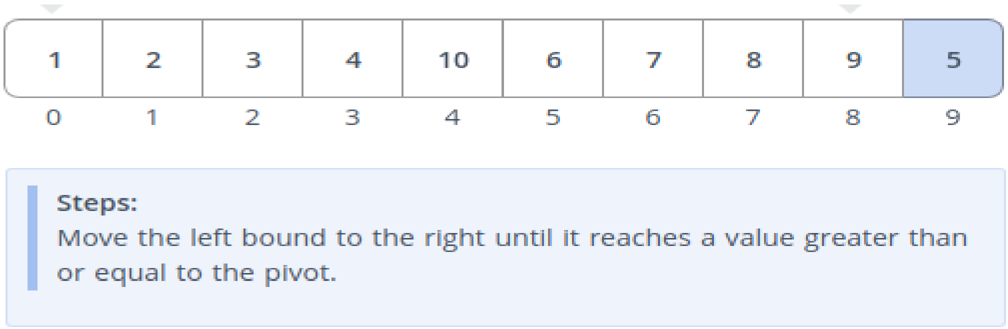
1	2	3	4	10	6	7	8	9	5
0	1	2	3	4	5	6	7	8	9

> **Steps:**
> Move the pivot to the end.



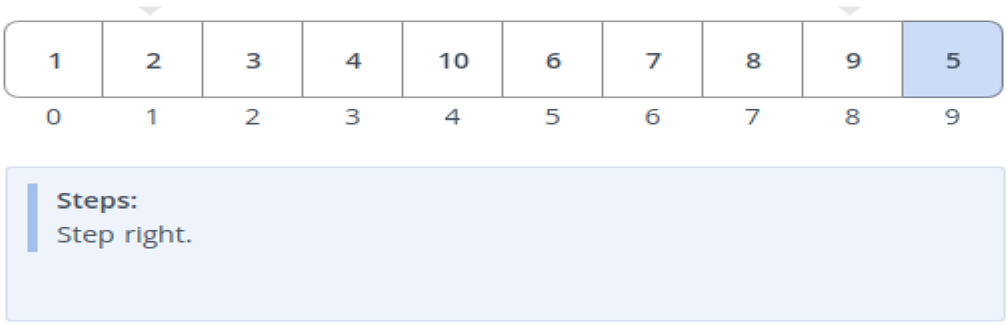
1	2	3	4	10	6	7	8	9	5
0	1	2	3	4	5	6	7	8	9

> **Steps:**
> Partition the subarray.



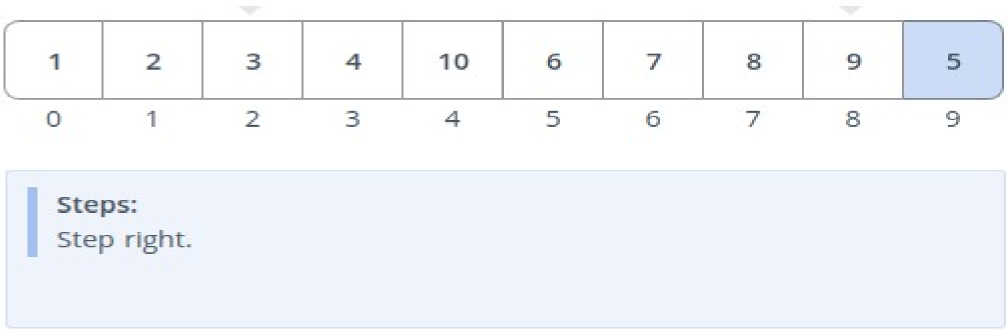
1	2	3	4	10	6	7	8	9	5
0	1	2	3	4	5	6	7	8	9

Steps:
Move the left bound to the right until it reaches a value greater than or equal to the pivot.



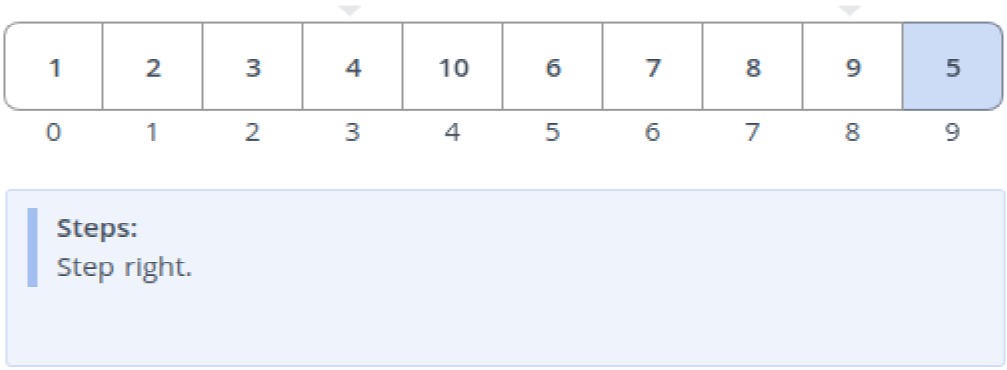
1	2	3	4	10	6	7	8	9	5
0	1	2	3	4	5	6	7	8	9

Steps:
Step right.



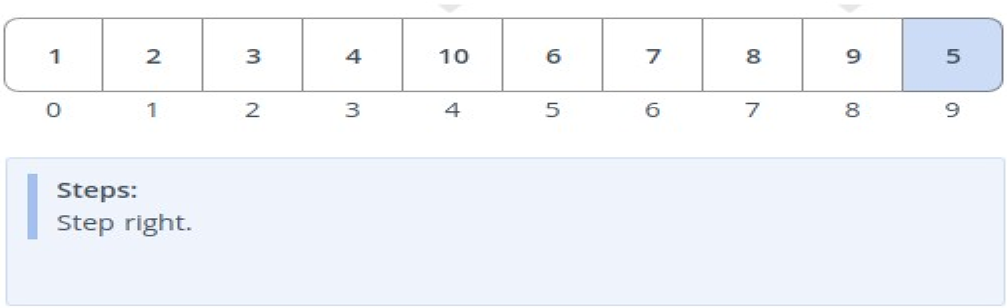
1	2	3	4	10	6	7	8	9	5
0	1	2	3	4	5	6	7	8	9

Steps:
Step right.



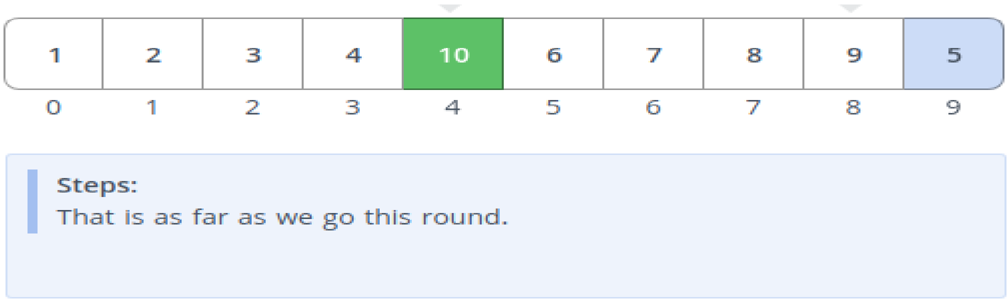
1	2	3	4	10	6	7	8	9	5
0	1	2	3	4	5	6	7	8	9

Steps:
Step right.



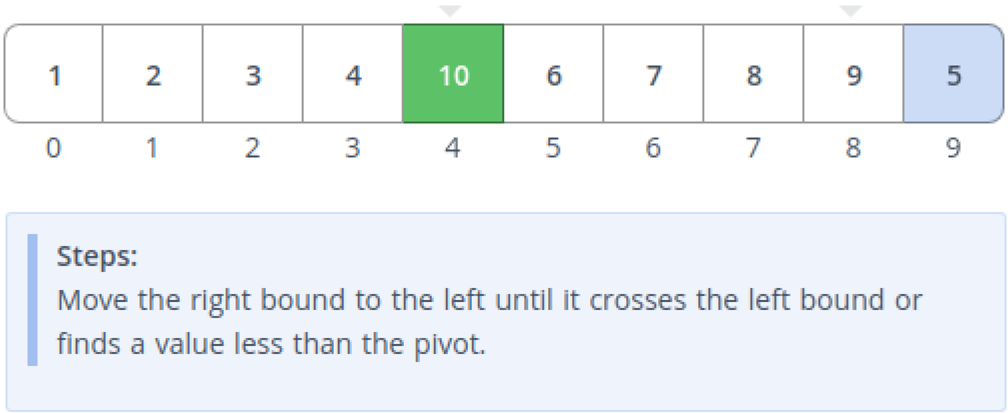
1	2	3	4	10	6	7	8	9	5
0	1	2	3	4	5	6	7	8	9

Steps:
Step right.



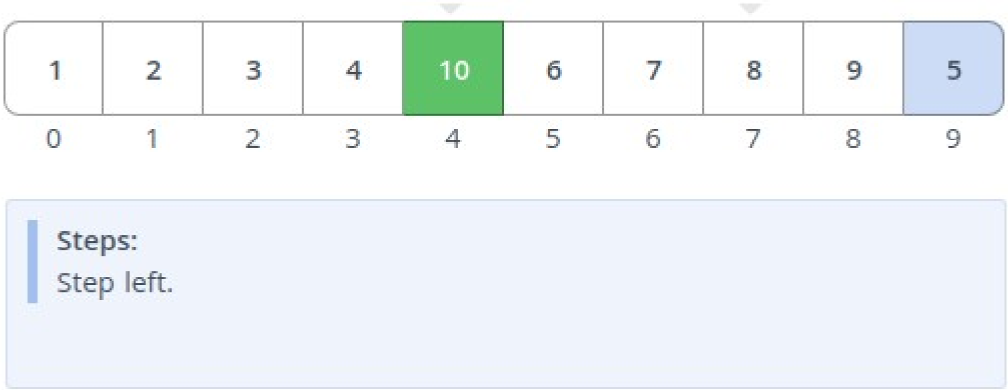
1	2	3	4	10	6	7	8	9	5
0	1	2	3	4	5	6	7	8	9

Steps:
That is as far as we go this round.



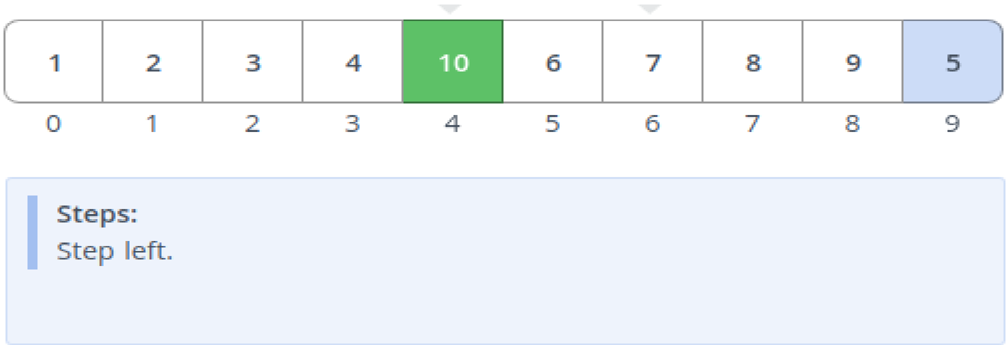
1	2	3	4	10	6	7	8	9	5
0	1	2	3	4	5	6	7	8	9

Steps:
Move the right bound to the left until it crosses the left bound or finds a value less than the pivot.



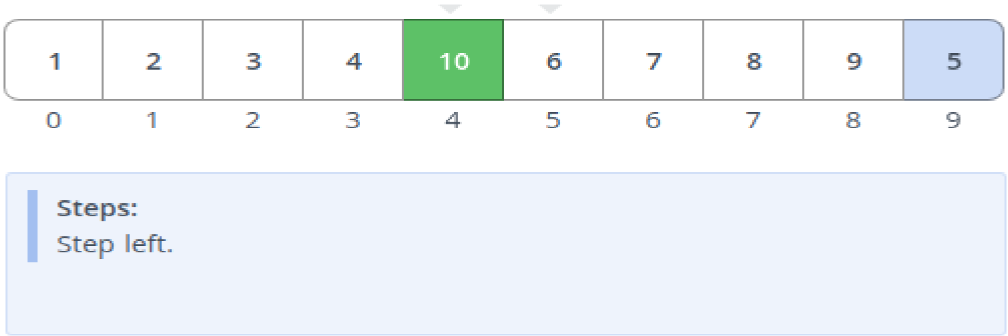
1	2	3	4	10	6	7	8	9	5
0	1	2	3	4	5	6	7	8	9

Steps:
Step left.



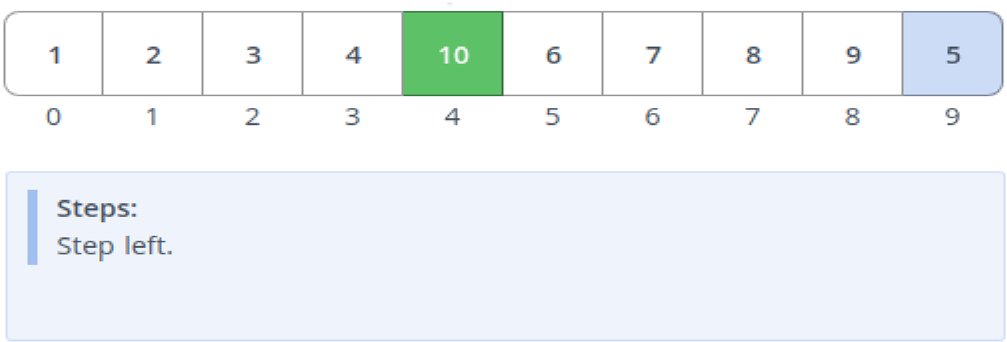
1	2	3	4	10	6	7	8	9	5
0	1	2	3	4	5	6	7	8	9

Steps:
Step left.



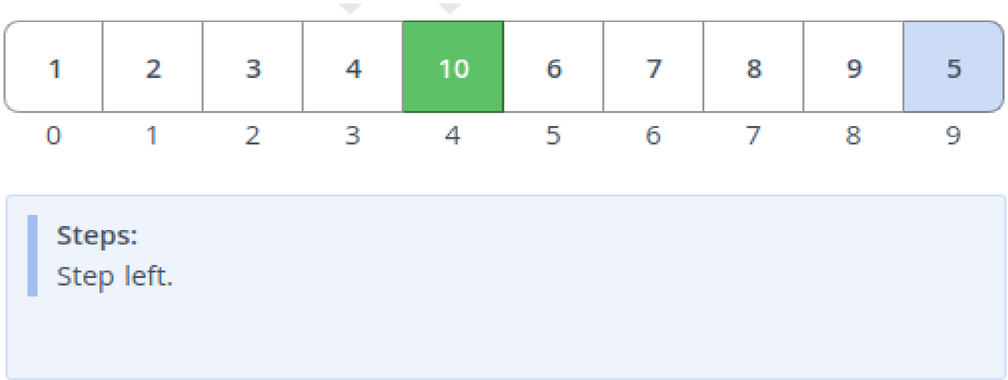
1	2	3	4	10	6	7	8	9	5
0	1	2	3	4	5	6	7	8	9

Steps:
Step left.



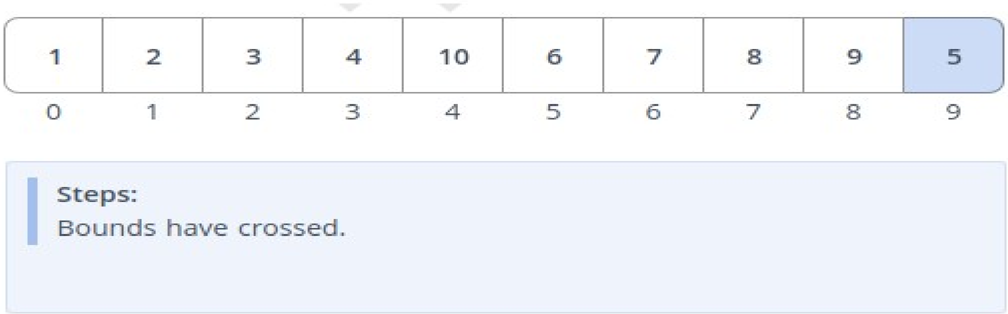
1	2	3	4	10	6	7	8	9	5
0	1	2	3	4	5	6	7	8	9

Steps:
Step left.



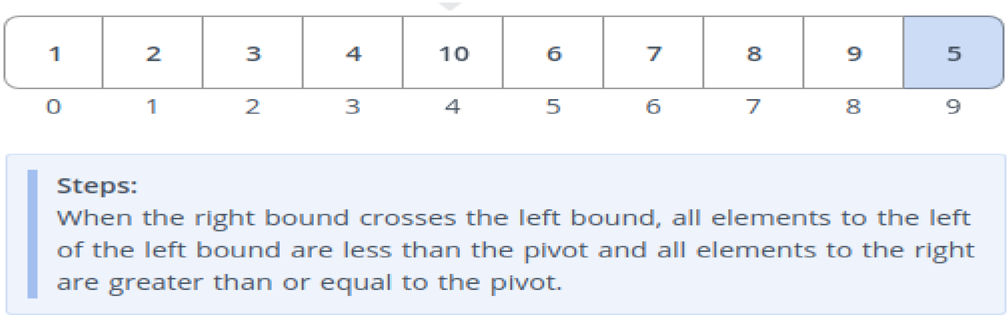
1	2	3	4	10	6	7	8	9	5
0	1	2	3	4	5	6	7	8	9

Steps:
Step left.



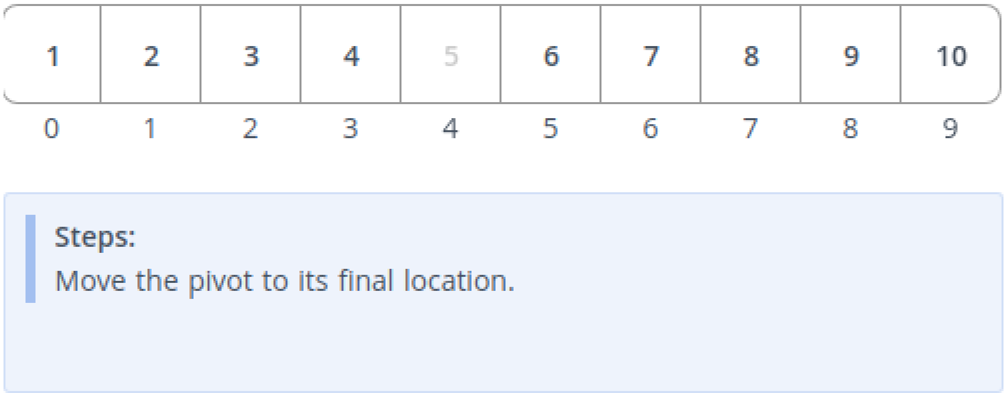
1	2	3	4	10	6	7	8	9	5
0	1	2	3	4	5	6	7	8	9

Steps:
Bounds have crossed.



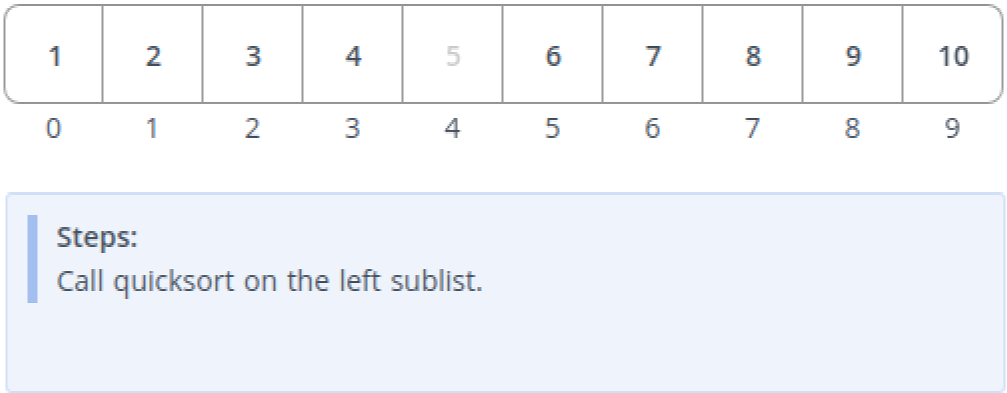
1	2	3	4	10	6	7	8	9	5
0	1	2	3	4	5	6	7	8	9

Steps:
When the right bound crosses the left bound, all elements to the left of the left bound are less than the pivot and all elements to the right are greater than or equal to the pivot.



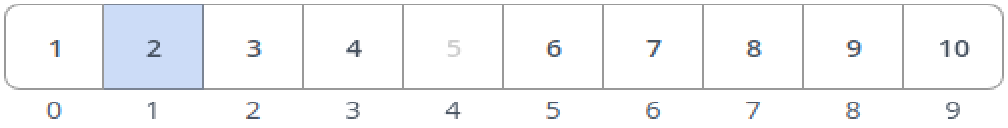
1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Move the pivot to its final location.



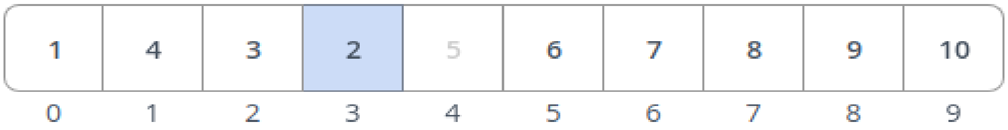
1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Call quicksort on the left sublist.



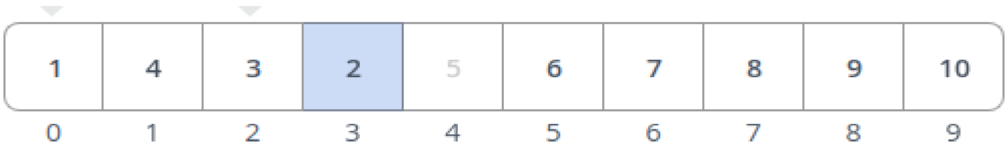
1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

> **Steps:**
> Select the pivot.



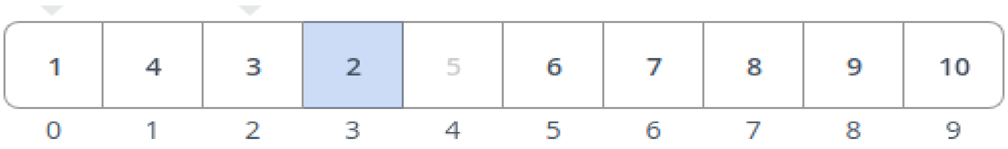
1	4	3	2	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

> **Steps:**
> Move the pivot to the end.



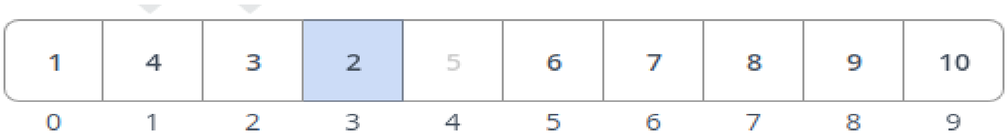
1	4	3	2	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

> **Steps:**
> Partition the subarray.



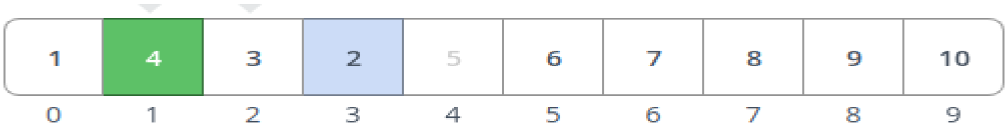
1	4	3	2	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

> **Steps:**
> Move the left bound to the right until it reaches a value greater than or equal to the pivot.



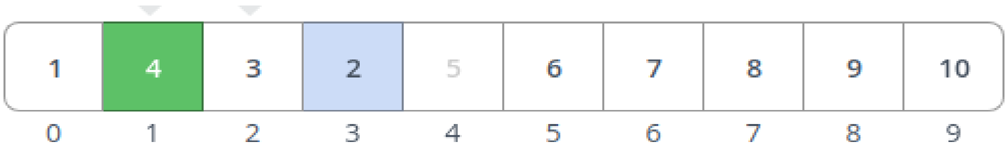
1	4	3	2	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Step right.



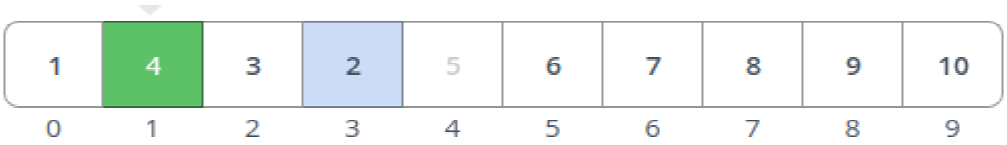
1	4	3	2	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
That is as far as we go this round.



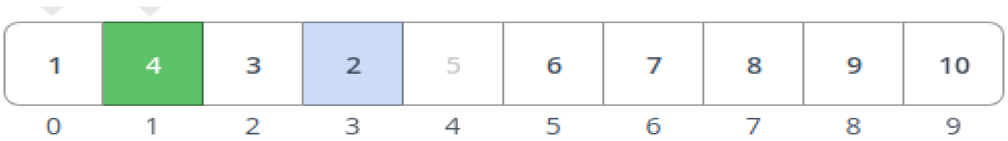
1	4	3	2	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Move the right bound to the left until it crosses the left bound or finds a value less than the pivot.



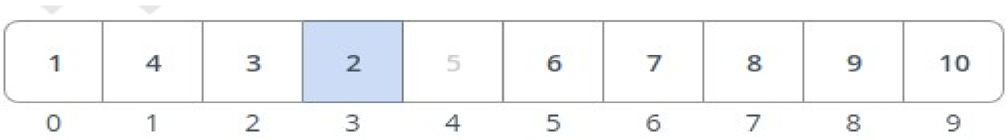
1	4	3	2	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Step left.



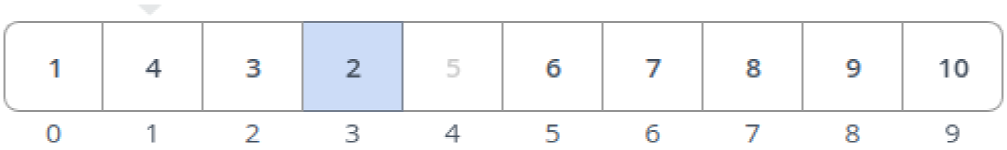
1	4	3	2	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Step left.



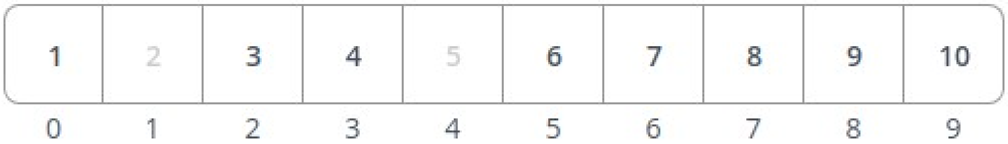
1	4	3	2	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Bounds have crossed.



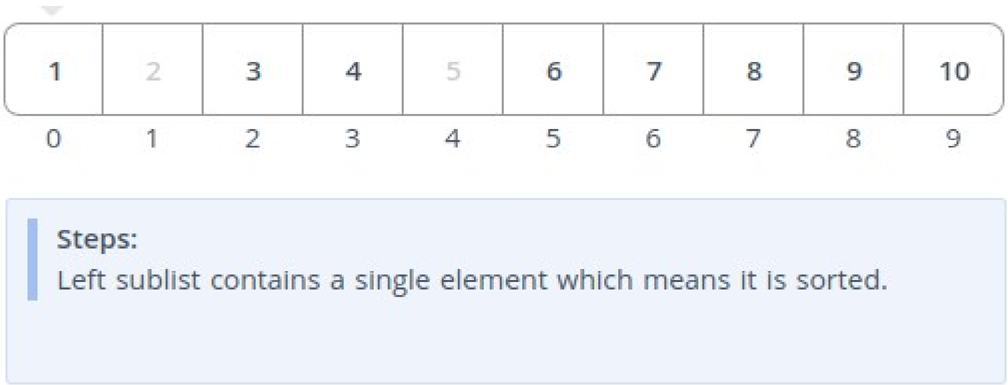
1	4	3	2	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
When the right bound crosses the left bound, all elements to the left of the left bound are less than the pivot and all elements to the right are greater than or equal to the pivot.



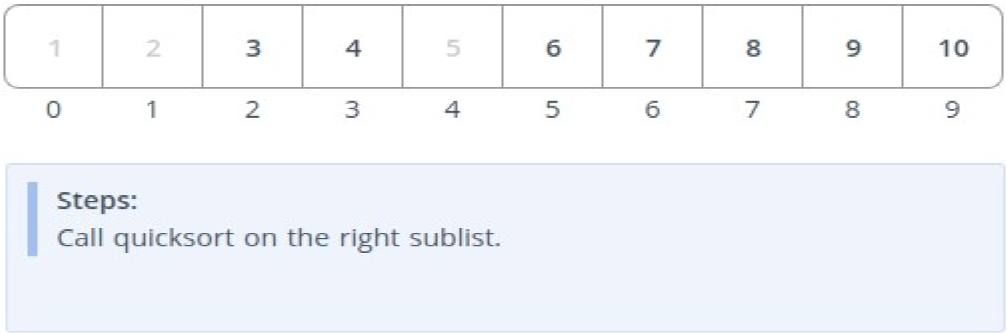
1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Move the pivot to its final location.



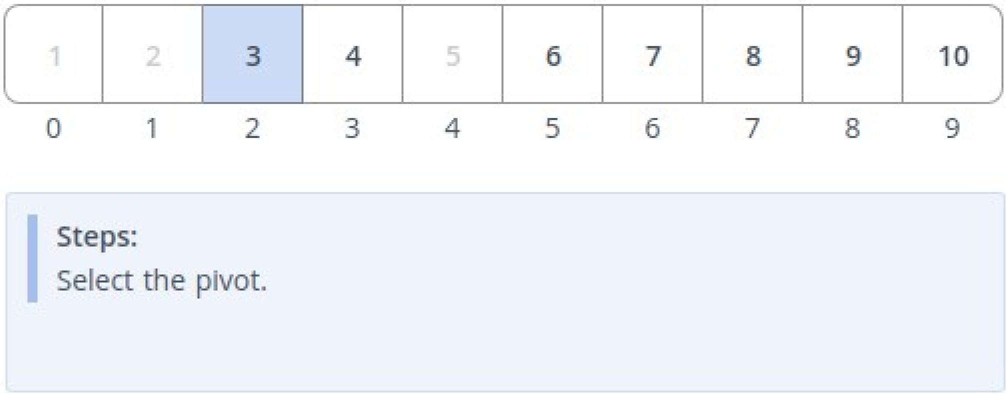
1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Left sublist contains a single element which means it is sorted.



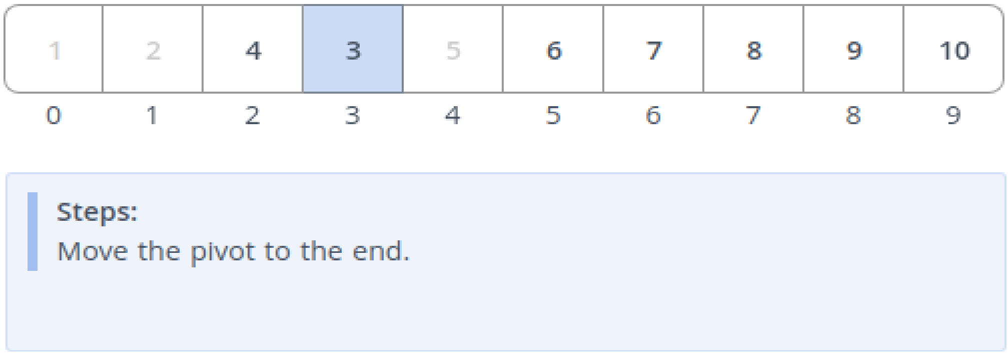
1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Call quicksort on the right sublist.



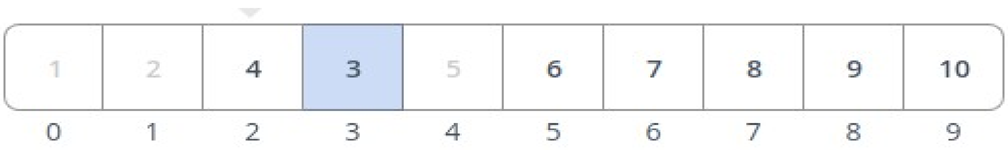
1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Select the pivot.



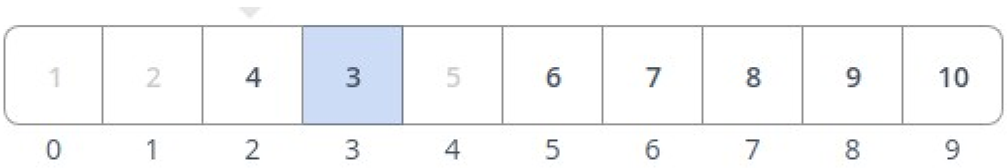
1	2	4	3	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Move the pivot to the end.



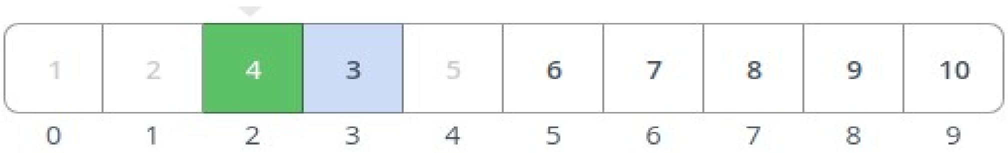
1	2	4	3	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

> **Steps:**
> Partition the subarray.



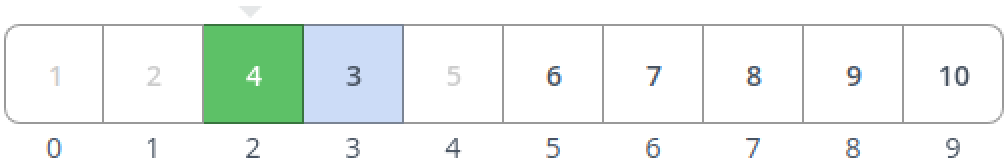
1	2	4	3	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

> **Steps:**
> Move the left bound to the right until it reaches a value greater than or equal to the pivot.



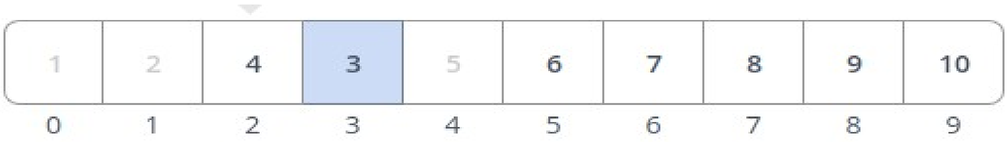
1	2	4	3	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

> **Steps:**
> That is as far as we go this round.



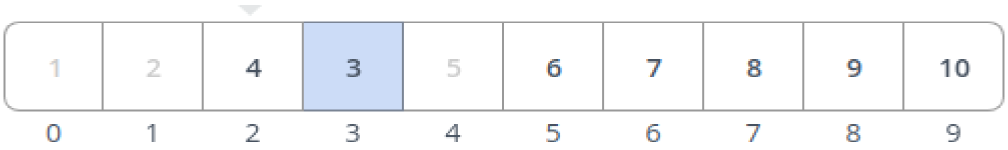
1	2	4	3	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

> **Steps:**
> Move the right bound to the left until it crosses the left bound or finds a value less than the pivot.



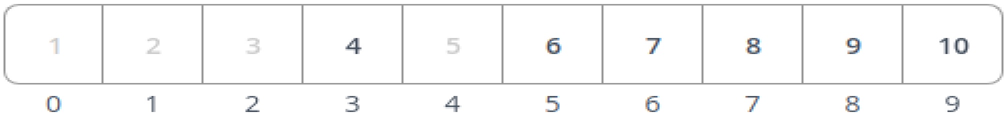
1	2	4	3	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Bounds have crossed.



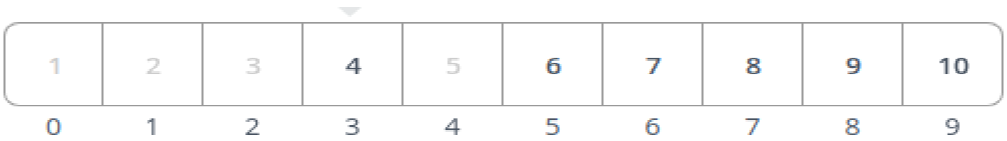
1	2	4	3	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
When the right bound crosses the left bound, all elements to the left of the left bound are less than the pivot and all elements to the right are greater than or equal to the pivot.



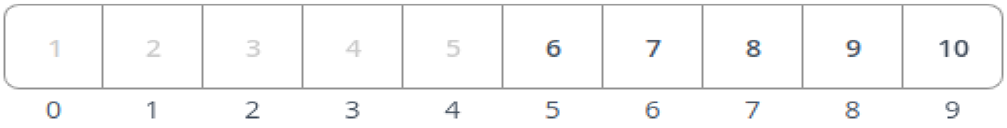
1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Move the pivot to its final location.



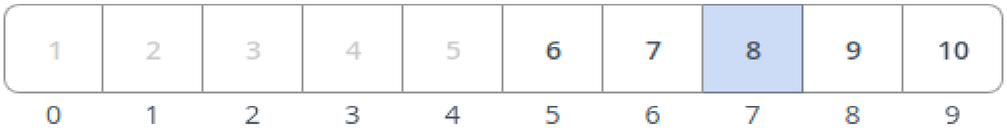
1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Right sublist contains a single element which means it is sorted.



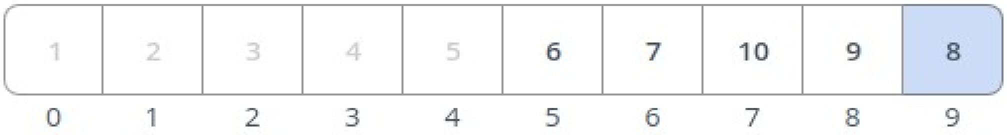
1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Call quicksort on the right sublist.



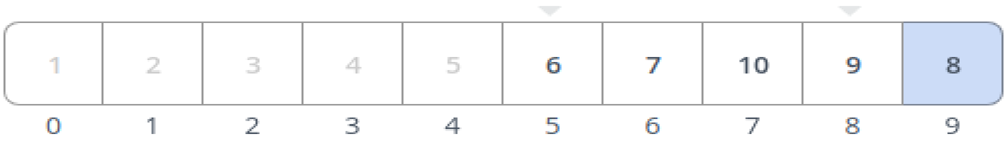
1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Select the pivot.



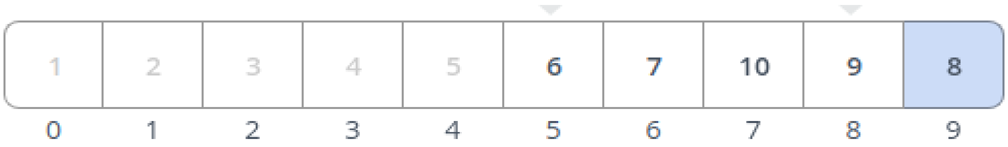
1	2	3	4	5	6	7	10	9	8
0	1	2	3	4	5	6	7	8	9

Steps:
Move the pivot to the end.



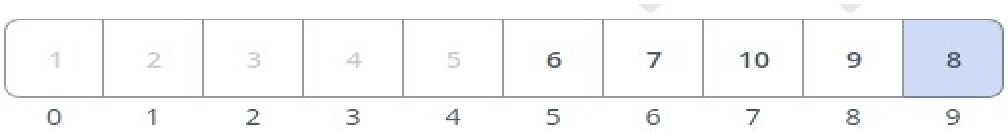
1	2	3	4	5	6	7	10	9	8
0	1	2	3	4	5	6	7	8	9

Steps:
Partition the subarray.



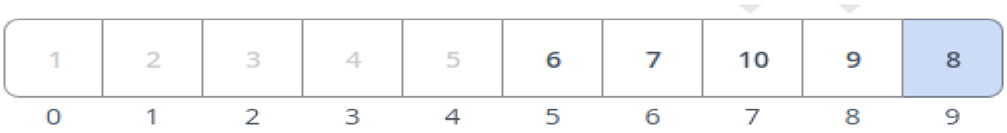
1	2	3	4	5	6	7	10	9	8
0	1	2	3	4	5	6	7	8	9

Steps:
Move the left bound to the right until it reaches a value greater than or equal to the pivot.



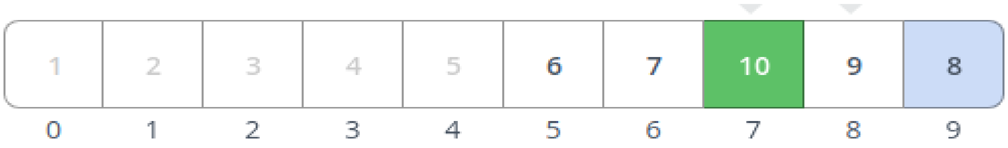
1	2	3	4	5	6	7	10	9	8
0	1	2	3	4	5	6	7	8	9

Steps:
Step right.



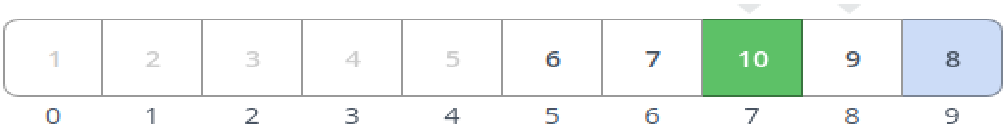
1	2	3	4	5	6	7	10	9	8
0	1	2	3	4	5	6	7	8	9

Steps:
Step right.



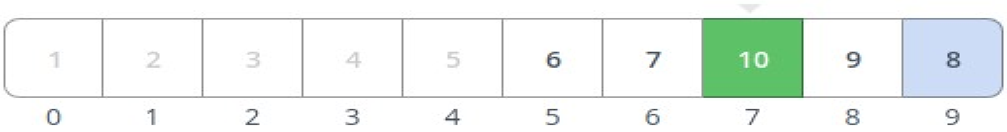
1	2	3	4	5	6	7	10	9	8
0	1	2	3	4	5	6	7	8	9

Steps:
That is as far as we go this round.



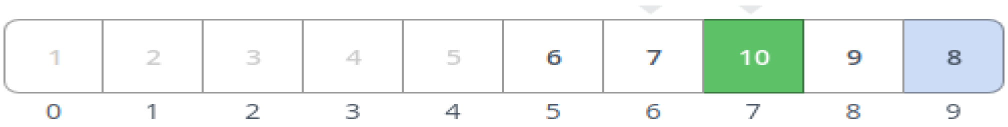
1	2	3	4	5	6	7	10	9	8
0	1	2	3	4	5	6	7	8	9

> **Steps:**
> Move the right bound to the left until it crosses the left bound or finds a value less than the pivot.



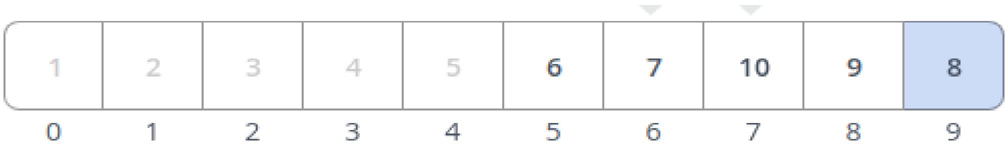
1	2	3	4	5	6	7	10	9	8
0	1	2	3	4	5	6	7	8	9

> **Steps:**
> Step left.



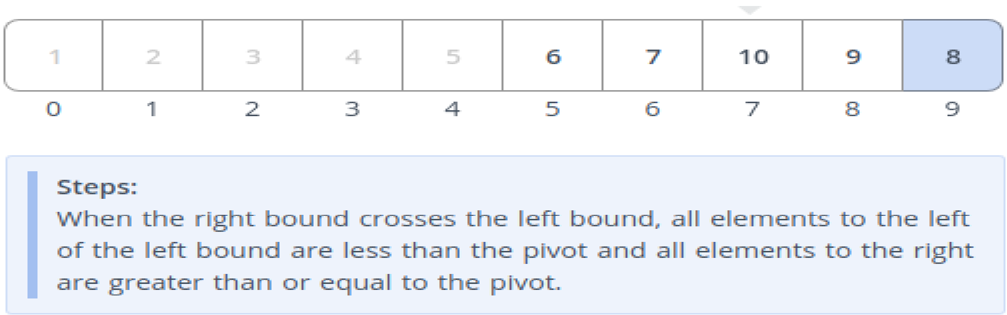
1	2	3	4	5	6	7	10	9	8
0	1	2	3	4	5	6	7	8	9

> **Steps:**
> Step left.



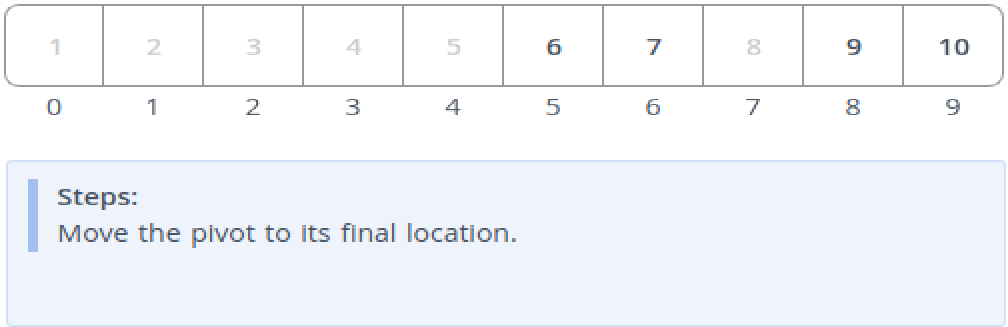
1	2	3	4	5	6	7	10	9	8
0	1	2	3	4	5	6	7	8	9

> **Steps:**
> Bounds have crossed.



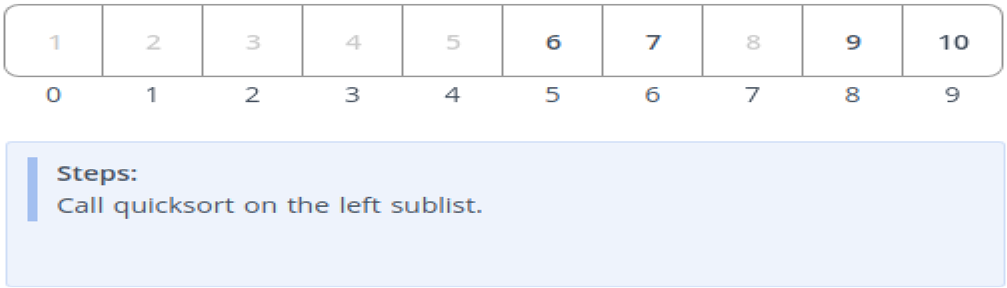
1	2	3	4	5	6	7	10	9	8
0	1	2	3	4	5	6	7	8	9

Steps:
When the right bound crosses the left bound, all elements to the left of the left bound are less than the pivot and all elements to the right are greater than or equal to the pivot.



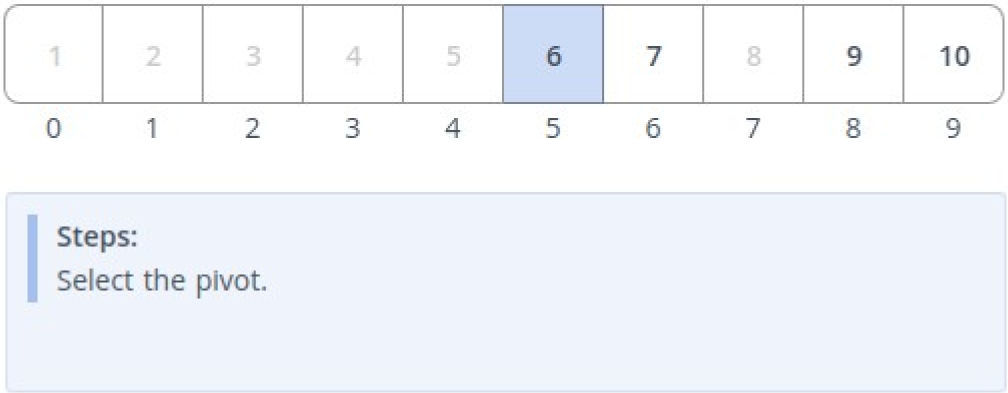
1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Move the pivot to its final location.



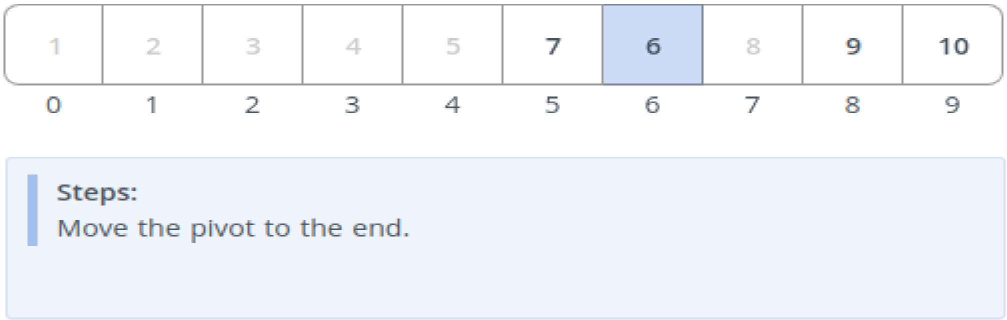
1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Call quicksort on the left sublist.



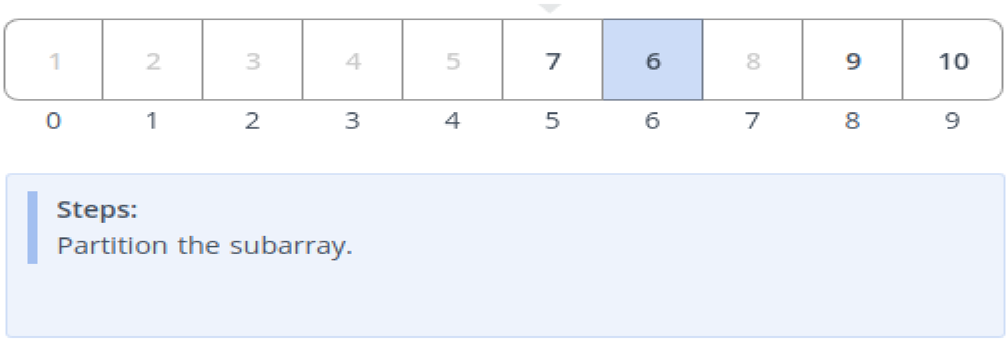
1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Select the pivot.



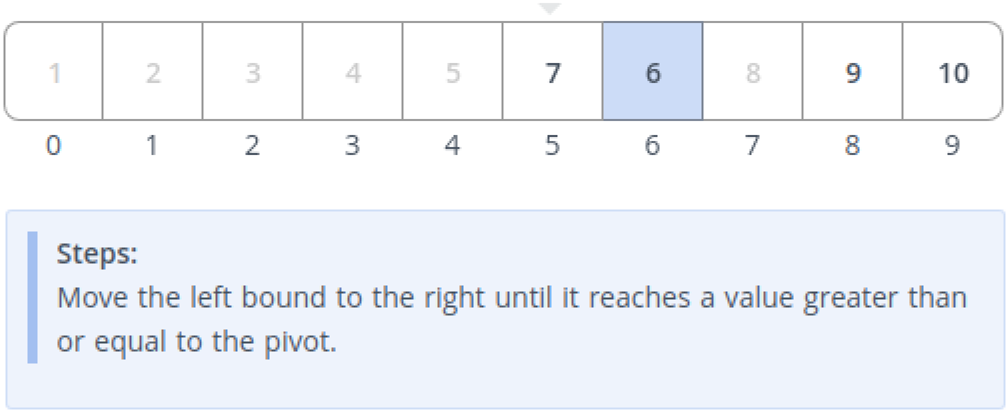
1	2	3	4	5	7	6	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Move the pivot to the end.



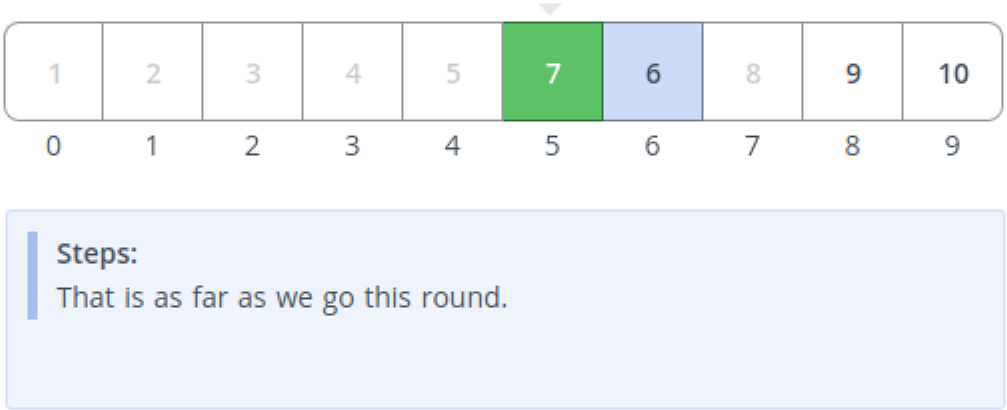
1	2	3	4	5	7	6	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Partition the subarray.



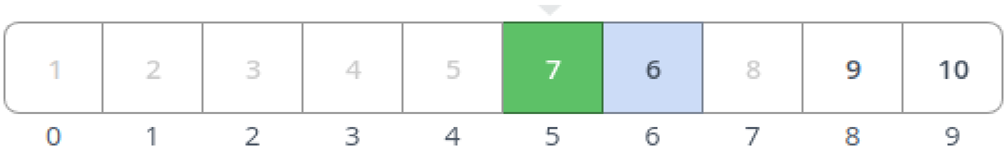
1	2	3	4	5	7	6	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Move the left bound to the right until it reaches a value greater than or equal to the pivot.



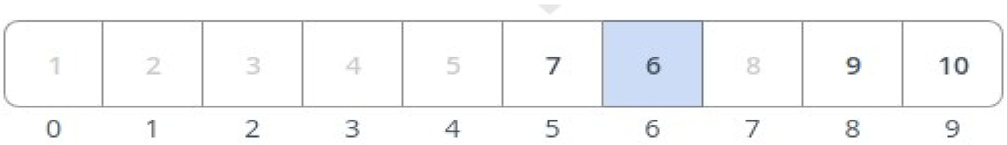
1	2	3	4	5	7	6	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
That is as far as we go this round.



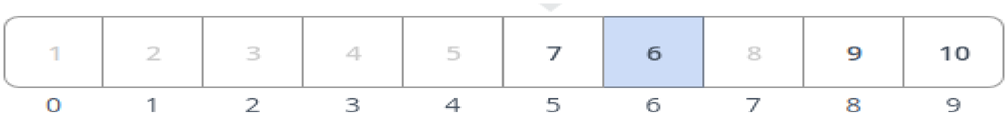
1	2	3	4	5	7	6	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Move the right bound to the left until it crosses the left bound or finds a value less than the pivot.



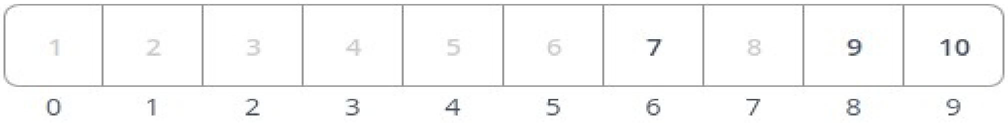
1	2	3	4	5	7	6	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Bounds have crossed.



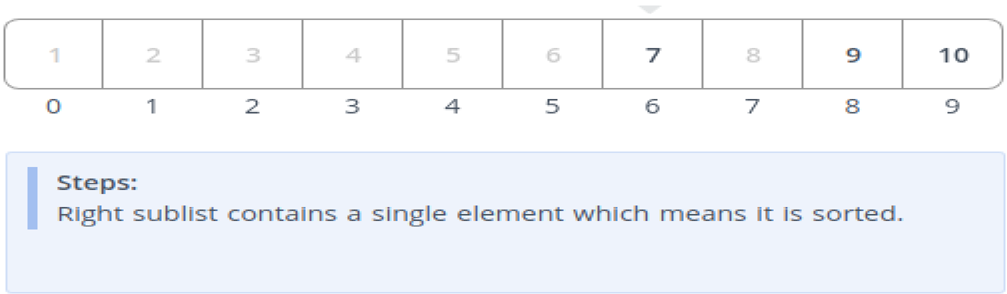
1	2	3	4	5	7	6	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
When the right bound crosses the left bound, all elements to the left of the left bound are less than the pivot and all elements to the right are greater than or equal to the pivot.



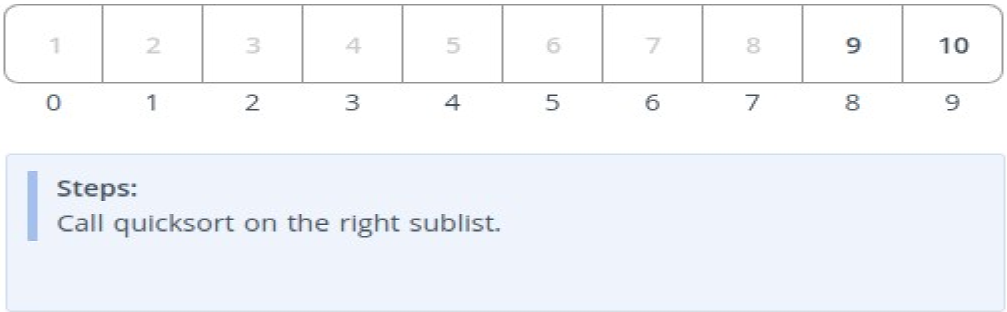
1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Move the pivot to its final location.



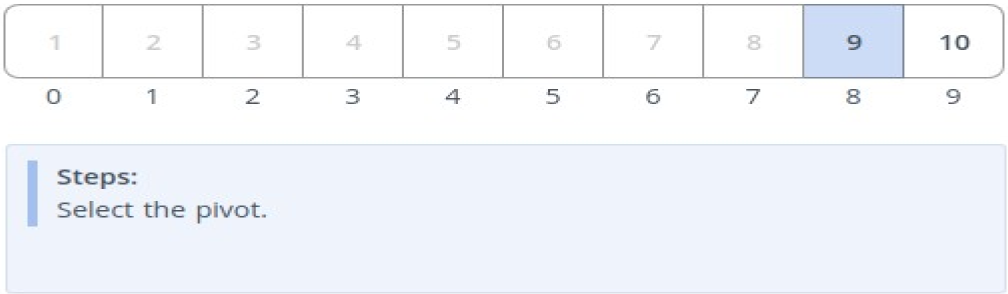
1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Right sublist contains a single element which means it is sorted.



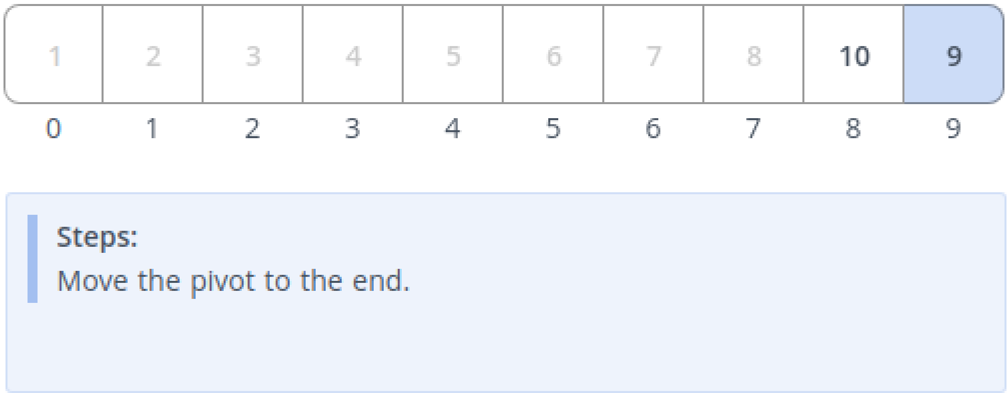
1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Call quicksort on the right sublist.



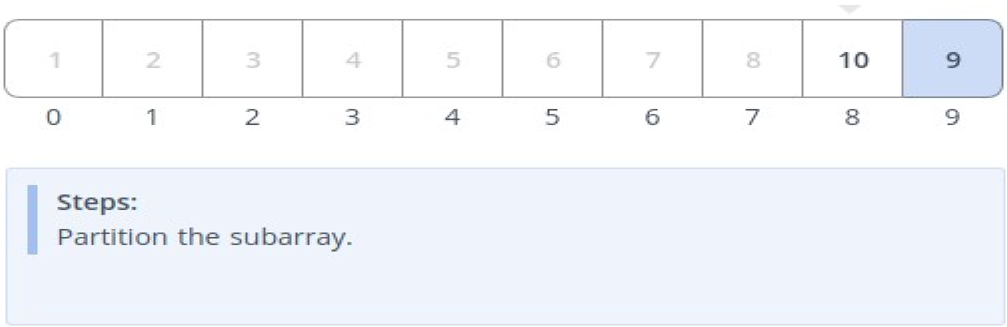
1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Select the pivot.



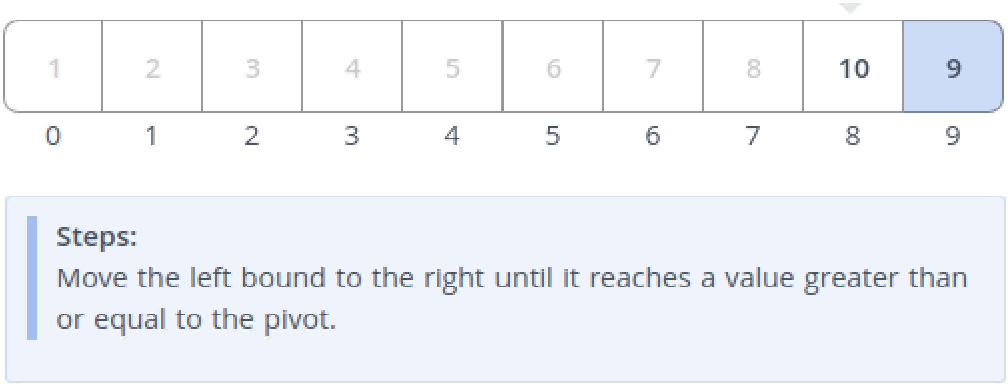
1	2	3	4	5	6	7	8	10	9
0	1	2	3	4	5	6	7	8	9

Steps:
Move the pivot to the end.



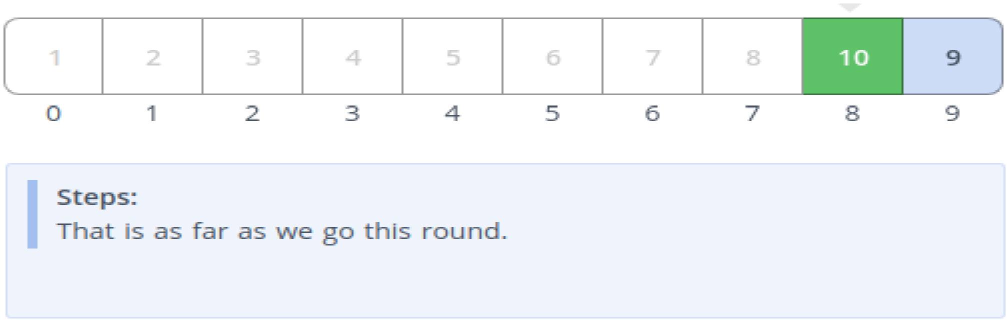
1	2	3	4	5	6	7	8	10	9
0	1	2	3	4	5	6	7	8	9

Steps:
Partition the subarray.



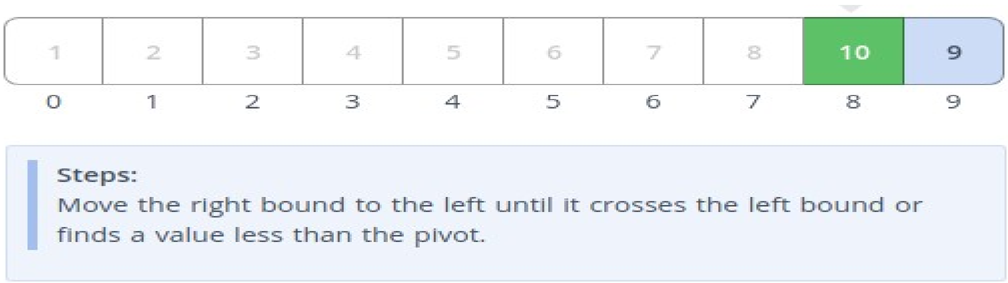
1	2	3	4	5	6	7	8	10	9
0	1	2	3	4	5	6	7	8	9

Steps:
Move the left bound to the right until it reaches a value greater than or equal to the pivot.



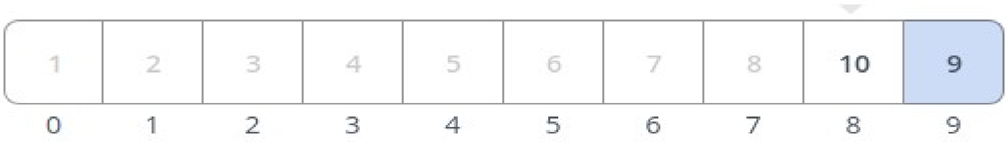
1	2	3	4	5	6	7	8	10	9
0	1	2	3	4	5	6	7	8	9

Steps:
That is as far as we go this round.



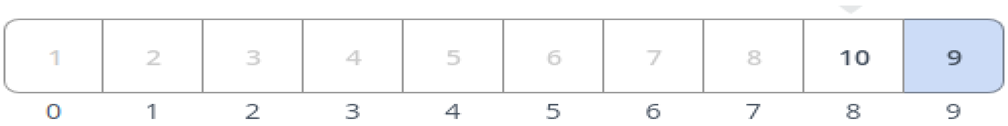
1	2	3	4	5	6	7	8	10	9
0	1	2	3	4	5	6	7	8	9

Steps:
Move the right bound to the left until it crosses the left bound or finds a value less than the pivot.



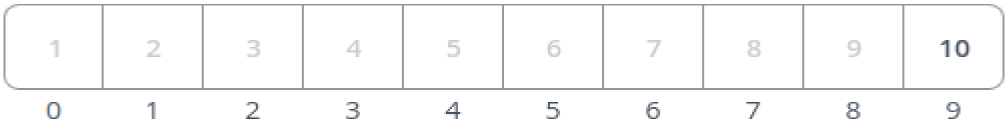
1	2	3	4	5	6	7	8	10	9
0	1	2	3	4	5	6	7	8	9

Steps:
Bounds have crossed.



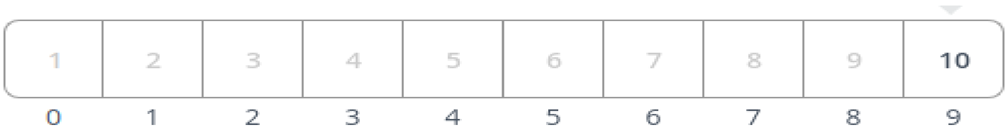
1	2	3	4	5	6	7	8	10	9
0	1	2	3	4	5	6	7	8	9

Steps:
When the right bound crosses the left bound, all elements to the left of the left bound are less than the pivot and all elements to the right are greater than or equal to the pivot.



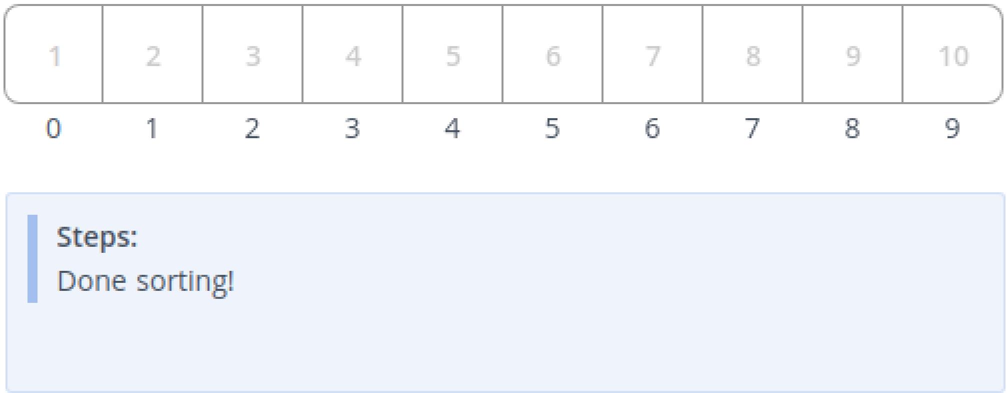
1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Move the pivot to its final location.



1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Right sublist contains a single element which means it is sorted.



1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Steps:
Done sorting!

total steps = 80
* B is an ordered integer array with 10 elements from large to small
* C = {5, 2, 13, 9, 1, 7, 6, 8, 1, 15, 4, 11}
* D = {‘S’, ‘B’, ‘I’, ‘M’, ‘H’, ‘Q’, ‘C’, ‘L’, ‘R’, ‘E’, ‘P’, ‘K’}

THESE 3 Arrays is done same as array A since the algorithms is same I ignored to rewrite the steps to save your time and space.