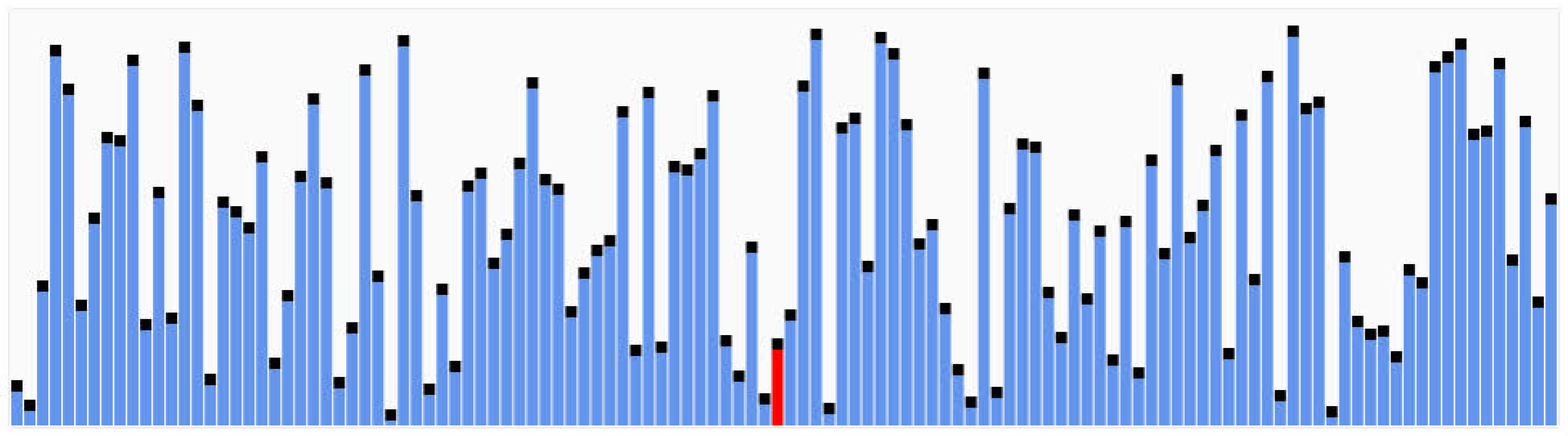


QuickSort Algorithm

Dewi Sintiari
nld.sintiari@gmail.com

March 14th, 2022

Objectives

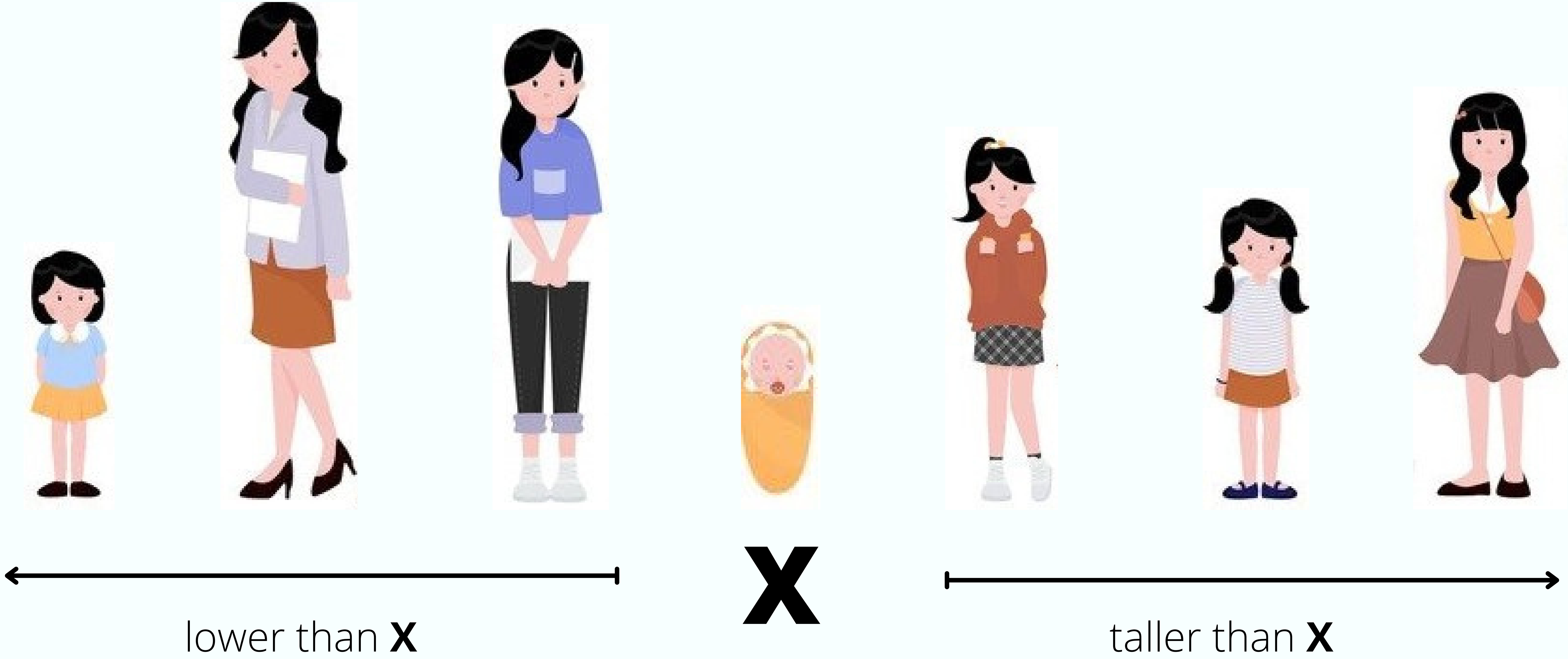
- To understand the principle of Quicksort algorithm
- Able to apply Quicksort algorithm for sorting data
- Able to analyze the time-complexity of Quicksort algorithm

Preliminary

- Quicksort is developed by British computer scientist Tony Hoare in 1959 and published in 1961
- It's still a commonly used algorithm for sorting
- When implemented well, it can be quite fast

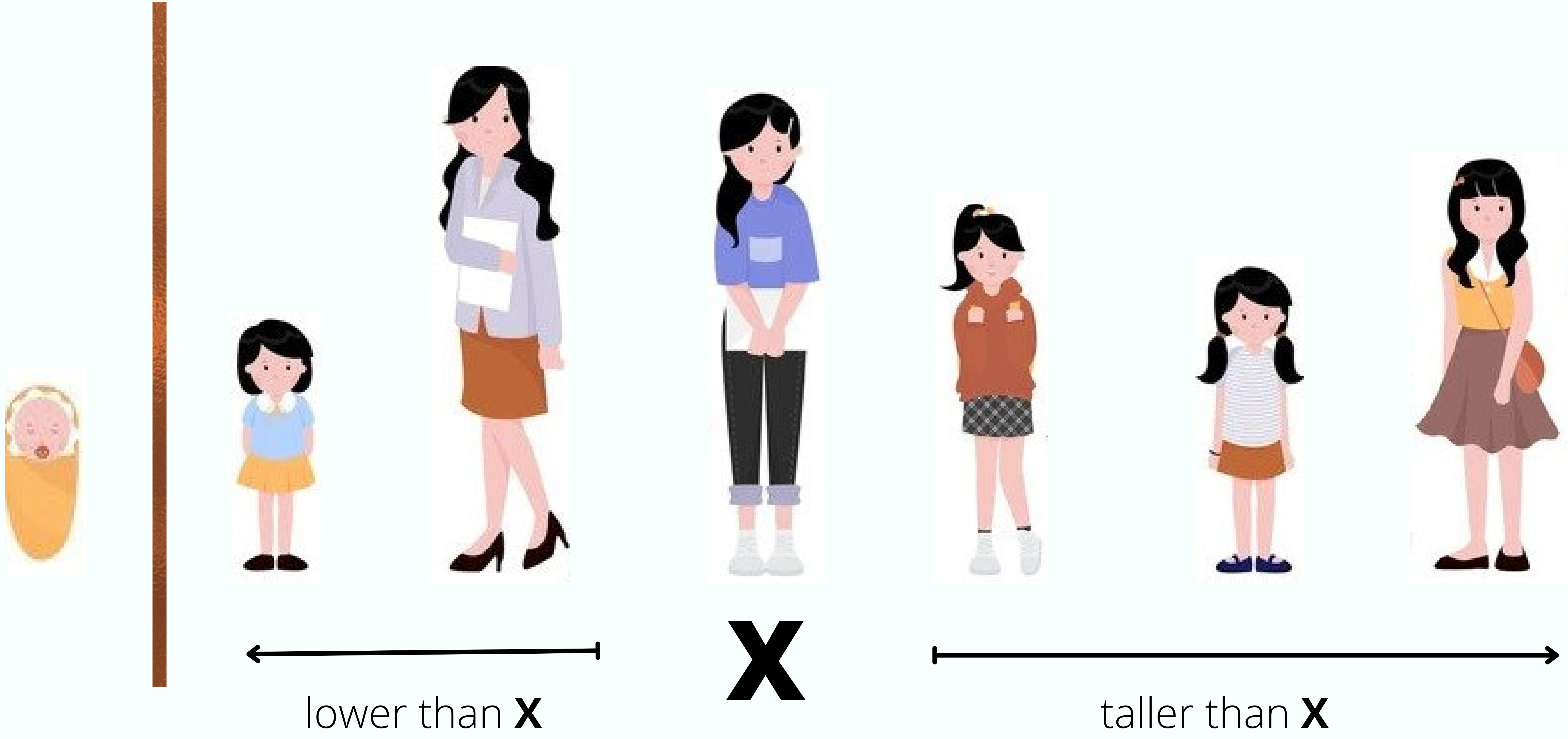
How to sort the girls "quickly" ?

How to sort the girls "quickly" ?



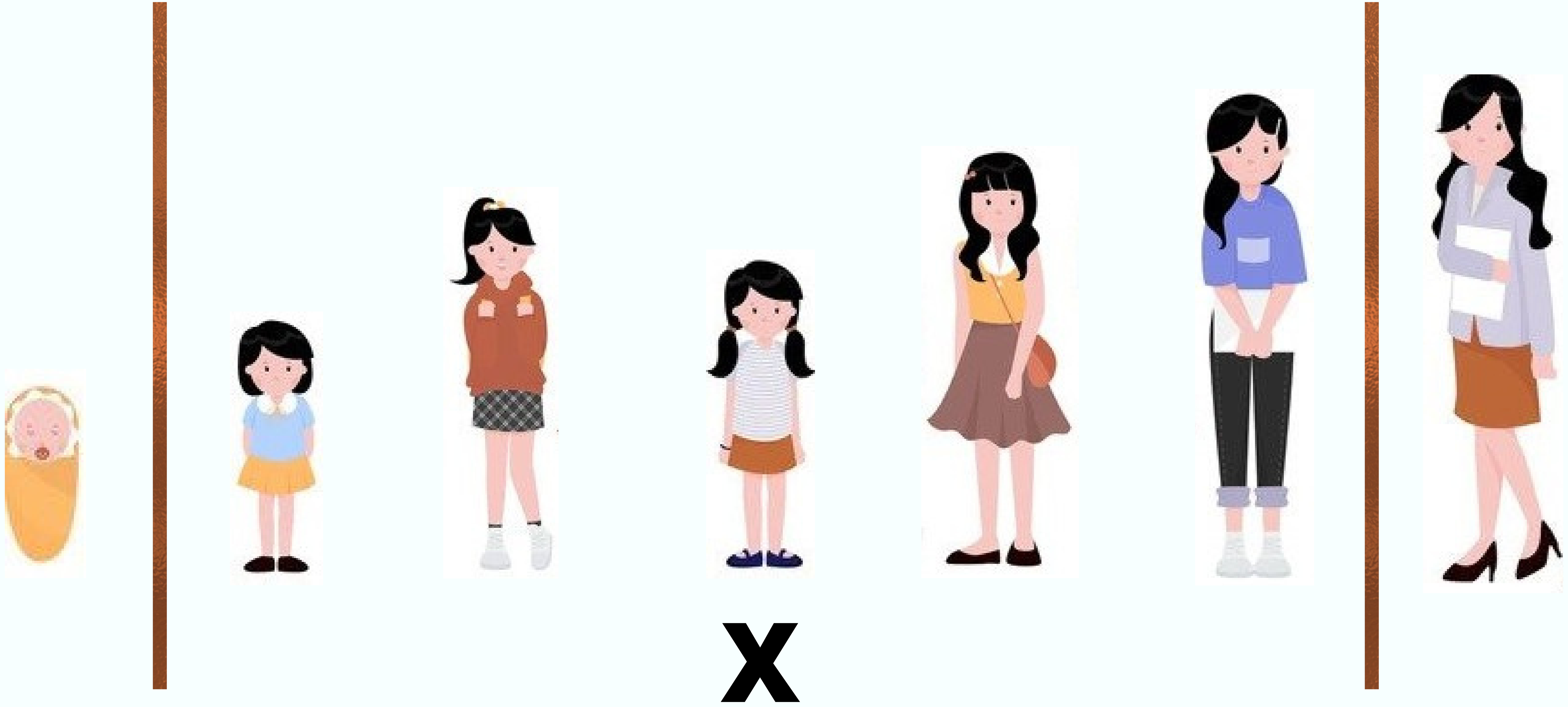
How to sort the girls "quickly" ?

How to sort the girls "quickly" ?

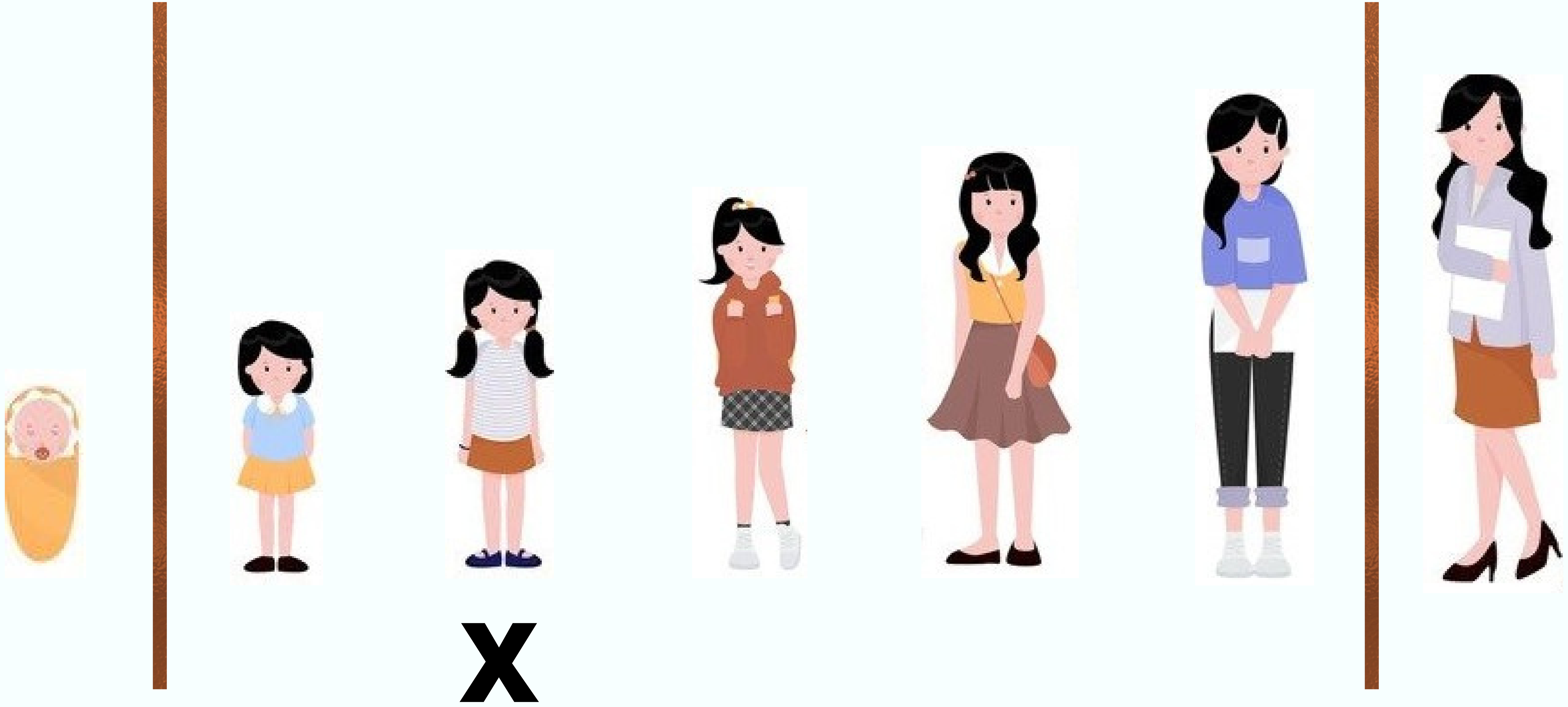


How to sort the girls "quickly"?

How to sort the girls "quickly" ?



How to sort the girls "quickly" ?



How to sort the girls "quickly" ?

The idea of QuickSort

We say that an element **X** is **sorted** if it is in the **correct** position

- All elements that are less than **X** appear before **X**
- All elements that are greater than **X** appear after **X**

The idea of QuickSort

Input: a list **A** of *unsorted* elements
Output: sorted list of **A**

Quicksort is a divide-and-conquer algorithm

- At each step, we split the problem into two subproblems, and solve each subproblem
- For every problem, select a *pivot* **X**
- Move all elements "smaller" than **X** before **X**
- Move all elements "bigger" than **X** after **X**

A =

10	16	8	12	15	6	3	9	5

Example

low									high
0	1	2	3	4	5	6	7	8	9
10	16	8	12	15	6	3	9	5	∞

Example

	0	1	2	3	4	5	6	7	8	9
	low									**high**
pivot = 10	10	16	8	12	15	6	3	9	5	∞

pivot is chosen as the *first element* of the array

QuickSort **Partitioning** Procedure

	low									high
	0	1	2	3	4	5	6	7	8	9
pivot = 10	**10**	16	8	12	15	6	3	9	5	∞
	i →								←	**j**

- **i** is the index that will look for **element > pivot**
- **j** is the index that will look for **element < pivot**
- such two elements will be **exchanged**

QuickSort Partitioning Procedure

	low									high
	0	1	2	3	4	5	6	7	8	9
pivot = 10	10	16	8	12	15	6	3	9	5	∞
		i ⟶							⟵ j	

QuickSort Partitioning Procedure

	low									high
	0	1	2	3	4	5	6	7	8	9
pivot = 10	**10**	**5**	8	12	15	6	3	9	**16**	∞
		i $\longrightarrow$						$\longleftarrow$	j	

QuickSort Partitioning Procedure

	low									high
	0	1	2	3	4	5	6	7	8	9
pivot = 10	10	5	8	12	15	6	3	9	16	∞
		i →						←	j	

QuickSort Partitioning Procedure

	low									high
	0	1	2	3	4	5	6	7	8	9
pivot = 10	10	5	8	12	15	6	3	9	16	∞
				i →				← j		

QuickSort **Partitioning** Procedure

	low									high
	0	1	2	3	4	5	6	7	8	9
pivot = 10	**10**	5	8	**9**	15	6	3	**12**	16	∞
				i ⟶				⟵ j		

QuickSort <u>Partitioning</u> Procedure

	low									high
	0	1	2	3	4	5	6	7	8	9
pivot = 10	**10**	5	8	9	15	6	3	12	16	∞
				i	⟶		⟵	**j**		

QuickSort Partitioning Procedure

	low									high
	0	1	2	3	4	5	6	7	8	9
pivot = 10	**10**	5	8	9	**15**	6	**3**	12	16	∞
					i		**j**			

QuickSort Partitioning Procedure

	low									high
	0	1	2	3	4	5	6	7	8	9
pivot = 10	10	5	8	9	3	6	15	12	16	∞
					i		j			

QuickSort Partitioning Procedure

	low									high
	0	1	2	3	4	5	6	7	8	9
pivot = 10	10	5	8	9	3	6	15	12	16	∞
					i		j			

QuickSort **Partitioning** Procedure

	low									high
	0	1	2	3	4	5	6	7	8	9
pivot = 10	**10**	5	8	9	3	**6**	**15**	12	16	∞
						j	**i**			

- we **STOP** (do not interchange **i** and **j**), now **i** is on the right of **j**

QuickSort Partitioning Procedure

	low									high
	0	1	2	3	4	5	6	7	8	9
pivot = 10	**6**	5	8	9	3	**10**	**15**	12	16	∞
						j	i			

- we **STOP** (do not interchange **i** and **j**), now **i** is on the right of **j**
- interchange **A[j]** and **pivot**

QuickSort **Partitioning** Procedure

	low									high
	0	1	2	3	4	5	6	7	8	9
pivot = 10	6	5	8	9	3	**10**	15	12	16	∞

< 10 **> 10**

- we **STOP** (do not interchange **i** and **j**), now **i** is on the right of **j**
- interchange **A[j]** and **pivot**
- Now **pivot is in the correct position**
  - all elements before pivot are < 10
  - all elements after pivot are > 10

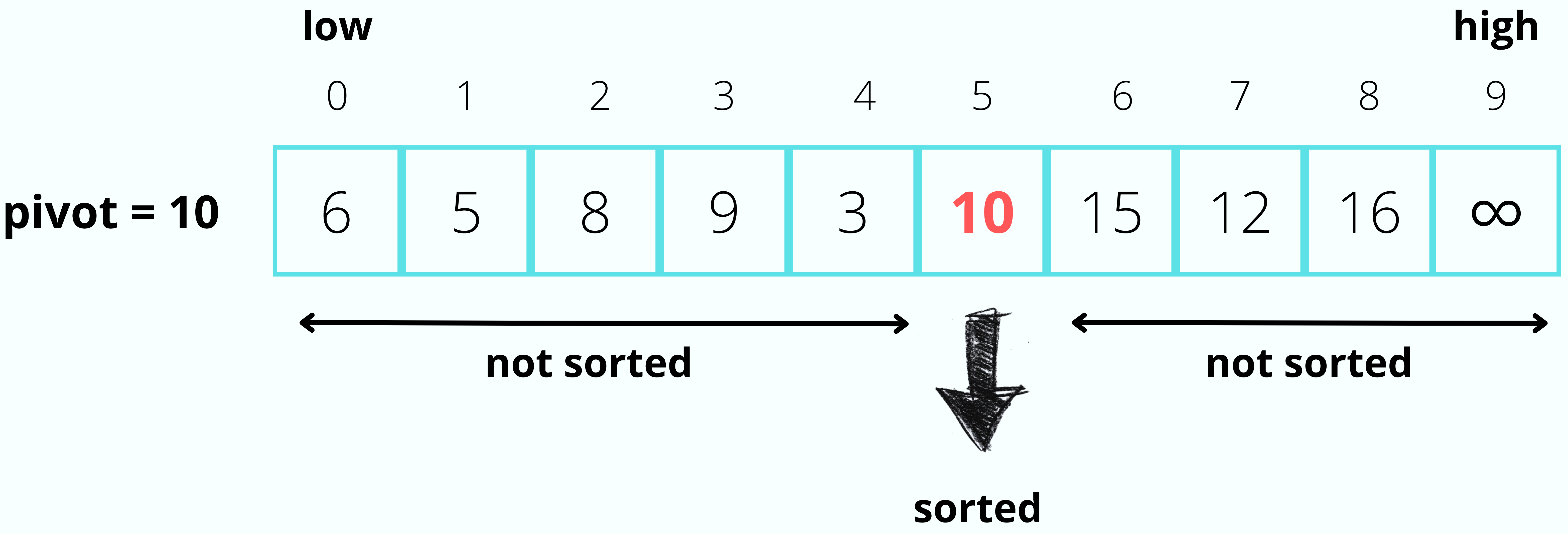
QuickSort Partitioning Procedure

	low									high
	0	1	2	3	4	5	6	7	8	9
pivot = 10	6	5	8	9	3	**10**	15	12	16	∞

not sorted **not sorted**

- we **STOP** (do not interchange **i** and **j**), now **i is on the right of j**
- interchange **A[j]** and **pivot**
- Now **pivot is in the correct position**
  - all elements before pivot are < 10
  - all elements after pivot are > 10

QuickSort Partitioning Procedure



This is called "partitioning position"

QuickSort **Partitioning** Procedure

low high

pivot = 10

0	1	2	3	4	5	6	7	8	9
6	5	8	9	3	10	15	12	16	∞

not sorted

not sorted

sorted

Pseudocode

Finding pivot's position

```
Partition(low,high):
    pivot = A[low]
    i=low
    j=high
    while (i<j)
        while (A[i]<=pivot)
            i+=1
        while (A[j]>pivot)
            j-=1
        if (i<j)
            swap(A[i],A[j])
    swap(A[low],A[j])
    return j
```

QuickSort Algorithm

	low									high
	0	1	2	3	4	5	6	7	8	9
pivot = 10	6	5	8	9	3	10	15	12	16	∞
	l					j				h

```
QuickSort(low,high):
    if (low<high):
        j = Partition(low,high)
        QuickSort(low,j)
        QuickSort(j+1,high)
```

Pseudocode

Finding pivot's position

```
Partition(low,high):
    pivot = A[low]
    i=low
    j=high
    while (i<j)
        while (A[i]<=pivot)
            i+=1
        while (A[j]>pivot)
            j-=1
        if (i<j)
            swap(A[i],A[j])
    swap(A[low],A[j])
    return j
```

QuickSort Algorithm

	low									high
	0	1	2	3	4	5	6	7	8	9
pivot = 10	6	5	8	9	3	10	15	12	16	∞
	l					j				h

```
QuickSort(low,high):
    if (low<high):
        j = Partition(low,high)
        QuickSort(low,j)
        QuickSort(j+1,high)
```

Question:

- why include **j** (it is sorted already) ?
- where is the 'infinity' for the left partition ?

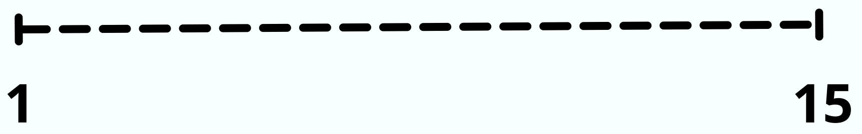
Pseudocode

Finding pivot's position

```
Partition(low,high):
    pivot = A[low]
    i=low
    j=high
    while (i<j)
        while (A[i]<=pivot)
            i+=1
        while (A[j]>pivot)
            j-=1
        if (i<j)
            swap(A[i],A[j])
    swap(A[low],A[j])
    return j
```

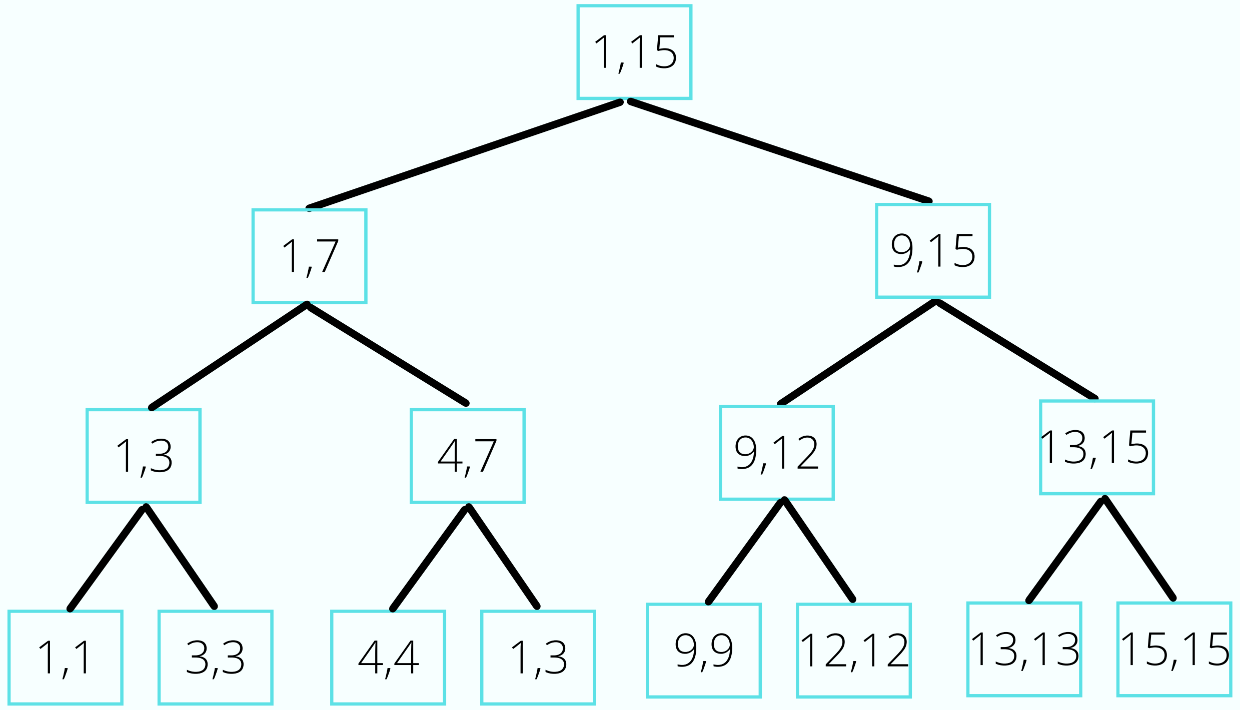
Time-complexity Analysis (best case)

15 elements to sort




```
QuickSort(l,h):
    if (l<h):
        j = Partition(l,h)
        QuickSort(l,j)
        QuickSort(j+1,h)
```

If the pivot is always in the middle



Complexity:

- The divide-and-conquer procedure takes time O(log n)
- The Partition procedure takes time O(n)

Best case time complexity = O(n log n)

Best case is *not* always possible !

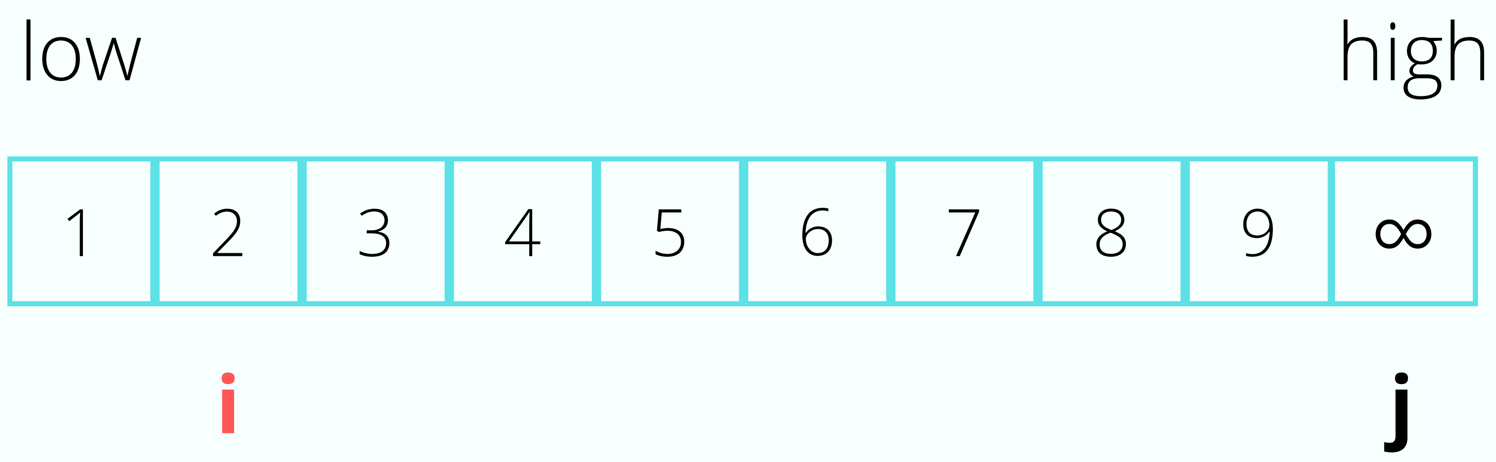
In each step, we must select the **median** as a pivot.
But this is not possible, eventhough it may happen randomly.

Time-complexity Analysis (worst case)

1	2	3	4	5	6	7	8	9	∞
i									j

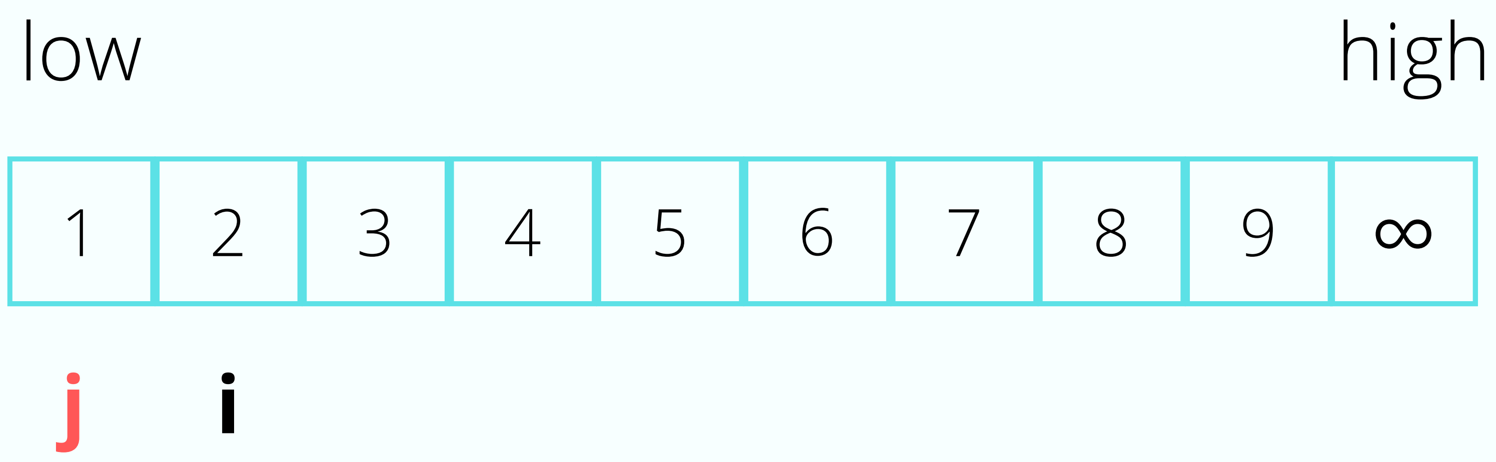
```
Partition(low,high):
    pivot = A[low]
    i=low
    j=high
    while (i<j)
        while (A[i]<=pivot)
            i+=1
        while (A[j]>pivot)
            j-=1
        if (i<j)
            swap(A[i],A[j])
    swap(A[low],A[j])
    return j
```

Time-complexity Analysis (worst case)



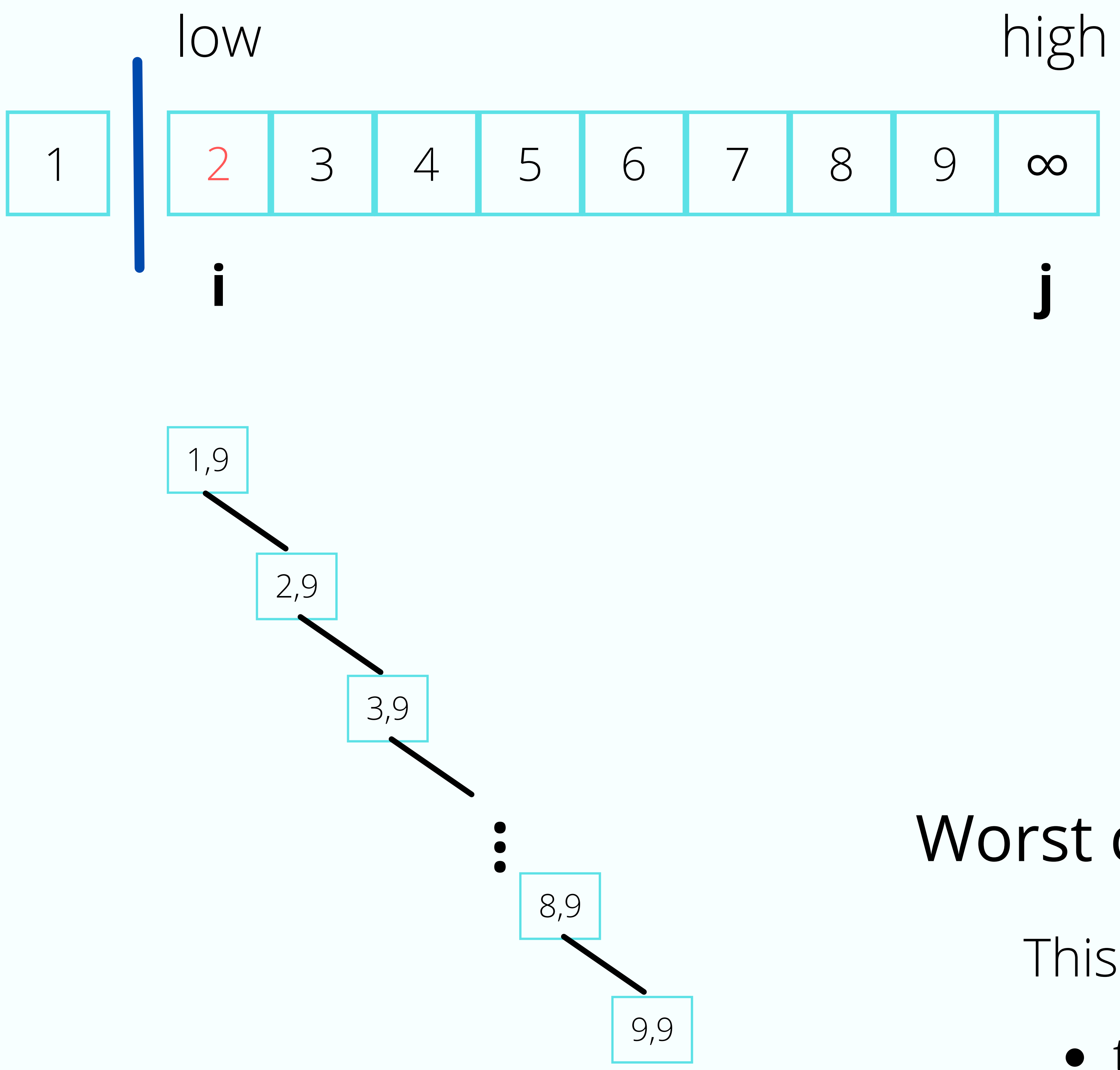

```
Partition(low,high):
   pivot = A[low]
   i=low
   j=high
   while (i<j)
      while (A[i]<=pivot)
         i+=1
      while (A[j]>pivot)
         j-=1
      if (i<j)
         swap(A[i],A[j])
   swap(A[low],A[j])
   return j
```

Time-complexity Analysis (worst case)




```
Partition(low,high):
    pivot = A[low]
    i=low
    j=high
    while (i<j)
        while (A[i]<=pivot)
            i+=1
        while (A[j]>pivot)
            j-=1
        if (i<j)
            swap(A[i],A[j])
    swap(A[low],A[j])
    return j
```

Time-complexity Analysis (worst case)




```
Partition(low,high):
    pivot = A[low]
    i=low
    j=high
    while (i<j)
        while (A[i]<=pivot)
            i+=1
        while (A[j]>pivot)
            j-=1
        if (i<j)
            swap(A[i],A[j])
    swap(A[low],A[j])
    return j
```

Worst case time complexity = $O(n) \times O(n) = O(n^2)$

This could happen when:

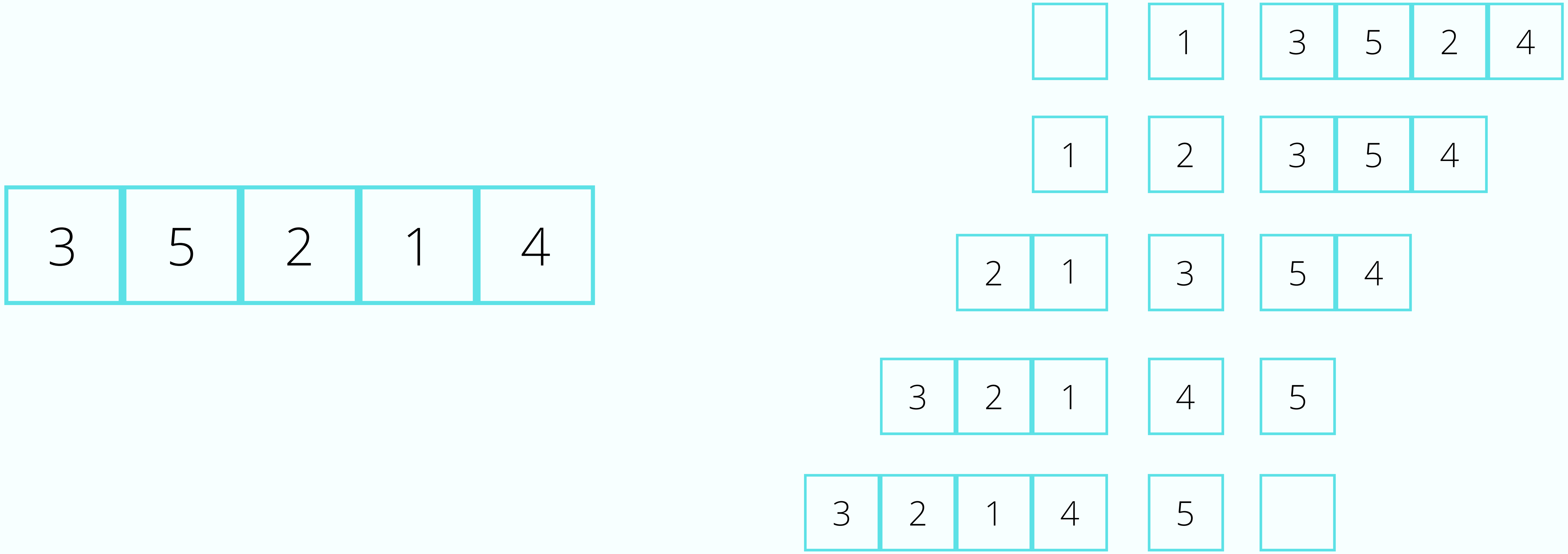
- the list is already **sorted**,
- or it is **sorted in the reverse order**

How to avoid the worst case ?

- so far, we choose the first element of the list
- this increases the chance of getting the worst-case complexity

Alternatives

- choose the pivot **randomly**
- choose the **middle-most** element of the list as the pivot



What we learned today

- The principle of Quicksort algorithm
- Best-case complexity = O(n log n)
- Worst-case complexity = $O(n^2)$
- A way of minimizing the probability of getting worst-case complexity is by changing the method of choosing the pivot

  Some ways of choosing pivot:
  - the first/last element
  - the middle-most element
  - randomly

Quiz

Suppose we are sorting an array of eight integers using quicksort, and we have just finished the first partitioning with the array looking like this:

2	5	1	7	9	12	11	10

Which statement is correct? Explain your argument!

A. The pivot could be either 7 or 9
B. The pivot could be 7, but it is not 9
C. The pivot is not 7, but it could be 9
D. Neither 7 nor 9 is the pivot

Quiz

Suppose we are sorting an array of eight integers using quicksort, and we have just finished the first partitioning with the array looking like this:

2	5	1	7	9	12	11	10

Which statement is correct? Explain your argument!

A. The pivot could be either 7 or 9
B. The pivot could be 7, but it is not 9
C. The pivot is not 7, but it could be 9
D. Neither 7 nor 9 is the pivot

Answer: **A**

Explanation

7 and 9 both are at their correct positions (as in a sorted array). Also, all elements on the left of 7 and 9 are smaller than 7 and 9 respectively and on right are greater than 7 and 9 respectively.