Searching & Sorting

Searching data involves determining whether a value (referred to as the search key) is present in the data and, if so, finding the value's location.

Two popular search algorithms are

- Linear search
- Binary search

Sorting places data in ascending or descending order, based on one or more sort keys.

Popular Sorting Algorithms

- Selection sort
- Insertion sort
- Bubble sort
- Heap sort
- Merge sort
- Quick sort

Big O: Constant Runtime

- Suppose an algorithm simply tests whether the first element of an array is equal to the second element.
- If the array has 10 elements, this algorithm requires only one comparison.
- If the array has 1000 elements, the algorithm still requires only one comparison.
- In fact, the algorithm is independent of the number of array elements.
- This algorithm is said to have a constant runtime, which is represented in Big O notation as O(1).

- An algorithm that is O(1) does not necessarily require only one comparison.
- O(1) just means that the number of comparisons is constant—it does not grow as the size of the array increases.
- An algorithm that tests whether the first element of an array is equal to any of the next three elements will always require three comparisons, but in Big O notation it's still considered O(1).
- O(1) is often pronounced "on the order of 1" or more simply "order 1."

Big O: Linear Runtime

- An algorithm that tests whether the first element of an array is equal to any of the other elements of the array requires at most n – 1 comparisons, where n is the number of elements in the array.
- If the array has 10 elements, the algorithm requires up to nine comparisons.
- If the array has 1000 elements, the algorithm requires up to 999 comparisons.
- As n grows larger, the n part of the expression n – 1 “dominates,” and subtracting one becomes inconsequential.
- Big O is designed to highlight these dominant terms and ignore terms that become unimportant as n grows.

- An algorithm that requires a total of n – 1 comparisons is said to be O(n) and is referred to as having a linear runtime.
- O(n) is often pronounced “on the order of n” or more simply “order n.”

Big O: Quadratic Runtime

- Now suppose you have an algorithm that tests whether any element of an array is duplicated elsewhere in the array.
- The first element must be compared with every other element in the array.
- The second element must be compared with every other element except the first (it was already compared to the first).
- The third element then must be compared with every other element except the first two.
- In the end, this algorithm will end up making $(n - 1) + (n - 2) + \ldots + 2 + 1$ or $n^2/2 - n/2$ comparisons.
- As n increases, the n^2 term dominates and the n term becomes inconsequential.
- Again, Big O notation highlights the n^2 term, leaving $n^2/2$.

- Big O is concerned with how an algorithm's runtime grows in relation to the number of items processed.
- Suppose an algorithm requires n^2 comparisons.
- With four elements, the algorithm will require 16 comparisons; with eight elements, 64 comparisons.
- With this algorithm, doubling the number of elements quadruples the number of comparisons.
- Consider a similar algorithm requiring $n^2/2$ comparisons.
- With four elements, the algorithm will require eight comparisons; with eight elements, 32 comparisons.
- Again, doubling the number of elements quadruples the number of comparisons.
- Both of these algorithms grow as the square of n, so Big O ignores the constant, and both algorithms are considered to be $O(n^2)$, which is referred to as quadratic runtime and pronounced "on the order of n-squared" or more simply "order n-squared."

$O(n^2)$ Performance

- When n is small, $O(n^2)$ algorithms will not noticeably affect performance.
- As n grows, you'll start to notice the performance degradation.
- An $O(n^2)$ algorithm running on a million-element array would require a trillion "operations" (where each could actually require several machine instructions to execute).
  - This could require hours to execute.
- A billion-element array would require a quintillion operations, a number so large that the algorithm could take decades! Unfortunately, $O(n^2)$ algorithms tend to be easy to write.

- Searching algorithms all accomplish the same goal—finding an element that matches a given search key, if such an element does, in fact, exist.
- The major difference is the amount of effort they require to complete the search.
- One way to describe this effort is with Big O notation.
- For searching and sorting algorithms, this is particularly dependent on the number of data elements.

Sequential Searching

- Can search from unordered list
- Time complexity O(n)

Binary searching

- Can search from ordered list only
- Time complexity $O(\log_2 n)$

Linear search

- Given a list, find a specific element in the list
  - List does NOT have to be sorted!

procedure linear_search (x: integer; a_1, a_2, …, a_n: integers)
i := 1
while ($i \leq n$ and $x \neq a_i$)
  i := i + 1
if $i \leq n$ **then** *location* := i
else *location* := 0

{*location* is the subscript of the term that equals x, or it is 0 if x is not found}

Linear search, take 1

procedure linear_search (x: integer; $a_1, a_2, \ldots, a_n$: integers)

$i := 1$

while ($i \leq n$ and $x \neq a_i$)

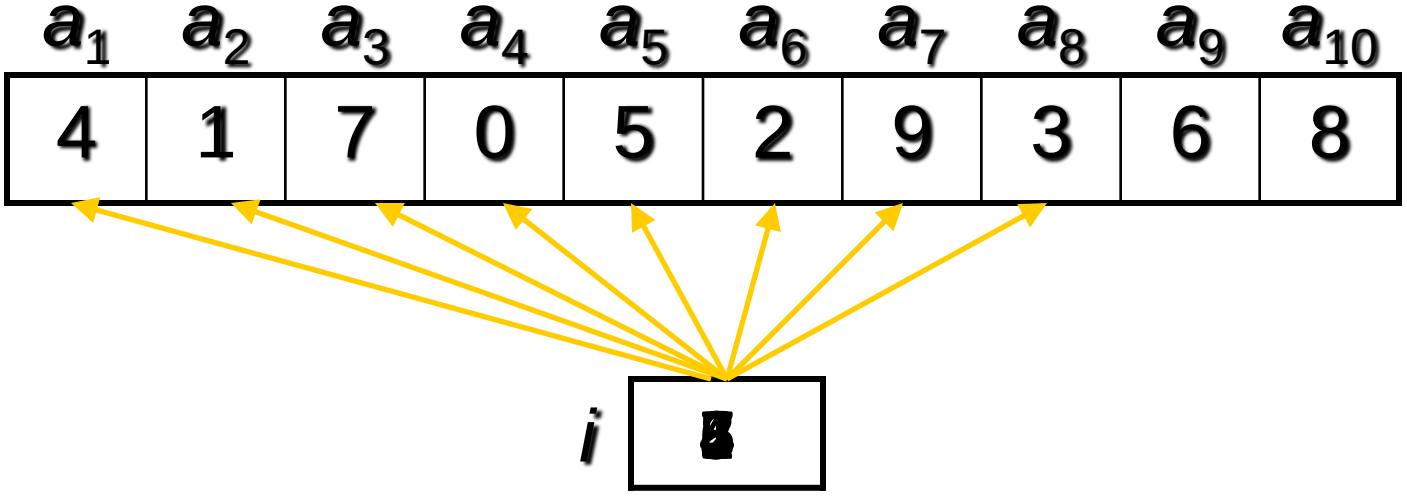
    $i := i + 1$

if $i \leq n$ then *location* := i

else *location* := 0

x	3

location	8



Linear search, take 2

procedure linear_search (*x*: integer; a_1, a_2, …, a_n: integers)

i := 1

while ($i \leq n$ and $x \neq a_i$)

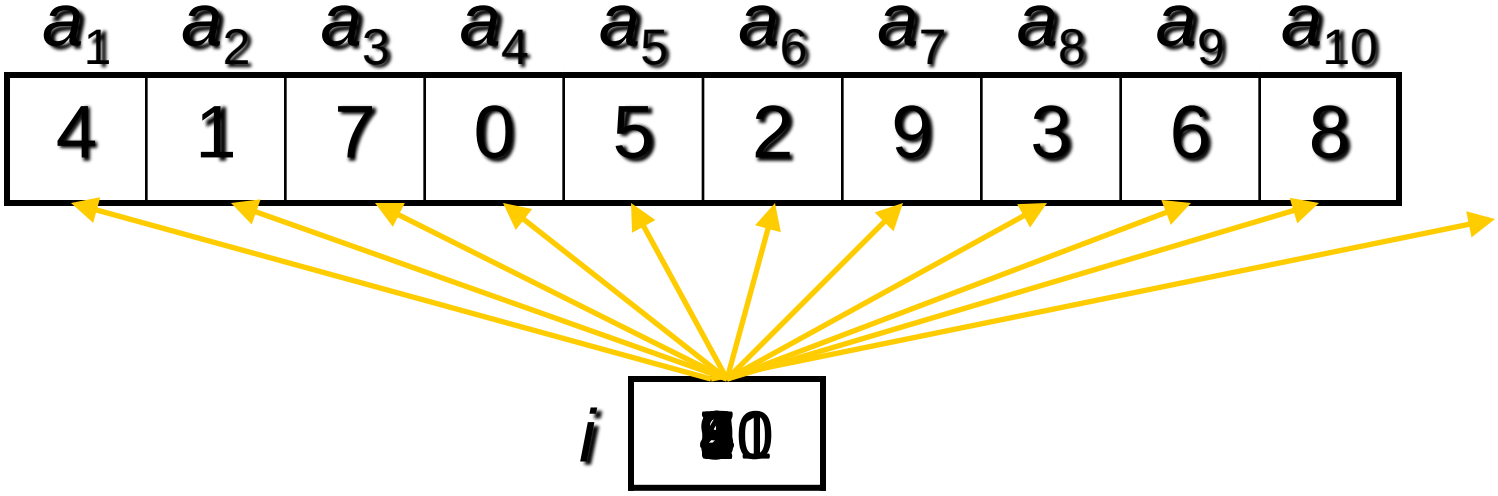
    i := *i* + 1

if $i \leq n$ then *location* := i

else *location* := 0

x	11

location	0



Linear search running time

- How long does this take?

- If the list has *n* elements, worst case scenario is that it takes *n* "steps"
  - Here, a step is considered a single step through the list

linear_search(arr, n, val)

Step-1 : [Initialize] Set pos=-1

Step-2: [Initialize] Set i=1

Step-3: Repeat Step 4 while i<=n

If(arr[i]=val)

Set pos=i

Print pos

Goto Step 6

[End of If]

Set i=i+1

[End of loop]

Step 5: if pos=-1

Print “Value is not present”

[End of if]

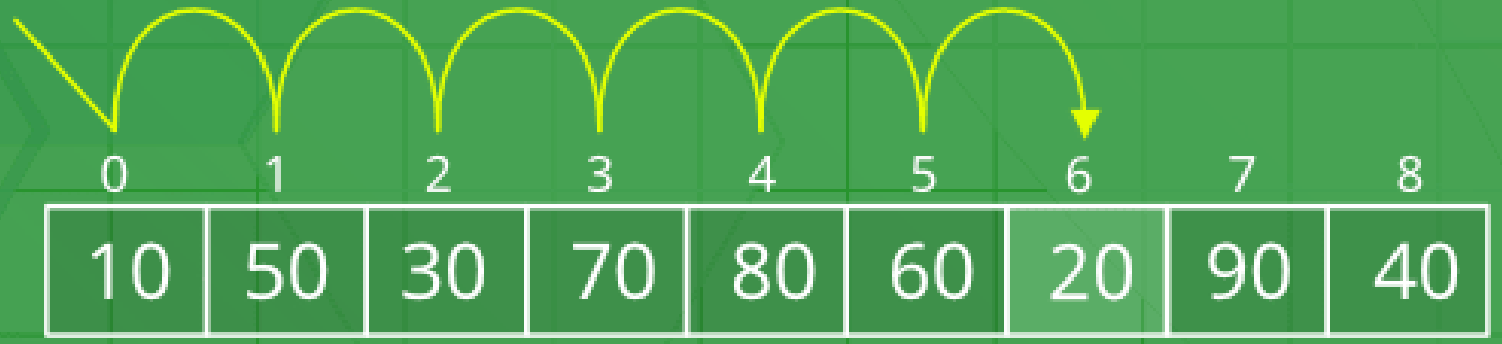
Step-6: Exit

A simple approach is to do **linear search**, i.e

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
- If x matches with an element, return the index.
- If x doesn’t match with any of elements, return -1.

Linear Search

Find '20'



Worst Case Analysis (Usually Done)

Worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be O(n).

Average Case Analysis (Sometimes done)

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by (n+1). Following is the value of average case time complexity.

Average Case Analysis (Sometimes done)

$$\frac{\sum_{i=1}^{n+1} O(i)}{(n+1)}$$

$$\frac{O((n+1)*(n+2)/2)}{(n+1)}$$

$$= O(n)$$

Linear Search's Runtime

- The linear search algorithm runs in O(n) time.
- The worst case in this algorithm is that every element must be checked to determine whether the search key is in the array.
- If the array's size doubles, the number of comparisons that the algorithm must perform also doubles.
- Linear search can provide outstanding performance if the element matching the search key happens to be at or near the front of the array.
- But we seek algorithms that perform well, on average, across all searches, including those where the element matching the search key is near the end of the array.
- If a program needs to perform many searches on large arrays, it may be better to implement a different, more efficient algorithm, such as the binary search which we present in the next section.

Binary search

- Given a list, find a specific element in the list
  - List MUST be sorted!
- Each time it iterates through, it cuts the list in half

procedure binary_search (x: integer; a_1, a_2, …, a_n: increasing integers)
i := 1{ i is left endpoint of search interval }
j := n { j is right endpoint of search interval }
while $i < j$
begin
    m := $\lfloor (i+j)/2 \rfloor$ { m is the point in the middle }
    if x > a_m **then** i := m+1
    else j := m
end
if $x = a_i$ **then** *location* := i
else *location* := 0

{*location* is the subscript of the term that equals x, or it is 0 if x is not found}

Binary search, take 1

procedure binary_search (x: integer; $a_1, a_2, \ldots, a_n$: increasing integers)

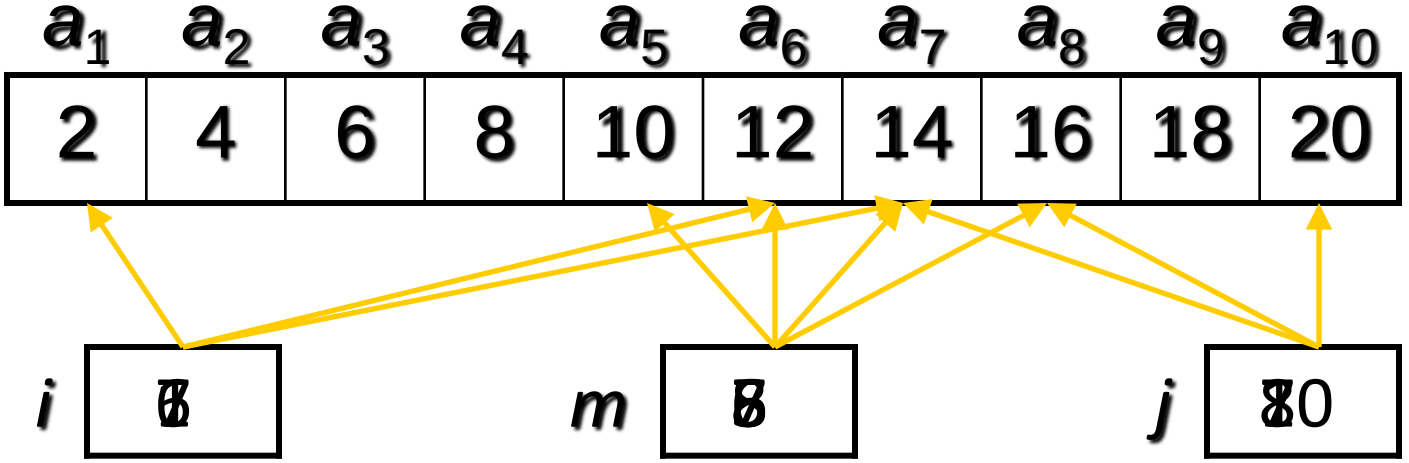
$i := 1$

$j := n$

while $i < j$
begin
  $m := \lfloor (i+j)/2 \rfloor$
  if $x > a_m$ then $i := m+1$
  else $j := m$
end

if $x = a_i$ then *location* $:= i$
else *location* := 0

x	14
location	7

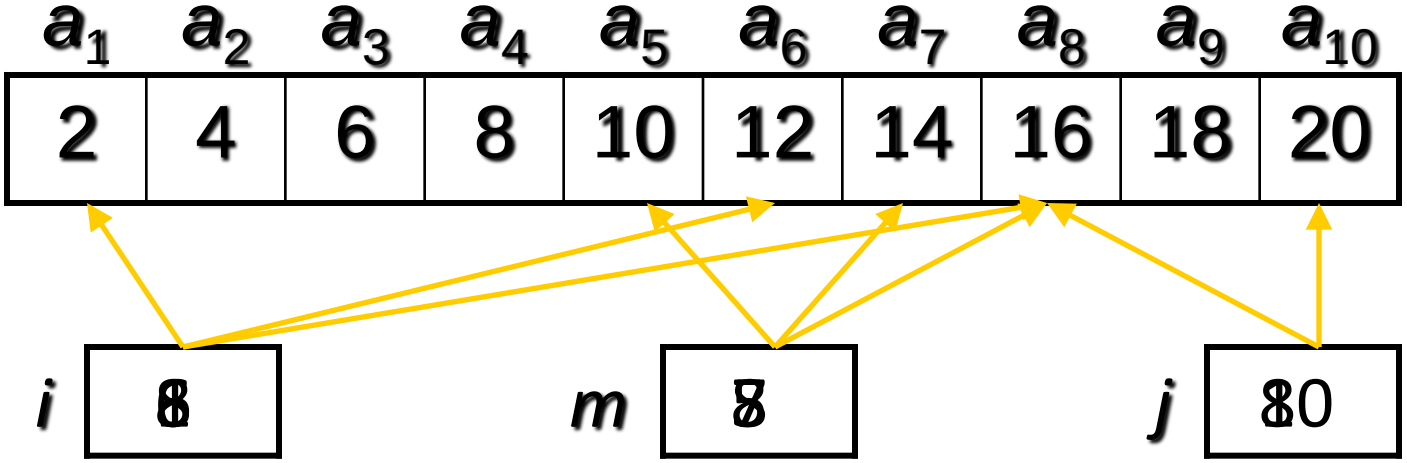


Binary search, take 2

procedure binary_search (x: integer; a_1, a_2, …, a_n: increasing integers)

```
i := 1
j := n
while i < j
begin
    m := ⌊(i+j)/2⌋
    if x > a_m then i := m+1
    else j := m
end
if x = a_i then location := i
else location := 0
```

x	15
location	0



Binary search

- A somewhat alternative view of what a binary search does…

Binary search running time

- How long does this take (worst case)?

- If the list has 8 elements
  - It takes 3 steps
- If the list has 16 elements
  - It takes 4 steps
- If the list has 64 elements
  - It takes 6 steps

- If the list has *n* elements
  - It takes $\log_2 n$ steps

Binary_search(arr, lb, ub, val)

```
Step-1 : [Initialize]  Set beg=lb
end = ub;
pos=-1
Step-2 : Repeat step 3 and 4 while beg<=end
Step-3 :    set mid=(beg+end)/2
Step-4:                  if arr[mid]=val
                                    Set pos=mid
                                    Print pos
                                    Go to step 6
                         Else
                                    If(arr[mid]> val)
                                              Set end = mid -1
                                    Else
                                              Set beg=mid+1
                                    [End of if]
                         [End of loop]
Step – 5: if pos=-1
                         Print " Value is not present in the array"
              [End of If]
Step-6 : Exit
```

Binary Search

Sorted Array of 10 elements: 2, 5, 8, 12, 16, 23, 38, 56, 72, 91

Let us say we want to search for 23.

Finding the given element:

Now to find 23, there will be many iterations with each having steps as mentioned in the figure above:

- Iteration 1:Array: 2, 5, 8, 12, 16, 23, 38, 56, 72, 91
    - Select the middle element. (here 16)
    - Since 23 is greater than 16, so we divide the array into two halves and consider the sub-array after element 16.
    - Now this subarray with the elements after 16 will be taken into next iteration.

- Iteration 2:Array: 23, 38, 56, 72, 91
  - Select the middle element. (now 56)
  - Since 23 is smaller than 56, so we divide the array into two halves and consider the sub-array before element 56.
  - Now this subarray with the elements before 56 will be taken into next iteration.
- Iteration 3:Array: 23, 38
  - Select the middle element. (now 23)
  - Since 23 is the middle element. So the iterations will now stop.

Calculating Time complexity:

Let say the iteration in Binary Search terminates after k iterations. In the above example, it terminates after 3 iterations, so here k = 3

At each iteration, the array is divided by half. So let's say the length of array at any iteration is n

- At Iteration 1,Length of array = n
- At Iteration 2,Length of array = $n/2$
- At Iteration 3,Length of array = $(n/2)/2 = n/2^2$
- Therefore, after Iteration k,Length of array = $n/2^k$
- Also, we know that afterAfter k divisions, the length of array becomes 1
- Therefore Length of array = $n/2^k = 1 \Rightarrow n = 2^k$
- Applying log function on both sides:$\Rightarrow \log_2(n) = \log_2(2^k) \Rightarrow \log2(n) = k \log_2(2)$
- As ($\log_a(a) = 1$)

Therefore,$\Rightarrow k = \log_2(n)$

Hence, the time complexity of Binary Search is

$\log_2 (n)$

Efficiency of Binary Search

- In the worst-case scenario, searching a sorted array of 1023 elements will take only 10 comparisons when using a binary search.
- Repeatedly dividing 1023 by 2 (because, after each comparison, we can eliminate from consideration half of the remaining elements) and rounding down (because we also remove the middle element) yields the values 511, 255, 127, 63, 31, 15, 7, 3, 1 and 0.
- The number 1023 (210 – 1) is divided by 2 only 10 times to get the value 0, which indicates that there are no more elements to test.

- Dividing by 2 is equivalent to one comparison in the binary search algorithm.
- Thus, an array of 1,048,575 (220 – 1) elements takes a maximum of 20 comparisons to find the key, and an array of approximately one billion elements takes a maximum of 30 comparisons to find the key.
- This is a tremendous performance improvement over the linear search.
- For a one-billion-element array, this is a difference between an average of 500 million comparisons for the linear search and a maximum of only 30 comparisons for the binary search!
- The maximum number of comparisons needed for the binary search of any sorted array is the exponent of the first power of 2 greater than the number of elements in the array, which is represented as log2 n.

- All logarithms grow at roughly the same rate, so in Big O notation the base can be omitted.
- This results in a Big O of $O(\log_2 n)$ for a binary search, which is also known as logarithmic runtime and pronounced “on the order of log n” or more simply “order log n.”

Binary Search Performance

- Successful Search
  - – Best Case – 1 comparison
  - – Worst Case – $\log_2 N$ comparisons
- Unsuccessful Search
  - – Best Case =

  Worst Case – $\log_2 N$ comparisons
- Since the portion of an array to search is cut into half after every comparison, we compute how many times the array can be divided into halves.
- After K comparisons, there will be $N/2^K$ elements in the list.

  We solve for K when $N/2^K = 1$, deriving $K = \log_2 N$

Comparing N and $\log_2 N$ Performance

Array Size	Linear – N	Binary – $\log_2 N$
10	10	4
50	50	6
100	100	7
500	500	9
1000	1000	10
2000	2000	11
3000	3000	12
4000	4000	12
5000	5000	13
6000	6000	13
7000	7000	13
8000	8000	13
9000	9000	14
10000	10000	14

Bubble Sort

Bubble sort is an algorithm that compares the adjacent elements and swaps their positions if they are not in the intended order. The order can be ascending or descending.

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

Bubble Sort Example

9, 6, 2, 12, 11, 9, 3, 7

Bubblesort compares the numbers in pairs from left to right exchanging when necessary. Here the first number is compared to the second and as it is larger they are exchanged.

Now the next pair of numbers are compared. Again the 9 is the larger and so this pair is also exchanged.

In the third comparison, the 9 is not larger than the 12 so no exchange is made. We move on to compare the next pair without any change to the list.

The 12 is larger than the 11 so they are exchanged.

The twelve is greater than the 9 so they are exchanged

The end of the list has been reached so this is the end of the first pass. The twelve at the end of the list must be largest number in the list and so is now in the correct position. We now start a new pass from left to right.

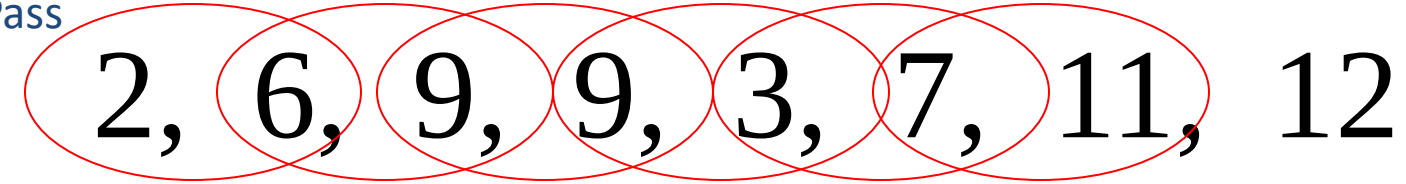
The 12 is greater than the 7 so they are exchanged.

Bubble Sort Example

First Pass

6, 2, 9, 11, 9, 3, 7, 12

Second Pass



Notice that this time we do not have to compare the last two numbers as we know the 12 is in position. This pass therefore only requires 6 comparisons.

Bubble Sort Example

First Pass

6, 2, 9, 11, 9, 3, 7, 12

Second Pass

2, 6, 9, 9, 3, 7, 11, 12

Third Pass

This time the 11 and 12 are in position. This pass therefore only requires 5 comparisons.

Bubble Sort Example

First Pass

6, 2, 9, 11, 9, 3, 7, 12

Second Pass

2, 6, 9, 9, 3, 7, 11, 12

Third Pass

2, 6, 9, 3, 7, 9, 11, 12

Fourth Pass

2, 6, 3, 7, 9, 9, 11, 12

Each pass requires fewer comparisons. This time only 4 are needed.

Bubble Sort Example

First Pass 6, 2, 9, 11, 9, 3, 7, 12

Second Pass 2, 6, 9, 9, 3, 7, 11, 12

Third Pass 2, 6, 9, 3, 7, 9, 11, 12

Fourth Pass 2, 6, 3, 7, 9, 9, 11, 12

Fifth Pass 2, 3, 6, 7, 9, 9, 11, 12

The list is now sorted but the algorithm does not know this until it completes a pass with no exchanges.

Bubble Sort Example

First Pass

6, 2, 9, 11, 9, 3, 7, 12

Second Pass

2, 6, 9, 9, 3, 7, 11, 12

Third Pass

2, 6, 9, 3, 7, 9, 11, 12

Fourth Pass

2, 6, 3, 7, 9, 9, 11, 12

Fifth Pass

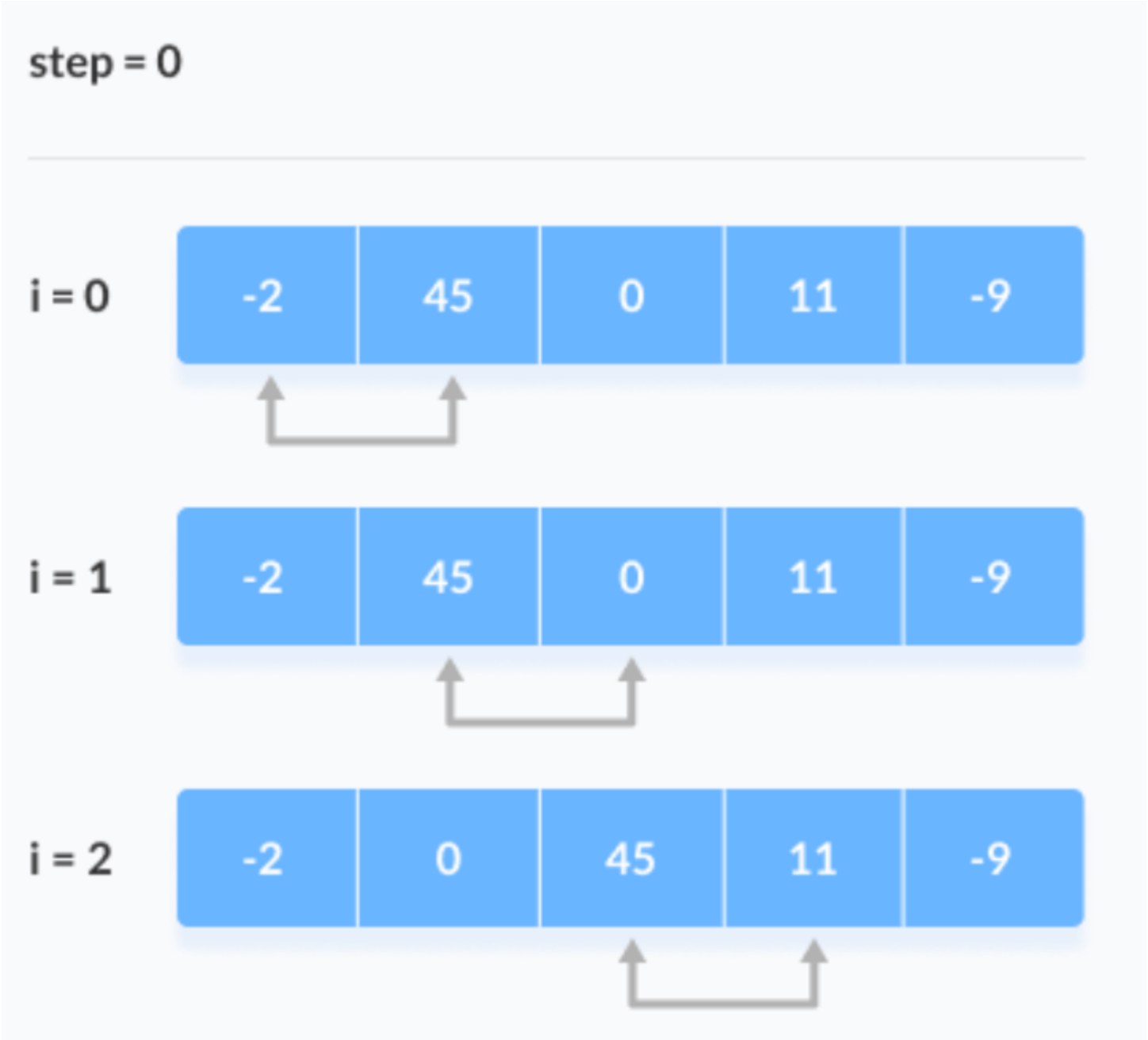
This pass no exchanges are made so the algorithm knows the list is sorted. It can therefore save time by not doing the final pass. With other lists this check could save much more work.

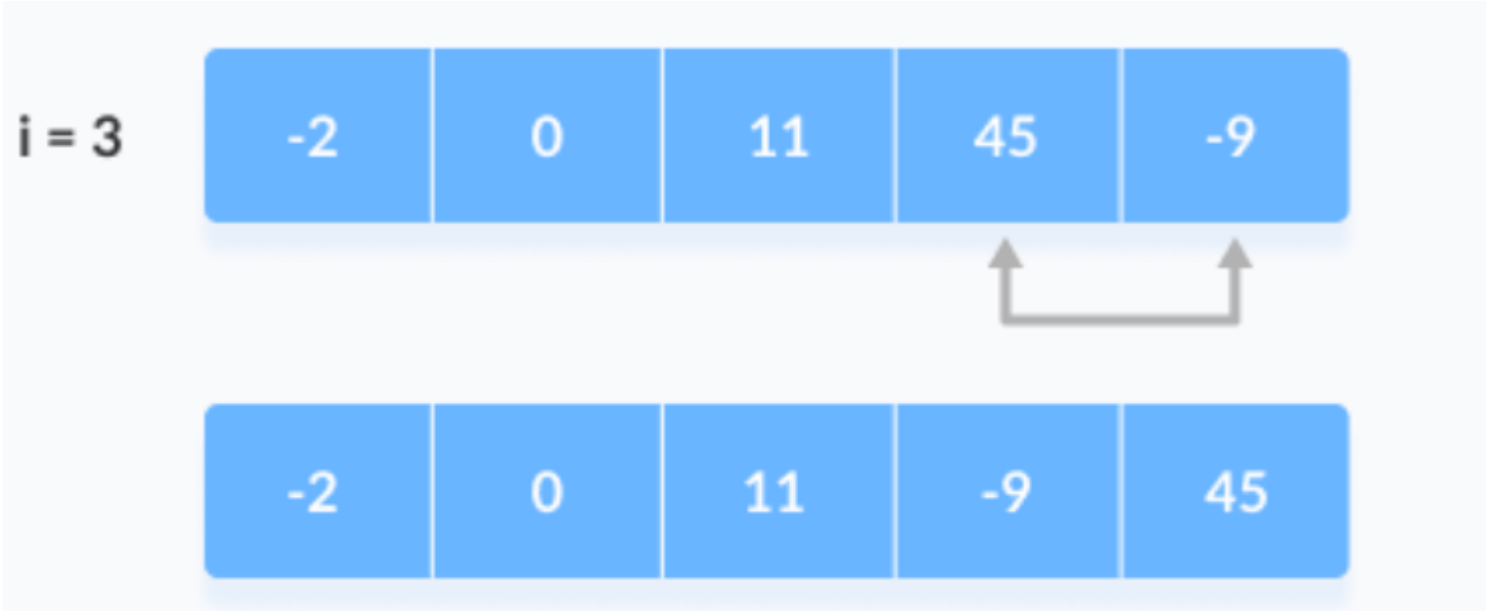
Sixth Pass

2, 3, 6, 7, 9, 9, 11, 12

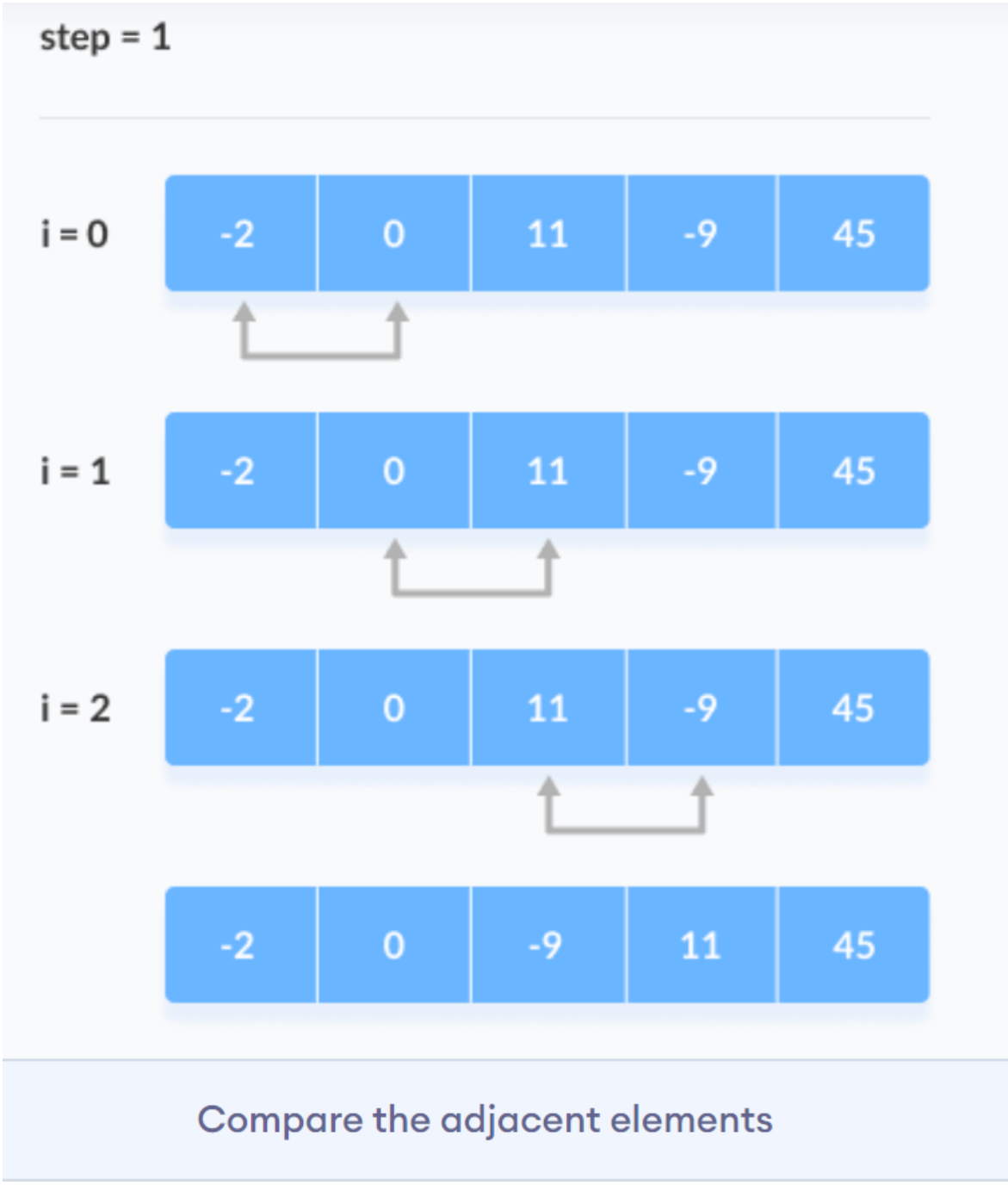
How Bubble Sort Works?

- Starting from the first index, compare the first and the second elements.If the first element is greater than the second element, they are swapped.
- Now, compare the second and the third elements. Swap them if they are not in order.
- The above process goes on until the last element.


step = 0
i = 0
-2
45
0
11
-9
i = 1
-2
45
0
11
-9
i = 2
-2
0
45
11
-9



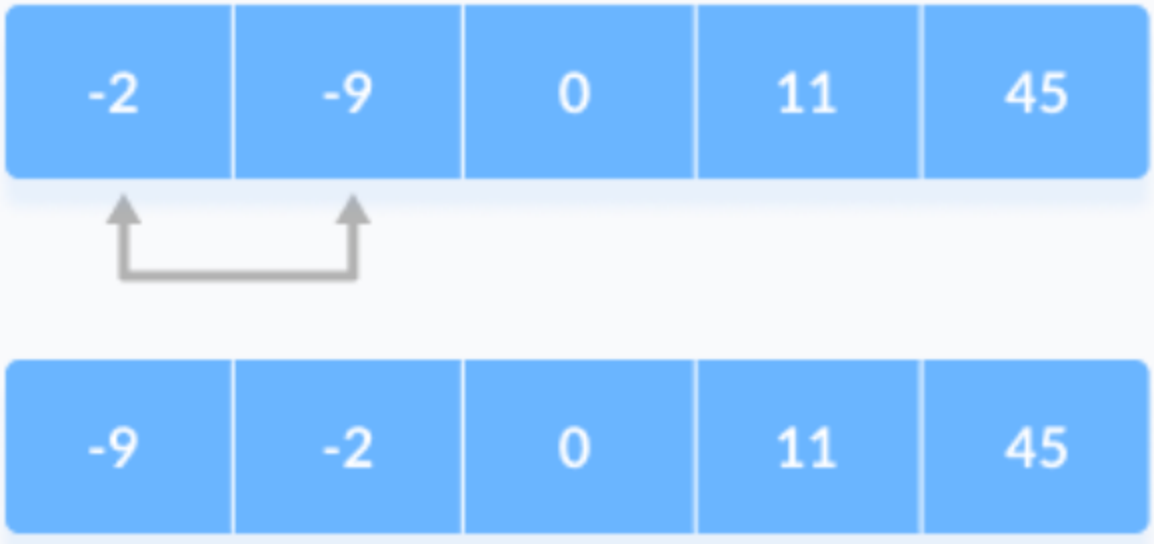
Compare the adjacent elements


step = 1
i = 0
-2
0
11
-9
45
i = 1
-2
0
11
-9
45
i = 2
-2
0
11
-9
45
-2
0
-9
11
45
Compare the adjacent elements

Compare the adjacent elements

step = 3

i = 0



Compare the adjacent elements

Bubble Sort Algorithm

```
bubbleSort(array)
for i <- 1 to indexOfLastUnsortedElement-1
        if leftElement > rightElement
                swap leftElement and rightElement
end bubbleSort
```

Optimized Bubble Sort

- In the above code, all possible comparisons are made even if the array is already sorted. It increases the execution time.
- The code can be optimized by introducing an extra variable swapped. After each iteration, if there is no swapping taking place then, there is no need for performing further loops.
- In such a case, variable swapped is set false. Thus, we can prevent further iterations.

Algorithm for optimized bubble sort is

2. The same process goes on for the remaining iterations.

After each iteration, the largest element among the unsorted elements is placed at the end.

In each iteration, the comparison takes place up to the last unsorted element.

The array is sorted when all the unsorted elements are placed at their correct positions.

```
bubbleSort(array)
swapped <- false
for i <- 1 to indexOfLastUnsortedElement-1
        if leftElement > rightElement
                swap leftElement and rightElement
        swapped <- true
end bubbleSort
```

Bubble_sort(arr, n)

Step 1: Repeat Step 2 for i=0 to n-1

Step 2: Repeat for j= 0 to (n-1)-1

Step 3: if arr[j]>arr[j+1

Swap arr[j] and arr[j+1]

[End of The Inner Loop]

[End of The outer loop]

Step 4: Exit

Analysis

Here, the number of comparisons are

$1 + 2 + 3 + \ldots + (n - 1) = n(n - 1)/2 = O(n^2)$

Clearly, the graph shows the n^2 nature of the bubble sort.

In this algorithm, the number of comparison is irrespective of the data set, i.e. whether the provided input elements are in sorted order or in reverse order or at random.

Complexity

Bubble Sort is one of the simplest sorting algorithms. Two loops are implemented in the algorithm.

Cycle	Number of Comparisons
1st	(n-1)
2nd	(n-2)
3rd	(n-3)
.....	.....
Last	1

Number of comparisons: (n - 1) + (n - 2) + (n - 3) +.....+ 1 = n(n - 1) / 2 nearly equals to n^2

Complexity: $O(n^2)$

Also, we can analyze the complexity by simply observing the number of loops. There are 2 loops so the complexity is $n*n = n^2$

Insertion Sort

Insertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position i.e, the position to which it belongs in a sorted array.

It iterates the input elements by growing the sorted array at each iteration. It compares the current element with the largest value in the sorted array. If the current element is greater, then it leaves the element in its place and moves on to the next element else it finds its correct position in the sorted array and moves it to that position. This is done by shifting all the elements, which are larger than the current element, in the sorted array to one position ahead

Insertion sort works similarly as we sort cards in our hand in a card game.

We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put at their right place.

A similar approach is used by insertion sort.

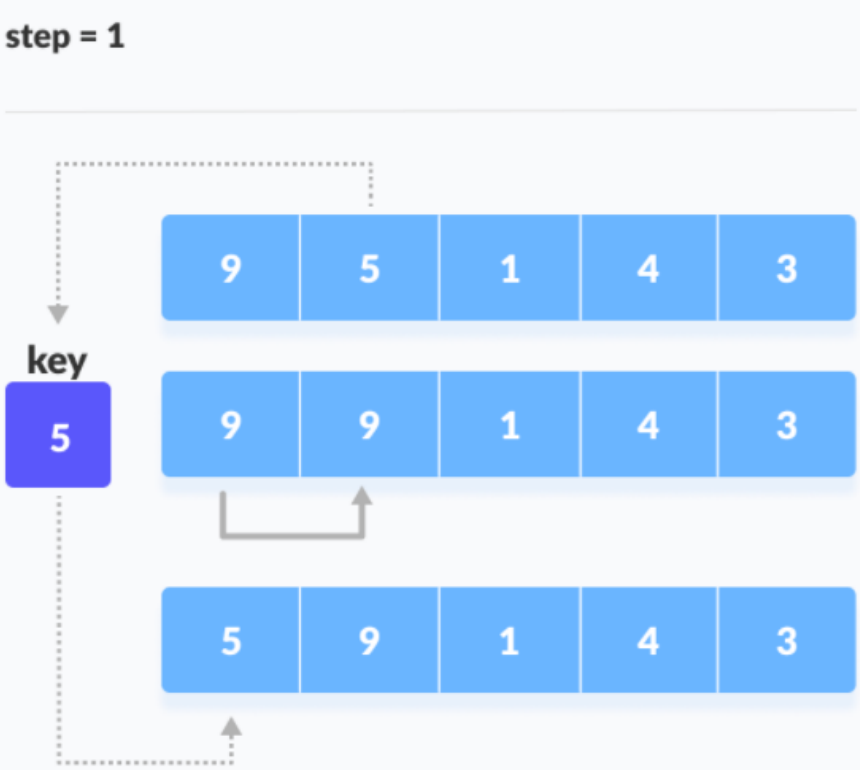
Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

How Insertion Sort Works?

Suppose we need to sort the following array.

1. The first element in the array is assumed to be sorted. Take the second element and store it separately in key.

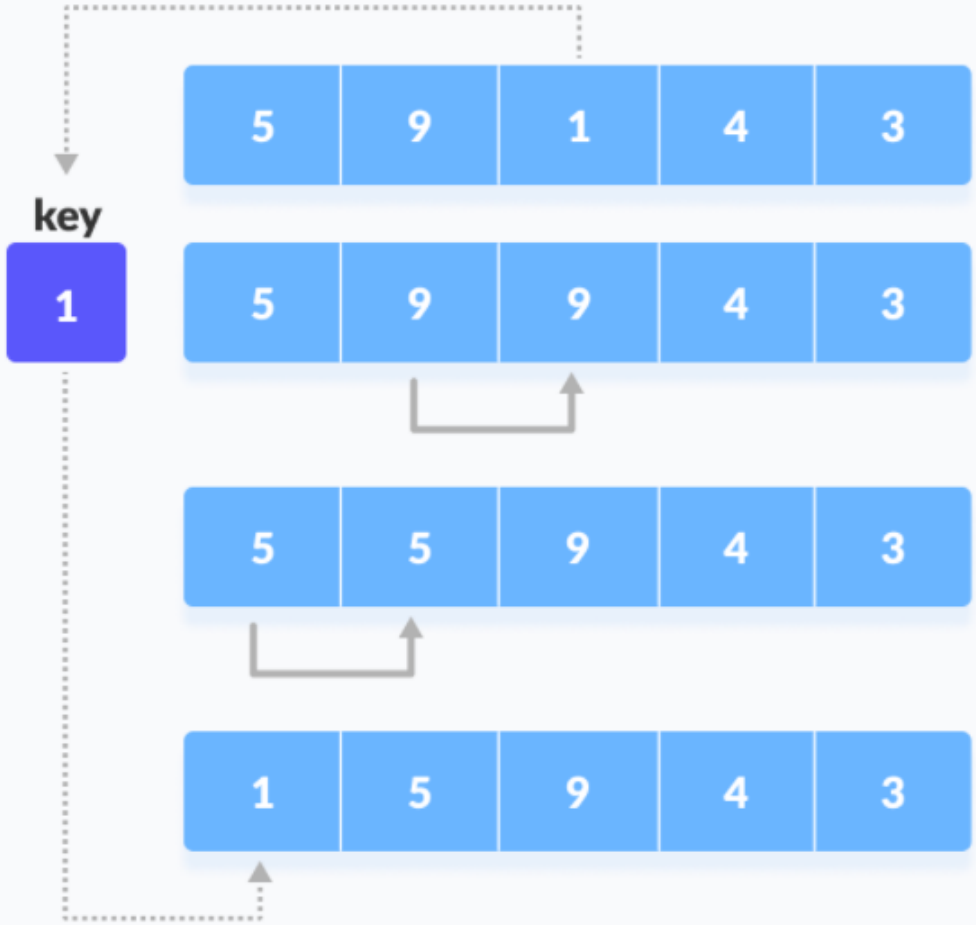
Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.



If the first element is greater than key, then key is placed in front of the first element.

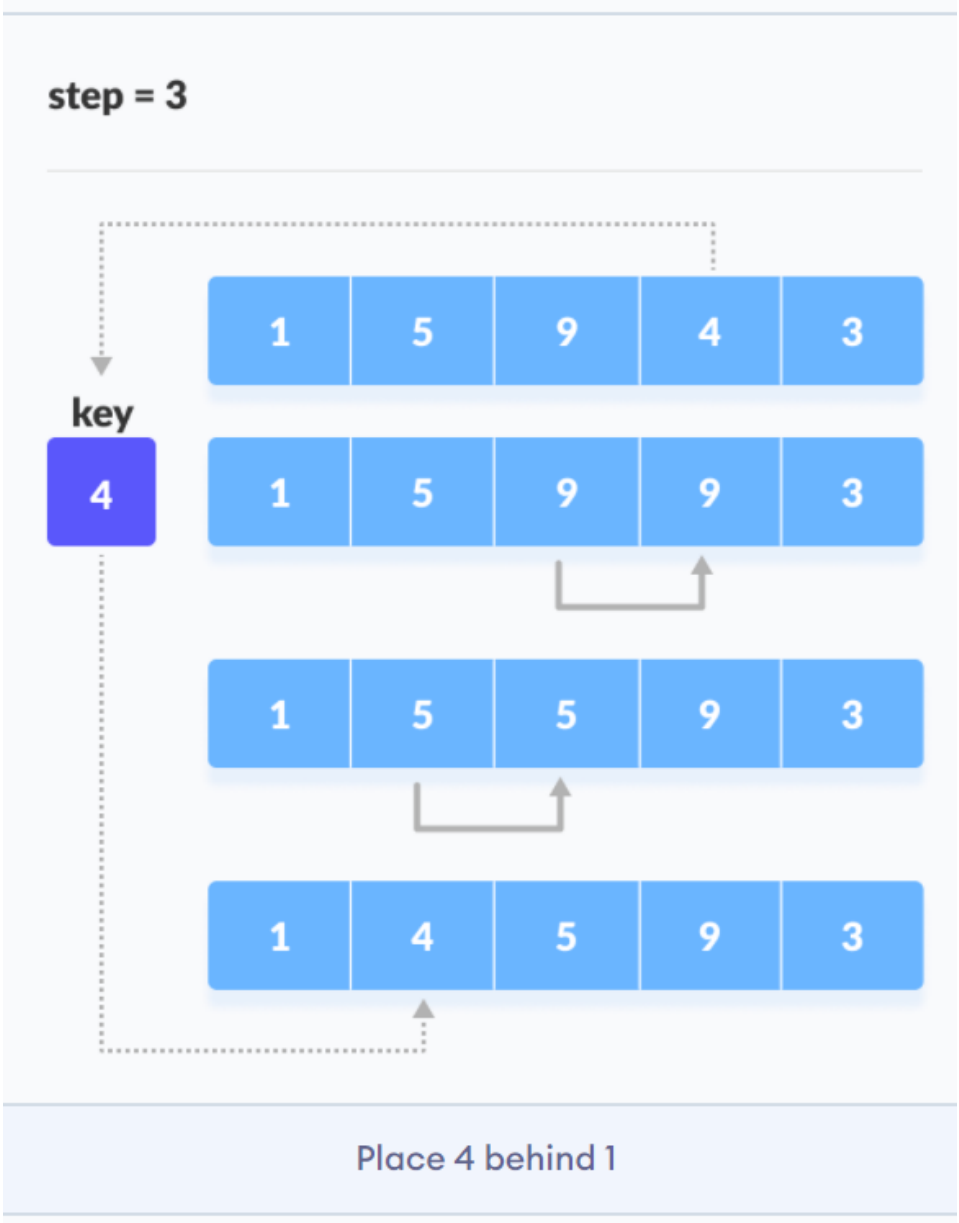
Now, the first two elements are sorted. Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.

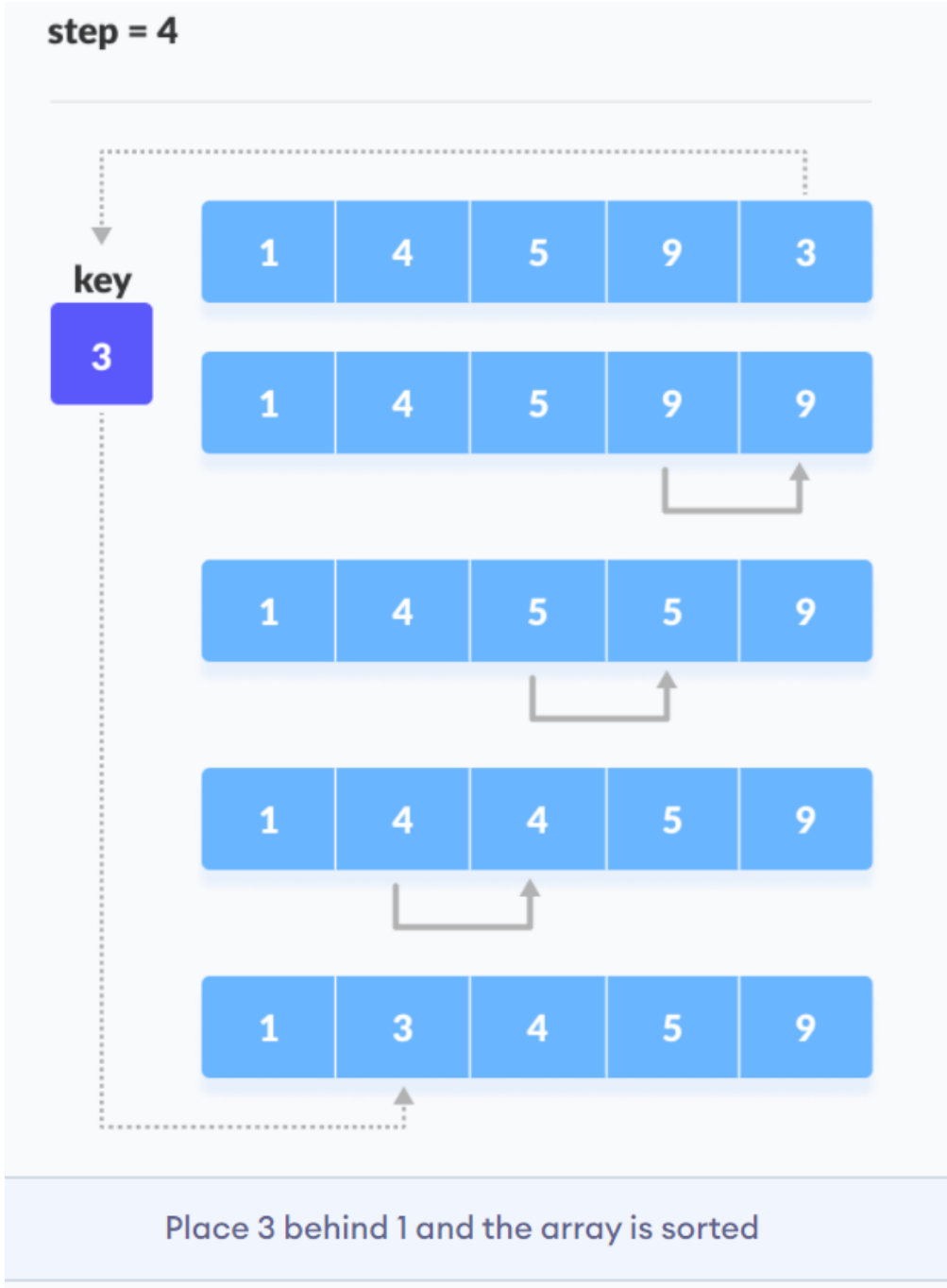
step = 2



Place 1 at the beginning

Similarly, place every unsorted element at its correct position.




step = 4
key
3
1 4 5 9 3
1 4 5 9 9
1 4 5 5 9
1 4 4 5 9
1 3 4 5 9
Place 3 behind 1 and the array is sorted

Insertion Sort Algorithm

```
insertionSort(array)
 mark first element as sorted
  for each unsorted element X
   'extract' the element X
   for j <- lastSortedIndex down to 0
  if current element j > X
  move sorted element to the right by 1
 break loop and insert X here
end insertionSort
```

Insertion_sort(arr, n)
Step -1: Repeat steps 2 to 5 for k=1 and n-1
Step-2: Set temp = arr[k]
Step -3: set j=k-1
Step 4 : Repeat while temp <=arr[j]
Set arr[[j+1] = arr[j]
Set j=j-1
[End of the inner loop]
Step 5: Set arr[j+1=Temp
[End of the loop]
Step-6 : Exit

Time Complexity of Insertion Sort

Worst Case Complexity: $O(n^2)$

Suppose, an array is in ascending order, and you want to sort it in descending order. In this case, worst case complexity occurs.

Each element has to be compared with each of the other elements so, for every nth element, (n-1) number of comparisons are made.

Thus, the total number of comparisons = $n*(n-1) \sim n^2$

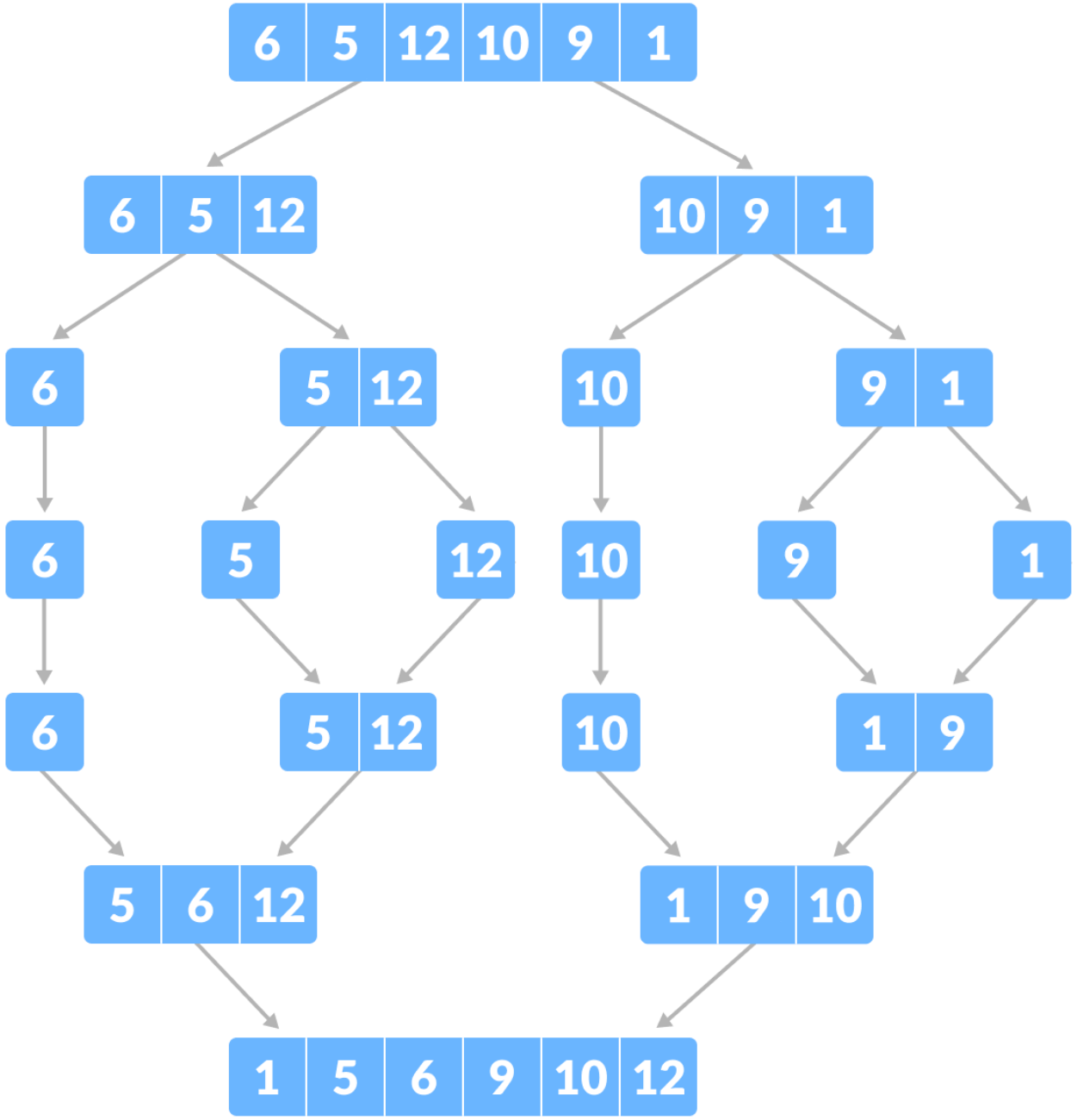
Best Case Complexity: $O(n)$

When the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run at all. So, there are only n number of comparisons. Thus, complexity is linear.

Merge Sort

Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several sublists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

Merge Sort is a kind of Divide and Conquer algorithm in computer programming. It is one of the most popular sorting algorithms and a great way to develop confidence in building recursive algorithms.


6 5 12 10 9 1
6 5 12
10 9 1
6
5 12
10
9 1
6
5
12
10
9
1
6
5 12
10
1 9
5 6 12
1 9 10
1 5 6 9 10 12

Divide and Conquer Strategy

Using the Divide and Conquer technique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem.

Suppose we had to sort an array A. A subproblem would be to sort a sub-section of this array starting at index p and ending at index r, denoted as A[p..r].

Divide

If q is the half-way point between p and r, then we can split the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

Conquer

In the conquer step, we try to sort both the subarrays A[p..q] and A[q+1, r]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

Combine

When the conquer step reaches the base step and we get two sorted subarrays A[p..q] and A[q+1, r] for array A[p..r], we combine the results by creating a sorted array A[p..r] from two sorted subarrays A[p..q] and A[q+1, r].

The MergeSort Algorithm

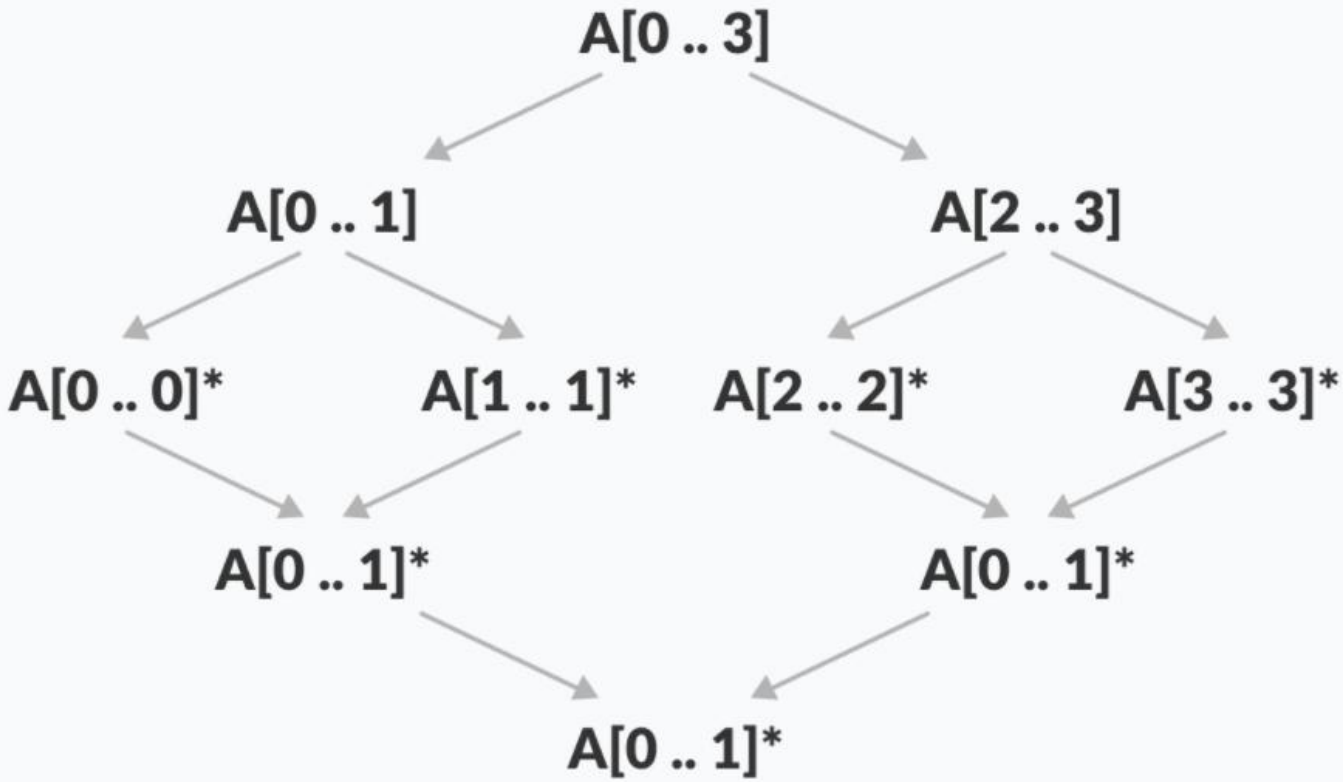
The MergeSort function repeatedly divides the array into two halves until we reach a stage where we try to perform MergeSort on a subarray of size 1 i.e. p == r.

After that, the merge function comes into play and combines the sorted arrays into larger arrays until the whole array is merged.

```
MergeSort(A, p, r):
if p > r
        return q = (p+r)/2
mergeSort(A, p, q)
mergeSort(A, q+1, r)
merge(A, p, q, r)
```

To sort an entire array, we need to call MergeSort(A, 0, length(A)-1).

As shown in the image below, the merge sort algorithm recursively divides the array into halves until we reach the base case of array with 1 element. After that, the merge function picks up the sorted sub-arrays and merges them to gradually sort the entire array.



Merge sort in action

Mearge(arr, beg, mid, end)

Step-1: [Initialize] Set I=beg, j=mid+1, index=0

Step-2: Repeat while (I<=mid) and (j<=end)

If(arr[i]<arr[j])

Set temp[index]=arr[i]

Set I=I+1

Else

Set Temp[Index]=arr[j]

Set j=j+1

[End Of If]

Set Index = Index + 1

[End of loop]

```
Step – 3: [Copy the remaining elements of right sub-array, if any]
                    If I>mid
                            Repeat while J<=end
                                    Set temp[index]=arr[j]
                                    Set index=index +1
                                    Set j=j+1
                            [End of Loop]
                    [Copy the remaining elements of left sub-array, if any]
                    Else
                            Repeat while I<=mid
                                    Set temp[index]=arr[i];
                                    Set index=index+1
                                    Set I=I+1
                            [End of loop]
                    [End of If]
Step-4 :[Copy the contents of temp back to arr] set k=0
Step 5: Repeat while k<Index
                    Set arr[k]=temp[k]
                    Set k=k+1
            [End of loop]
Step-6: End
```

Merge_sort(arr, beg, end)

Step -1 : If Beg<End

Set Mid=(Beg+end)/2

Call Merge_Sort(arr, Beg, Mid)

Call merge_sort(arr, mid+1, end)

Merge(arr, beg,mid, end)

[End of If]

Step-2: End

The merge Step of Merge Sort

Every recursive algorithm is dependent on a base case and the ability to combine the results from base cases. Merge sort is no different. The most important part of the merge sort algorithm is, you guessed it, merge step.

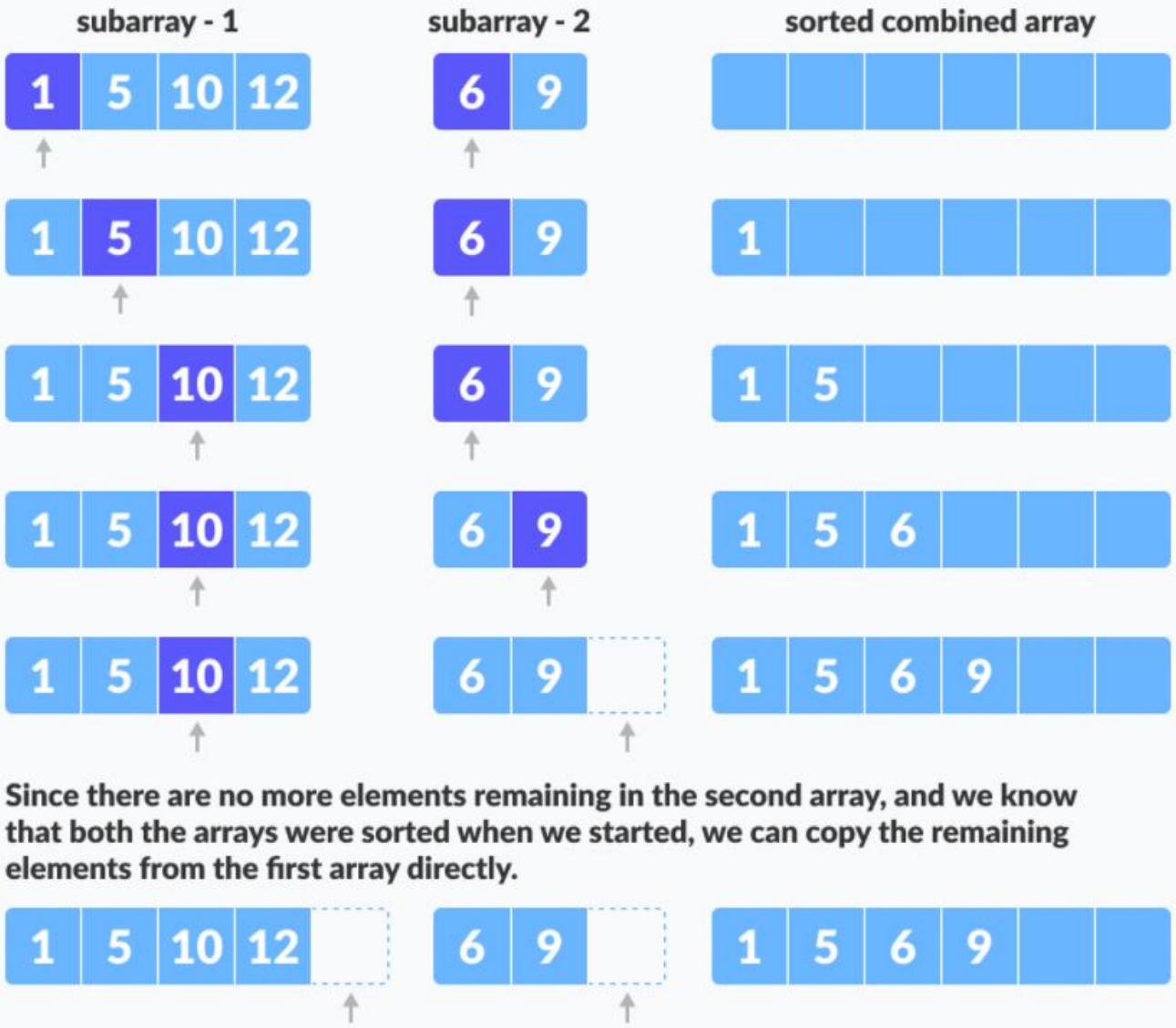
The merge step is the solution to the simple problem of merging two sorted lists(arrays) to build one large sorted list(array).

The algorithm maintains three pointers, one for each of the two arrays and one for maintaining the current index of the final sorted array.

Have we reached the end of any of the arrays?

No: Compare current elements of both arrays Copy smaller element into sorted array Move pointer of element containing smaller element

Yes: Copy all remaining elements of non-empty array



Since there are no more elements remaining in the second array, and we know that both the arrays were sorted when we started, we can copy the remaining elements from the first array directly.

Merge step

Writing the Code for Merge Algorithm

A noticeable difference between the merging step we described above and the one we use for merge sort is that we only perform the merge function on consecutive sub-arrays.

This is why we only need the array, the first position, the last index of the first subarray(we can calculate the first index of the second subarray) and the last index of the second subarray.

Our task is to merge two subarrays A[p..q] and A[q+1..r] to create a sorted array A[p..r]. So the inputs to the function are A, p, q and r

The merge function works as follows:

Create copies of the subarrays L ← A[p..q] and M ← A[q+1..r].

Create three pointers i, j and k

i maintains current index of L, starting at 1

j maintains current index of M, starting at 1

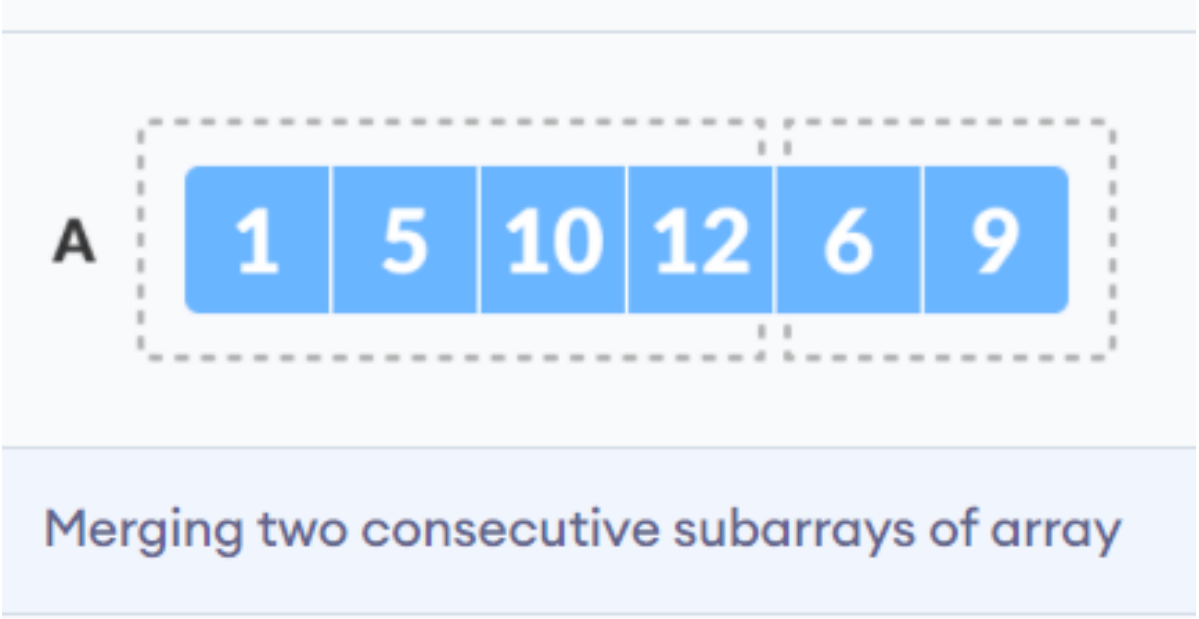
k maintains the current index of A[p..q], starting at p.

Until we reach the end of either L or M, pick the larger among the elements from L and M and place them in the correct position at A[p..q]

When we run out of elements in either L or M, pick up the remaining elements and put in A[p..q]

Merge( ) Function Explained Step-By-Step

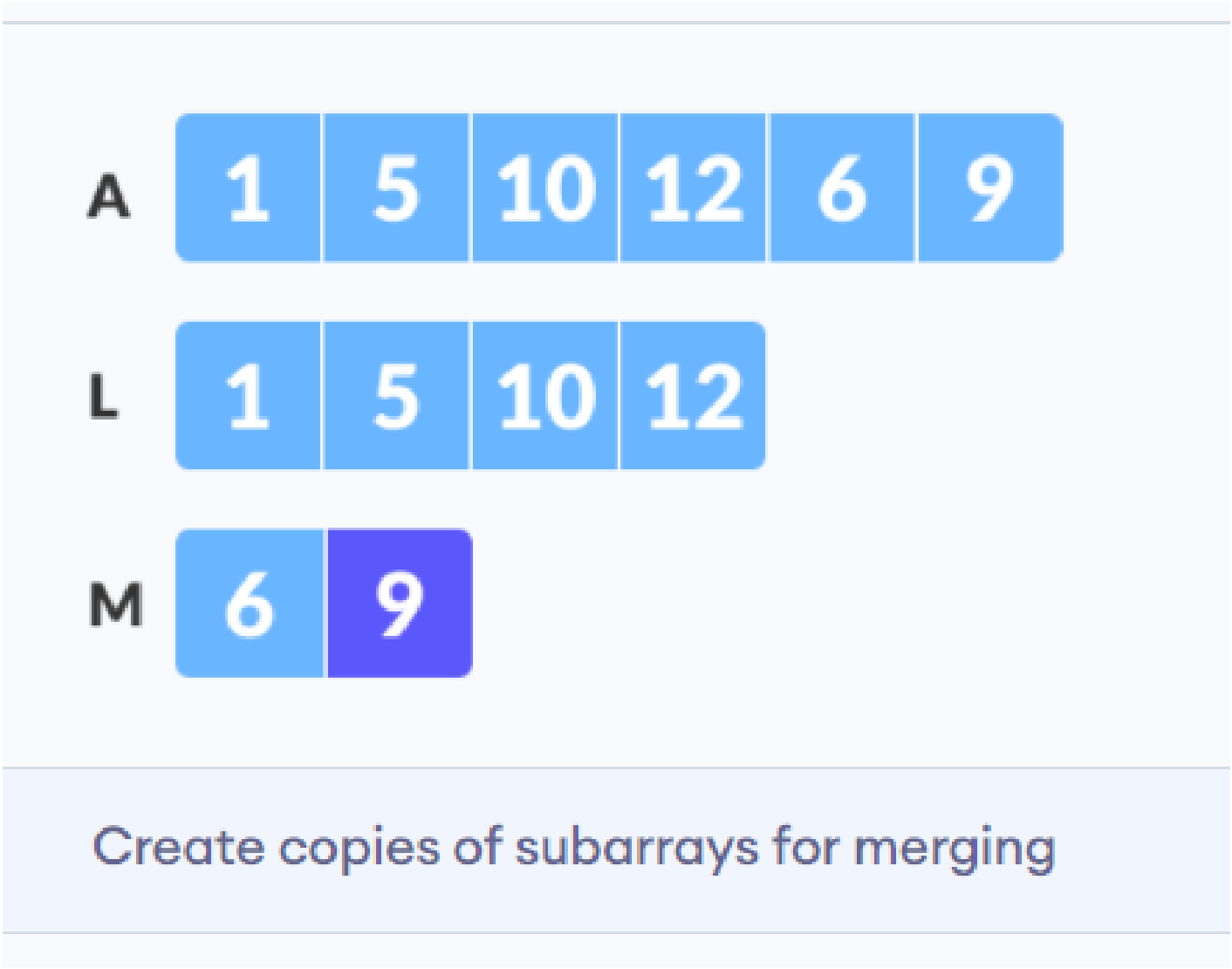
A lot is happening in this function, so let's take an example to see how this would work. As usual, a picture speaks a thousand words.



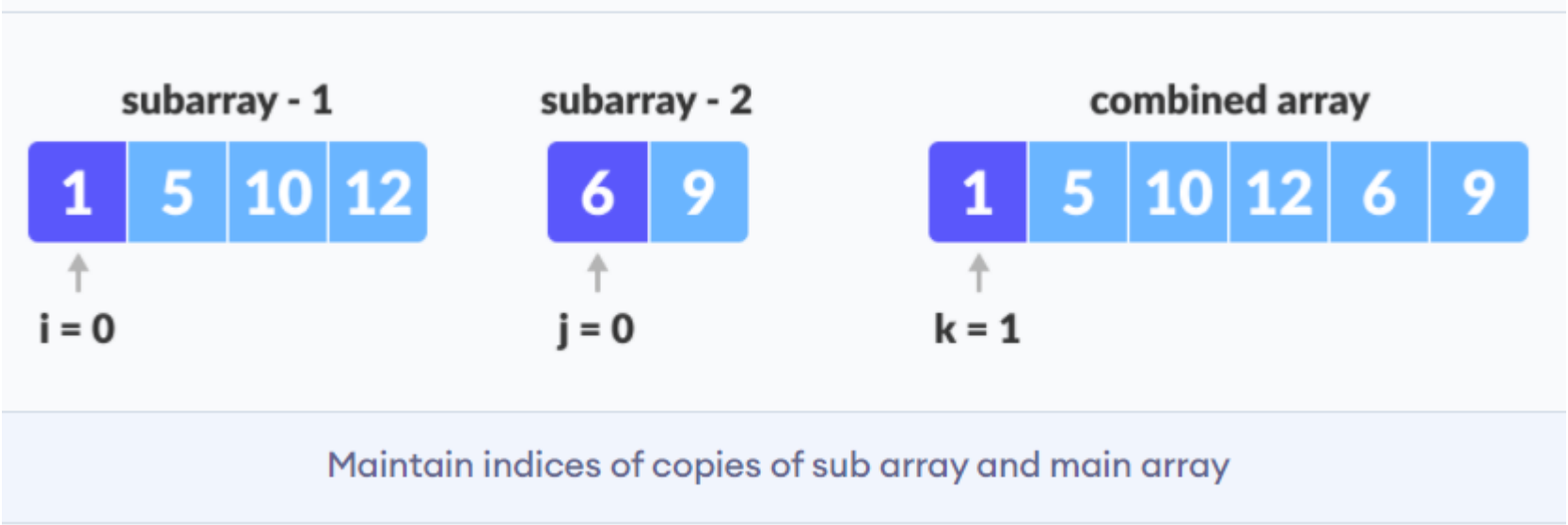
The array A[0..5] contains two sorted subarrays A[0..3] and A[4..5]. Let us see how the merge function will merge the two arrays.

void merge(int arr[], int p, int q, int r) { // Here, p = 0, q = 4, r = 5 (size of array)

Step 1: Create duplicate copies of sub-arrays to be sorted



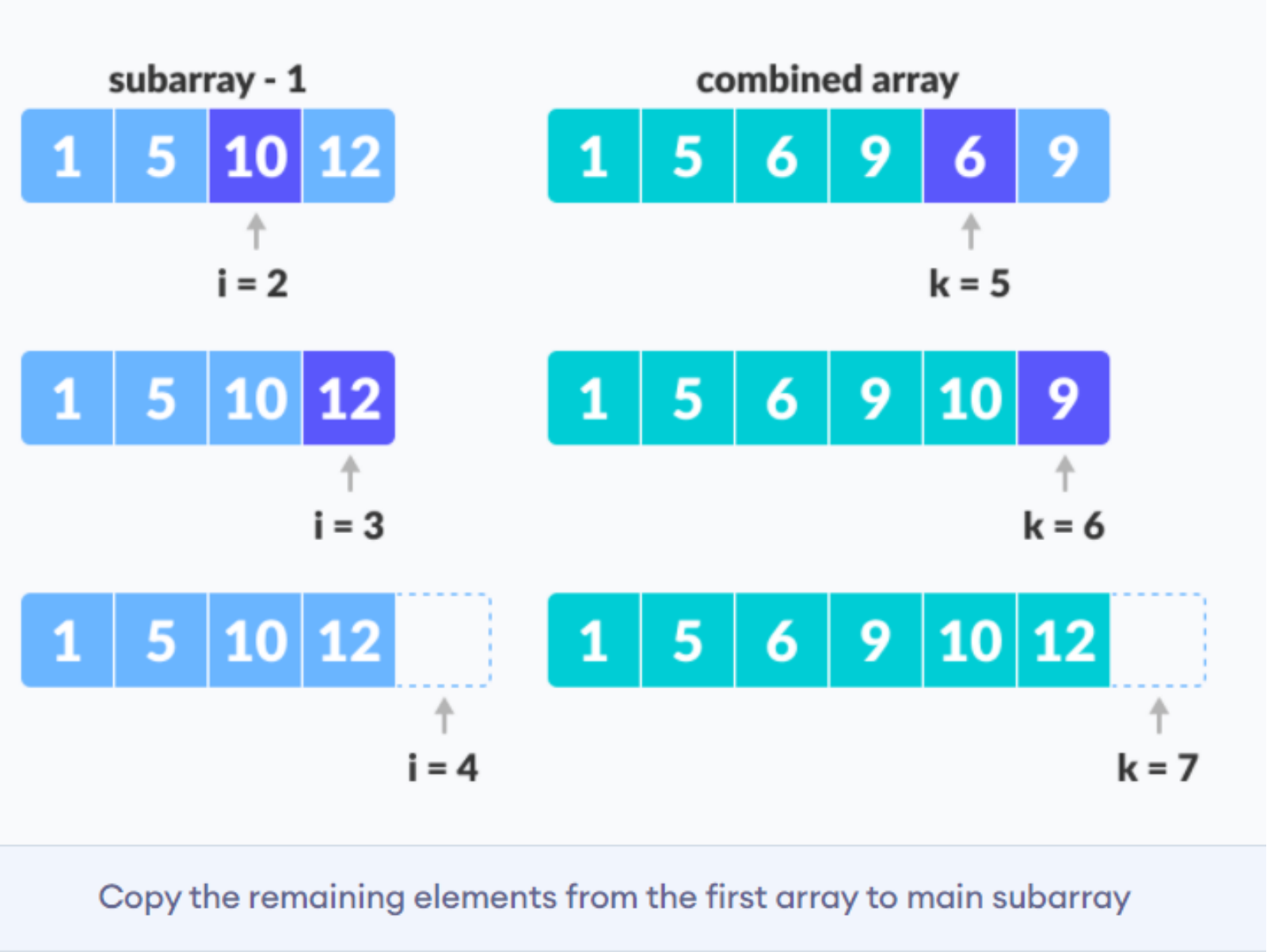
Step 2: Maintain current index of sub-arrays and main array

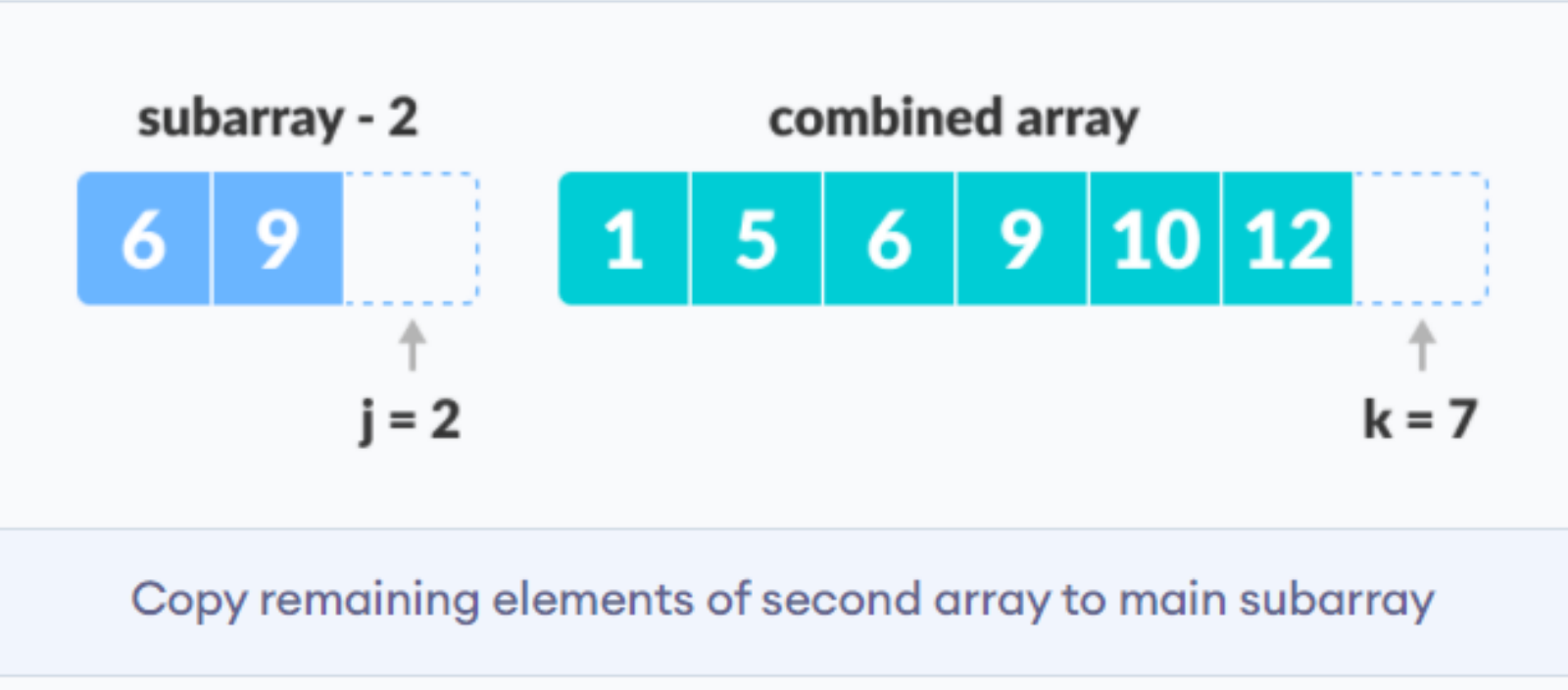


Step 3: Until we reach the end of either L or M, pick larger among elements L and M and place them in the correct position at A[p..r]

Comparing individual elements of sorted subarrays until we reach end of one

Step 4: When we run out of elements in either L or M, pick up the remaining elements and put in A[p..r]





This step would have been needed if the size of M was greater than L.

At the end of the merge function, the subarray A[p..r] is sorted.

Complexity Analysis of Merge Sort

As we have already learned in Binary Search that whenever we divide a number into half in every stpe, it can be represented using a logarithmic function, which is log n and the number of steps can be represented by log n + 1(at most)

Also, we perform a single step operation to find out the middle of any subarray, i.e. O(1).

And to **merge** the subarrays, made by dividing the original array of n elements, a running time of O(n) will be required.

Hence the total time for mergeSort function will become n(log n + 1), which gives us a time complexity of O(n*log n).

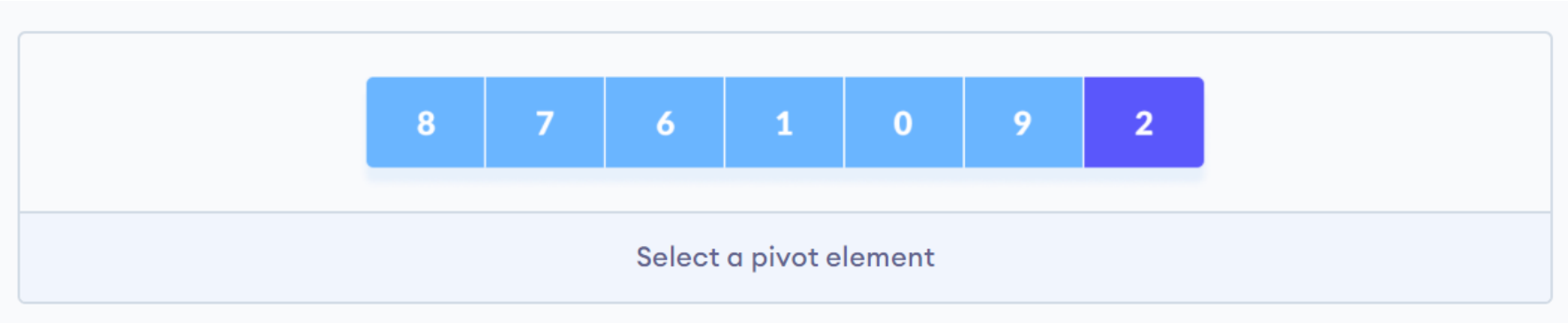
Quick Sort

QuickSort is one of the most efficient sorting algorithms and is based on the splitting of an array into smaller ones. The name comes from the fact that, quick sort is capable of sorting a list of data elements significantly faster than any of the common sorting algorithms. And like Merge sort, Quick sort also falls into the category of divide and conquer approach of problem-solving methodology.

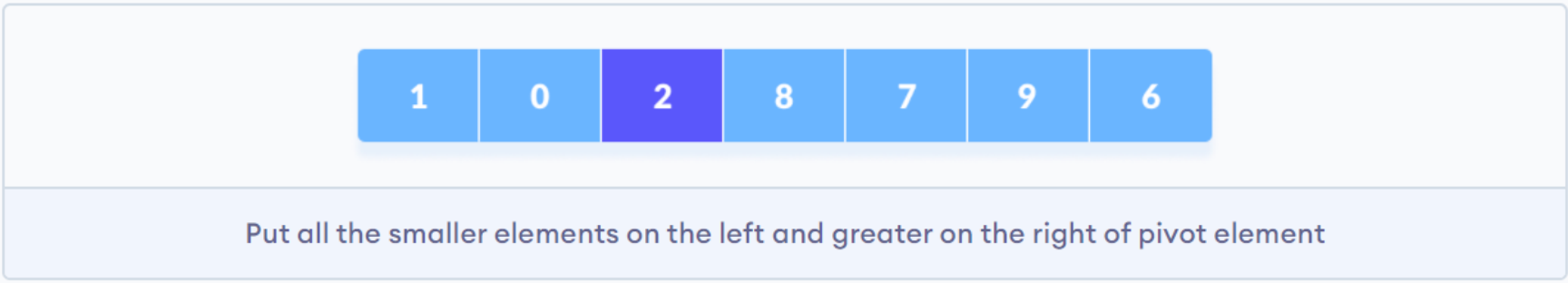
How QuickSort Works?

A pivot element is chosen from the array. You can choose any element from the array as the pivot element.

Here, we have taken the rightmost (ie. the last element) of the array as the pivot element.



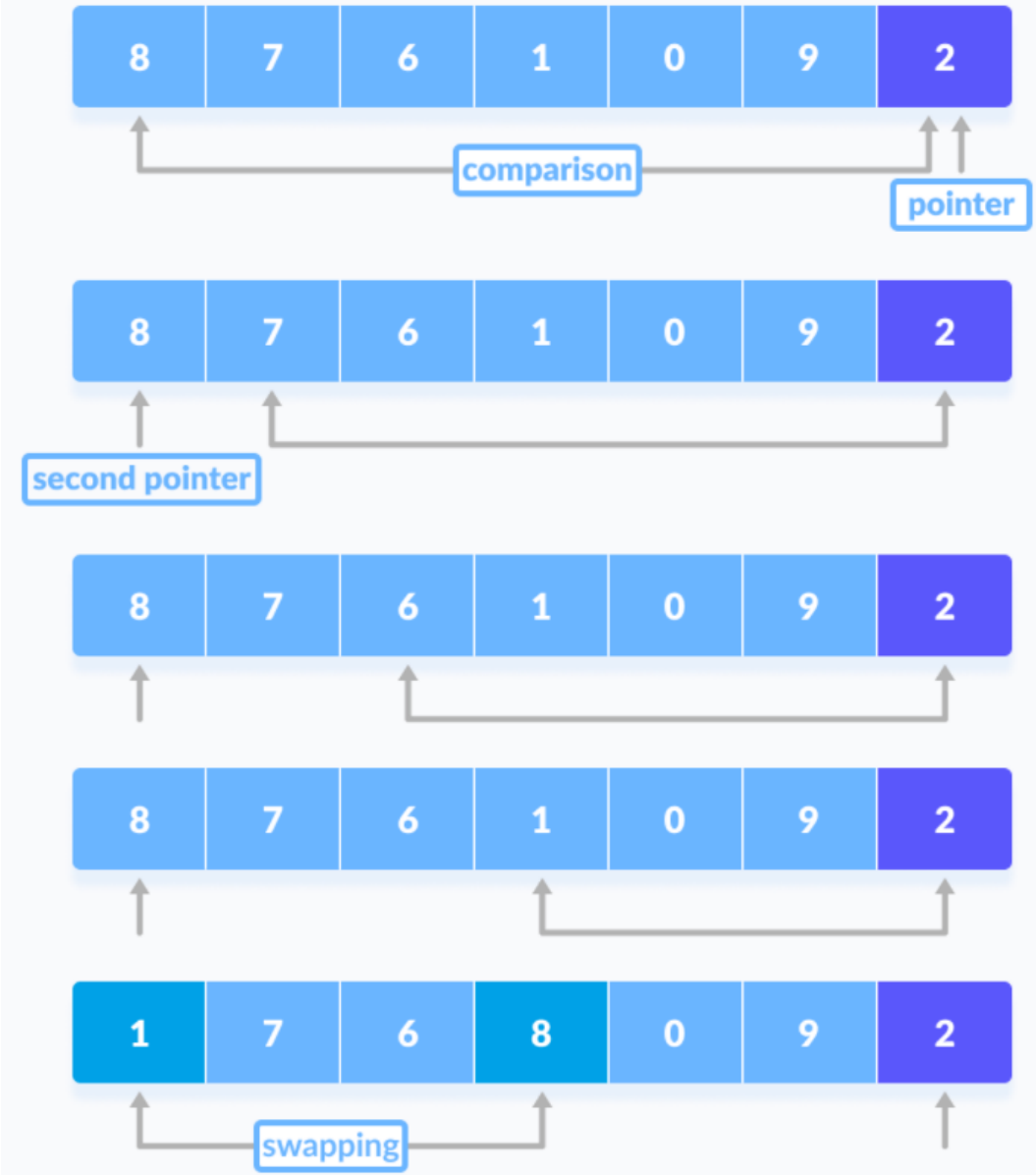
2. The elements smaller than the pivot element are put on the left and the elements greater than the pivot element are put on the right.



The above arrangement is achieved by the following steps.

(a) A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index. If the element greater than the pivot element is reached, a second pointer is set for that element.

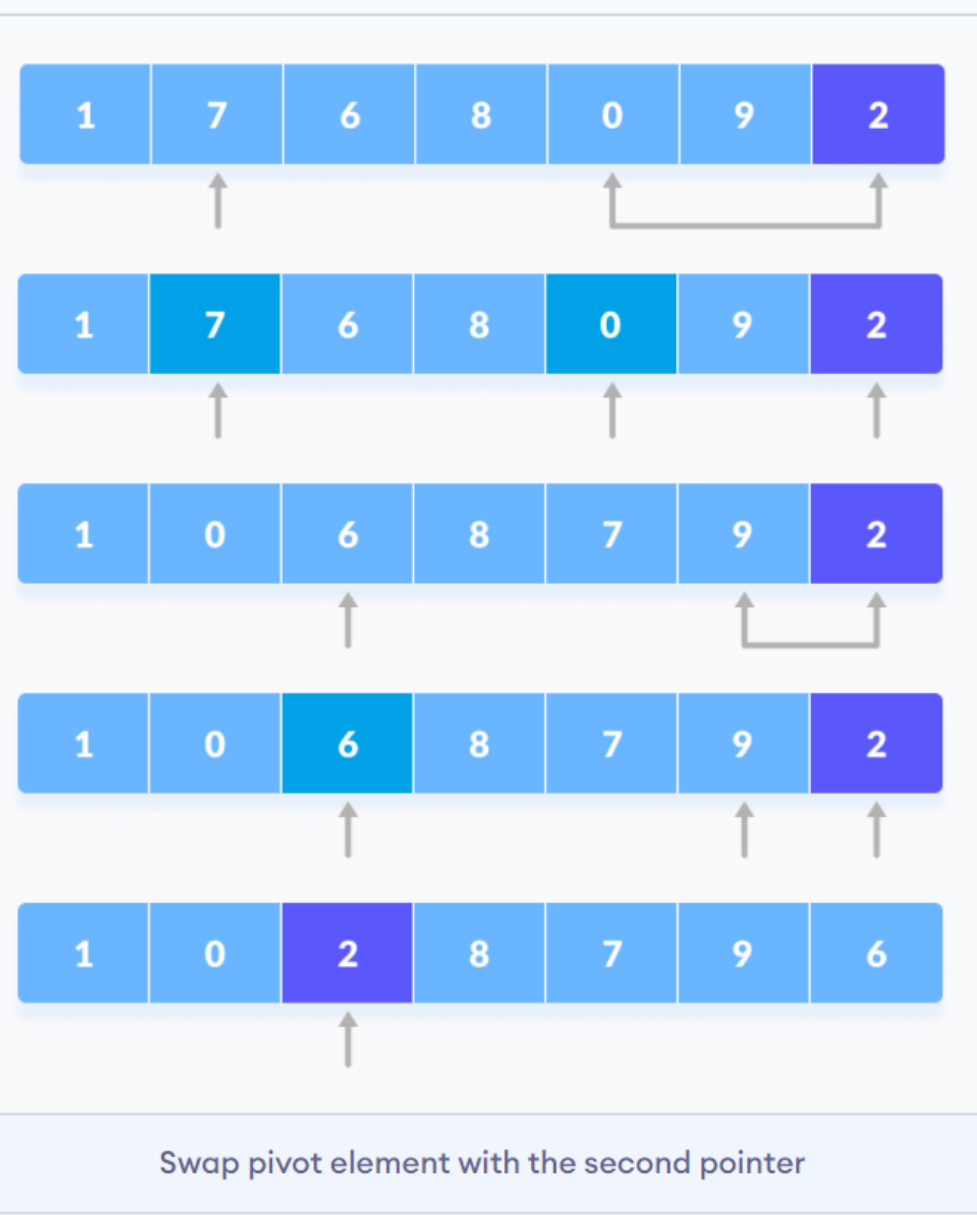
(b) Now, the pivot element is compared with the other elements (a third pointer). If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found earlier.



Comparison of pivot element with other elements

(c) The process goes on until the second last element is reached.

Finally, the pivot element is swapped with the second pointer.



Swap pivot element with the second pointer

(d) Now the left and right subparts of this pivot element are taken for further processing in the steps below.

3. Pivot elements are again chosen for the left and the right sub-parts separately. Within these sub-parts, the pivot elements are placed at their right position. Then, step 2 is repeated.

Quick Sort Algorithm

```
quickSort(array, leftmostIndex, rightmostIndex)
 if (leftmostIndex < rightmostIndex)
        pivotIndex <- partition(array,leftmostIndex, rightmostIndex)
        quickSort(array, leftmostIndex, pivotIndex)
        quickSort(array, pivotIndex + 1, rightmostIndex)
partition(array, leftmostIndex, rightmostIndex)
        set rightmostIndex as pivotIndex
        storeIndex <- leftmostIndex - 1
        for i <- leftmostIndex + 1 to rightmostIndex
        if element[i] < pivotElement
                swap element[i] and element[storeIndex]
        storeIndex++
        swap pivotElement and element[storeIndex+1]
return storeIndex + 1
```

Partition(arr, beg, end, loc)

Step -1: [Initailize] Set left=beg, right=end, loc=beg, flag=0

Step-2: Repeat steps 3 tp 6 while flag=0

Step 3: Repeat while arr[loc] <=arr[right] and loc!= right

Set right=right -1

[End of loop]

Step-4 : If loc=right

Set flag=1

Else

If arr[loc] > arr[right]

Swap arr[loc] with arr[right]

Set loc=right

[end of if]

Step 5: if flag=0

Repeat while arr[loc] >= arr[left] and loc != left

Set left = left + 1

[End of Loop]

Step-6: If loc=left

Set flag=1

Else

If arr[loc]< arr[left]

Swap arr[loc] with arr[left]

Set loc=left

[end of if]

[end of if]

Step7: [End of Loop]

Step 8: End

Quick_sort(arr, beg, end)

Step 1: if(beg<end)

call partition(arr, beg, end, loc)

call Quicksort(arr, beg, loc-1)

call quicksort(arr, loc+1, end)

[End of If]

Step2: End

Select pivot element of in each half and put at correct place using recursion

4. The sub-parts are again divided into smaller sub-parts until each subpart is formed of a single element.

5. At this point, the array is already sorted.

Quicksort uses recursion for sorting the sub-parts.

On the basis of Divide and conquer approach, quicksort algorithm can be explained as:

Divide

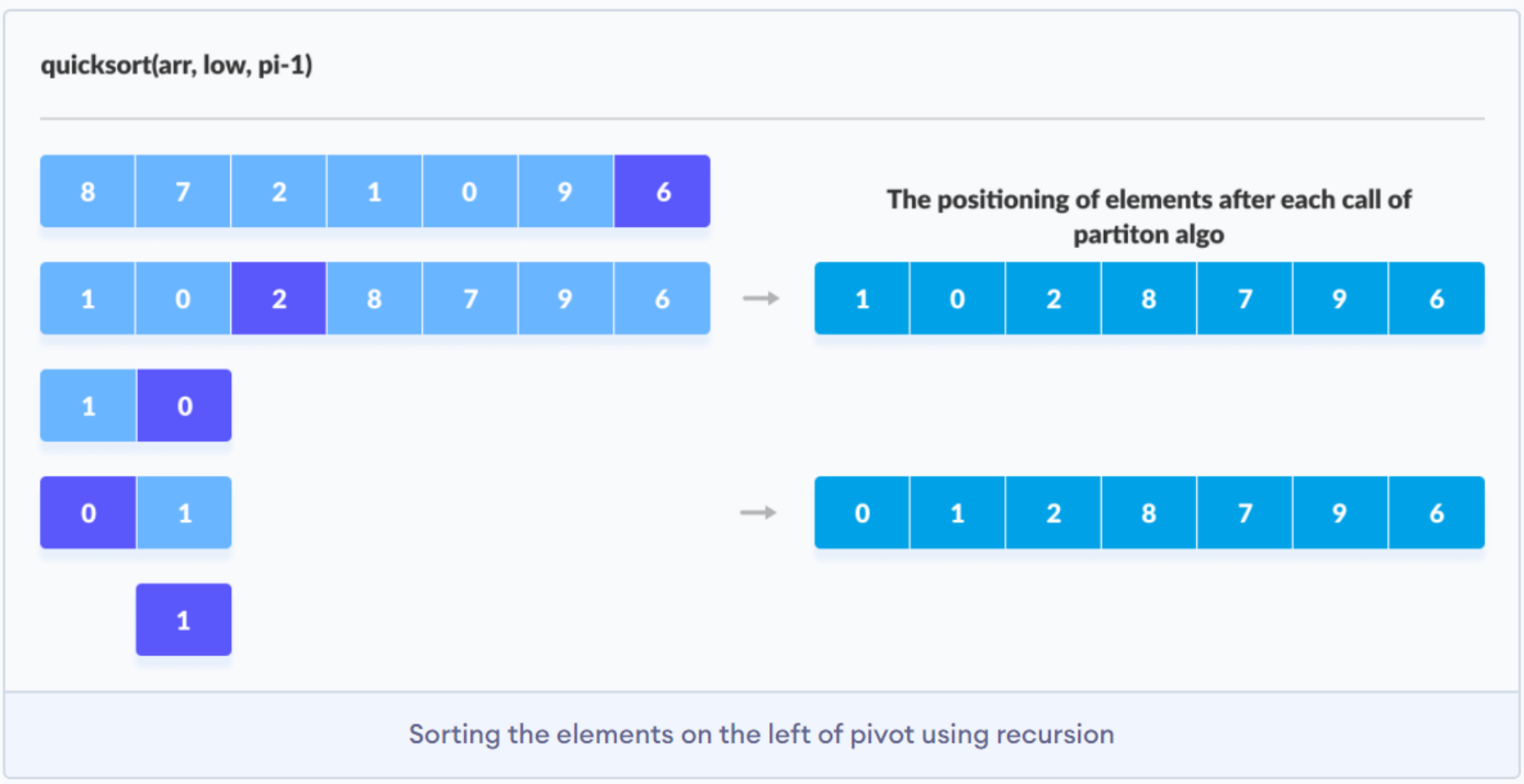
The array is divided into subparts taking pivot as the partitioning point. The elements smaller than the pivot are placed to the left of the pivot and the elements greater than the pivot are placed to the right.

Conquer

The left and the right subparts are again partitioned using the by selecting pivot elements for them. This can be achieved by recursively passing the subparts into the algorithm.

Combine

This step does not play a significant role in quicksort. The array is already sorted at the end of the conquer step.



Sorting the elements on the left of pivot using recursion

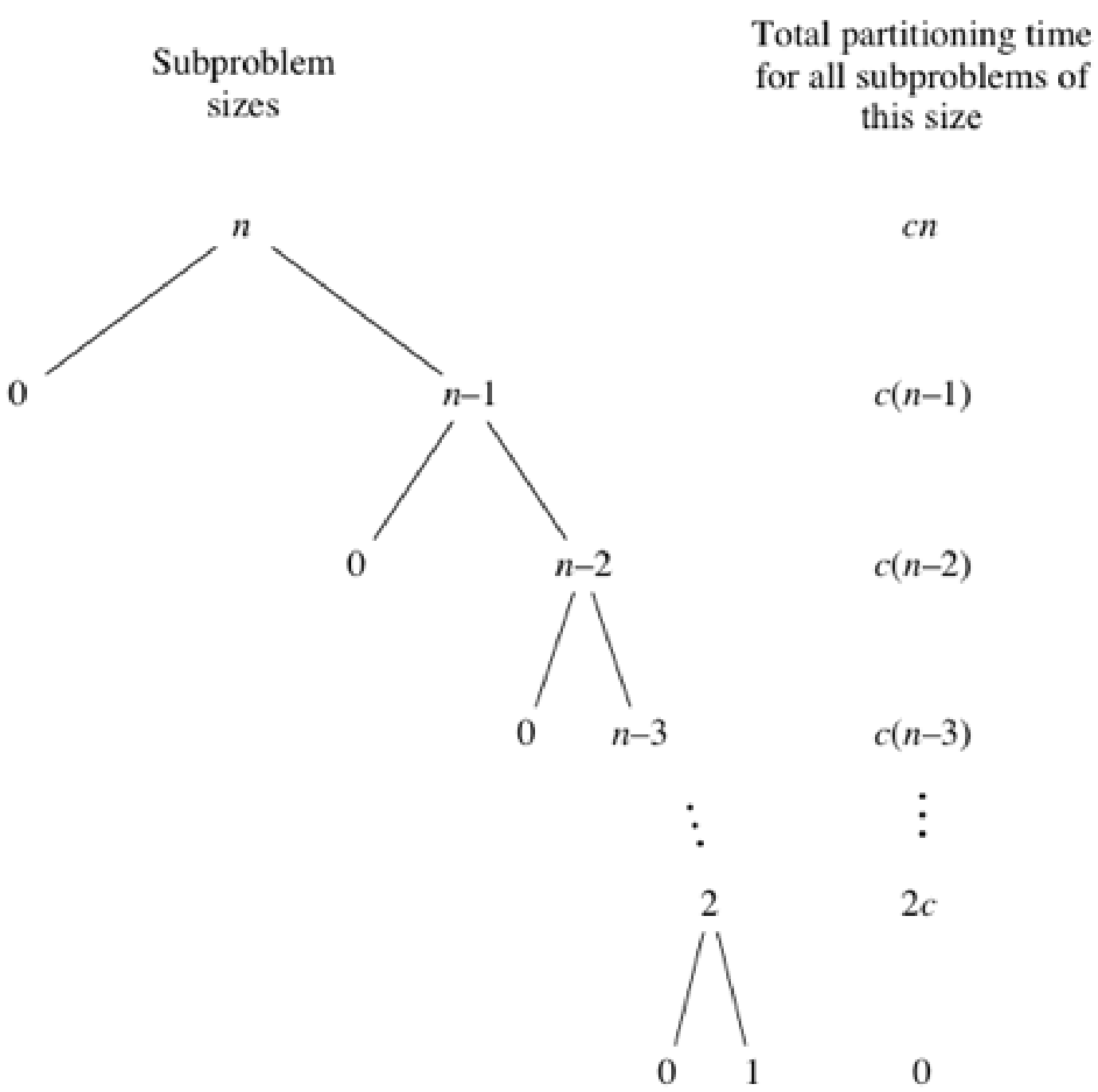
Sorting the elements on the right of pivot using recursion

Analysis of quicksort

suppose that the pivot chosen by the partition function is always either the smallest or the largest element in the *n*-element subarray. Then one of the partitions will contain no elements and the other partition will contain $n-1$ elements—all but the pivot. So the recursive calls will be on subarrays of sizes 0 and $n-1$.

Worst-case running time

When quicksort always has the most unbalanced partitions possible, then the original call takes cn time for some constant c, the recursive call on $n-1$ elements takes $c(n-1)$ time, the recursive call on $n-2$ elements takes $c(n-2)$ time, and so on. Here's a tree of the sub problem sizes with their partitioning times:


Subproblem sizes
Total partitioning time for all subproblems of this size
n
cn
0
n−1
c(n−1)
0
n−2
c(n−2)
0
n−3
c(n−3)
2
2c
0
1
0

When we total up the partitioning times for each level, we get

$$cn+c(n-1)+c(n-2)+\cdots+2c$$

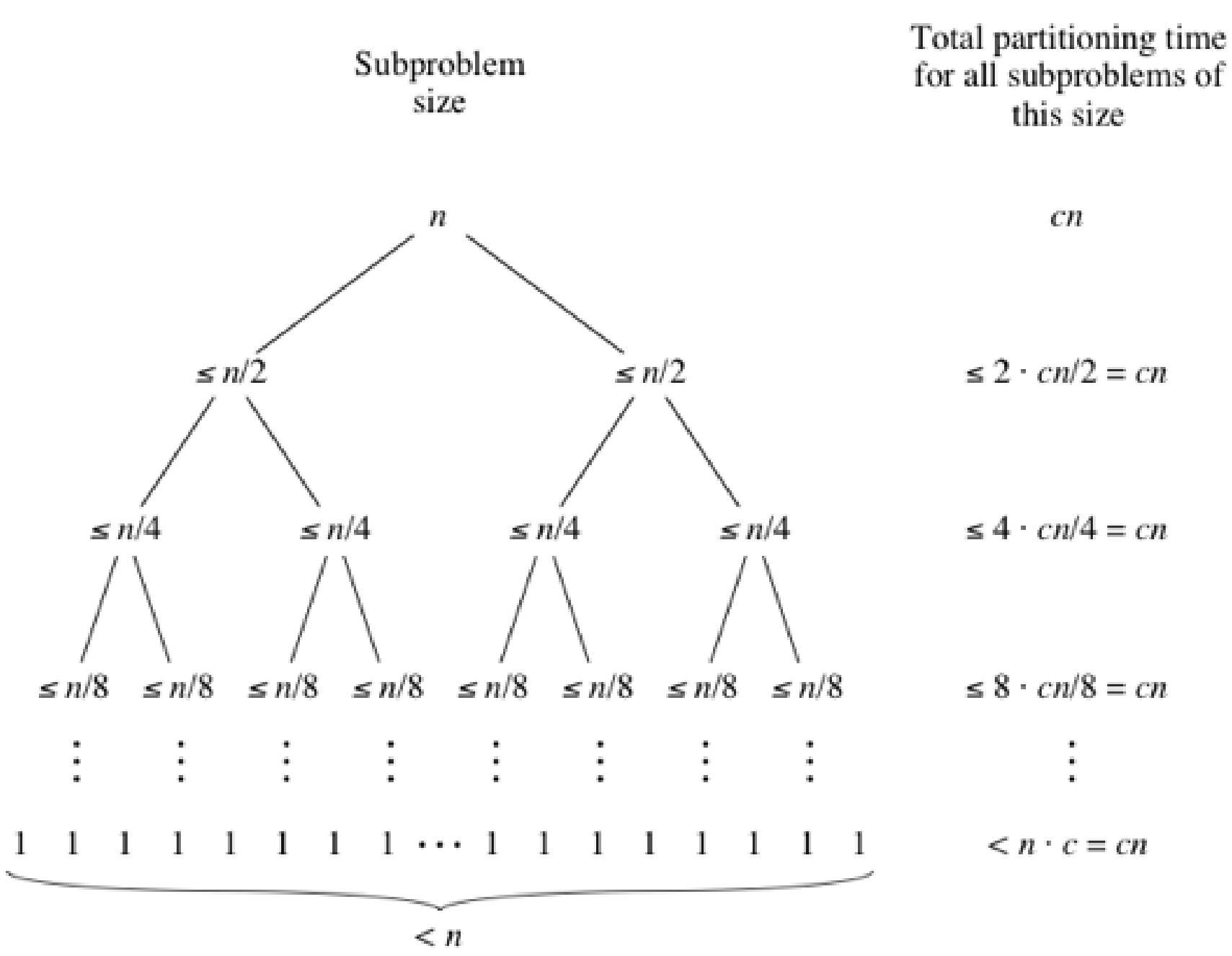
$$=c(n+(n-1)+(n-2)+\cdots+2)$$

$$=c((n+1)(n/2)-1)$$

The last line is because 1 + 2 + 3 + …..+ n is the arithmetic series (We subtract 1 because for quicksort, the summation starts at 2, not 1.) We have some low-order terms and constant coefficients, but when we use big-Θ notation, we ignore them. In big-Θ notation, quicksort's worst-case running time is $\Theta(n^2)$.

Best-case running time

Quicksort's best case occurs when the partitions are as evenly balanced as possible: their sizes either are equal or are within 1 of each other. The former case occurs if the subarray has an odd number of elements and the pivot is right in the middle after partitioning, and each partition has (n-1)/2. The latter case occurs if the subarray has an even number n of elements and one partition has $n/2$ elements with the other having $(n/2)-1$. In either of these cases, each partition has at most $n/2$ elements, and the tree of subproblem sizes looks a lot like the tree of subproblem sizes for merge sort, with the partitioning times looking like the merging times:


Subproblem size
Total partitioning time for all subproblems of this size
n
cn
≤ n/2
≤ n/2
≤ 2 · cn/2 = cn
≤ n/4
≤ n/4
≤ n/4
≤ n/4
≤ 4 · cn/4 = cn
≤ n/8 ≤ n/8 ≤ n/8 ≤ n/8 ≤ n/8 ≤ n/8 ≤ n/8 ≤ n/8
≤ 8 · cn/8 = cn
1 1 1 1 1 1 1 1 ··· 1 1 1 1 1 1 1 1
< n · c = cn
< n

Using big-Θ notation, we get the same result as for merge sort: $\Theta(n\log_2 n)$.

Selection Sort

1. Selection sort is one of the easiest approaches to sorting.
2. It is inspired from the way in which we sort things out in day to day life.
3. It is an in-place sorting algorithm because it uses no auxiliary data structures while sorting

How Selection Sort Works?

Consider the following elements are to be sorted in ascending order using selection sort-

6, 2, 11, 7, 5

Selection sort works as-

- It finds the first smallest element (2).
- It swaps it with the first element of the unordered list.
- It finds the second smallest element (5).
- It swaps it with the second element of the unordered list.

Similarly, it continues to sort the given elements.

As a result, sorted elements in ascending order are-

2, 5, 6, 7, 11

Algorithm

Step 1 − Set MIN to location 0

Step 2 − Search the minimum element in the list

Step 3 − Swap with value at location MIN

Step 4 − Increment MIN to point to next element

Step 5 − Repeat until list is sorted

```
for (i = 0 ; i < n-1 ; i++)
{
        index = i;
        for(j = i+1 ; j < n ; j++)
        {
                if(A[j] < A[index])
                index = j;
         }
        temp = A[i];
        A[i] = A[index];
        A[index] = temp;
}
```

Here,

i = variable to traverse the array A

index = variable to store the index of minimum element

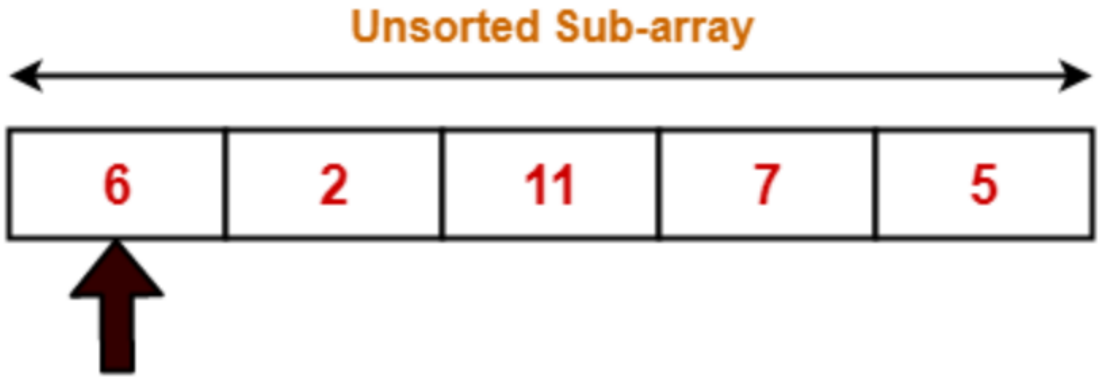
j = variable to traverse the unsorted sub-array

temp = temporary variable used for swapping

Consider the following elements are to be sorted in ascending order-

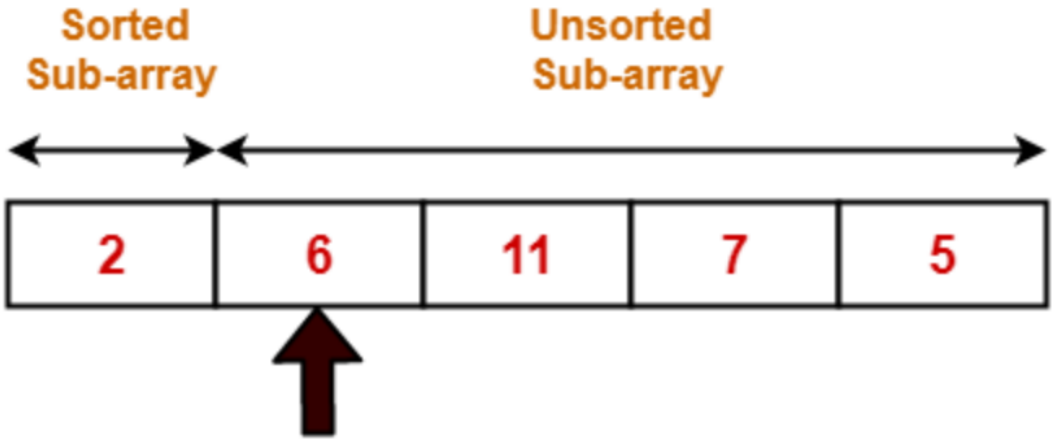
6, 2, 11, 7, 5

Step-01: For i = 0



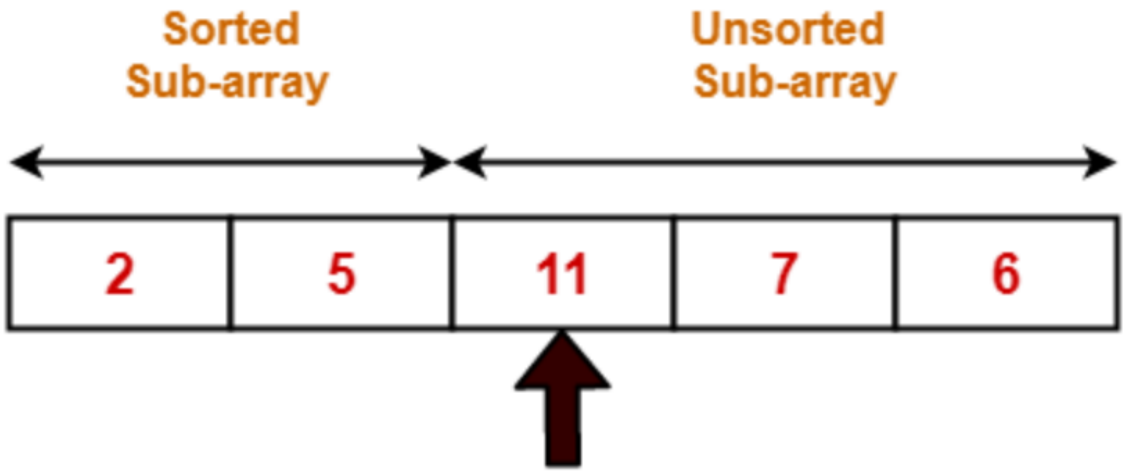
We start here, find the minimum element and swap it with the 1st element of array

Step-02: For i = 1



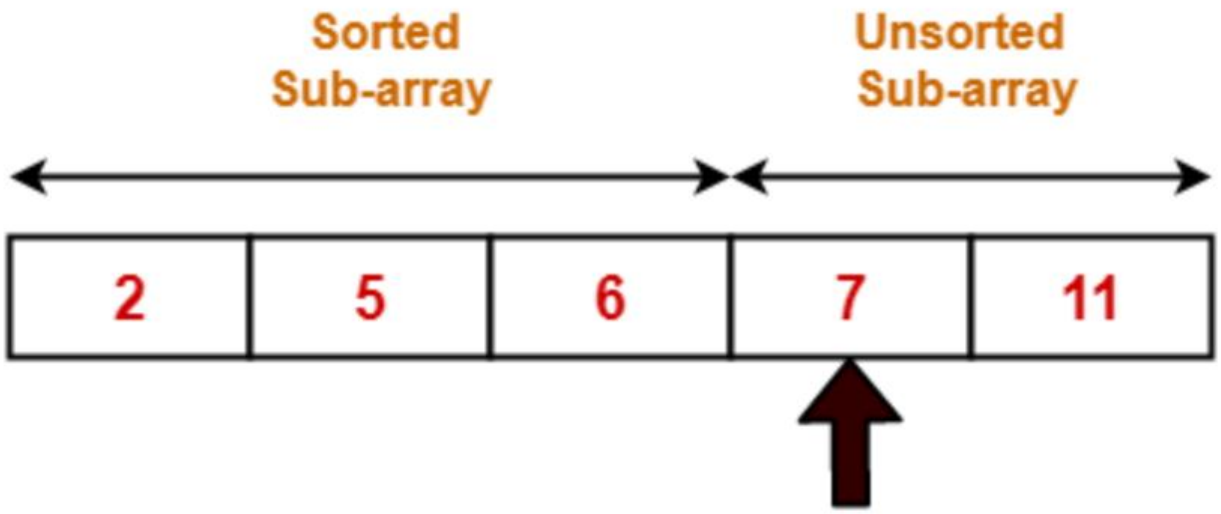
We start here, find the minimum element and swap it with the 2nd element of array

Step-03: For i = 2



We start here, find the minimum element and swap it with the 3rd element of array

Step-04: For i = 3

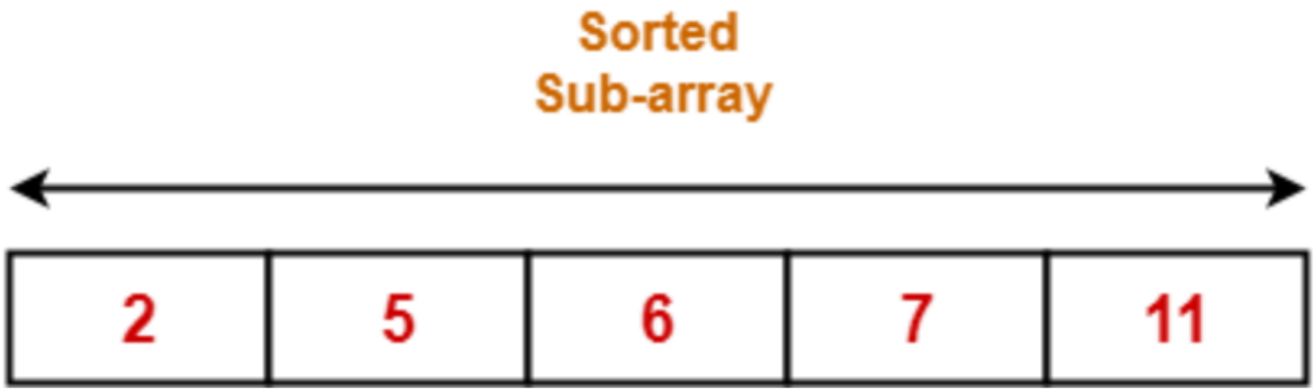


We start here, find the minimum element but there is no need to swap
(4th element is itself the minimum)

Step-05: For i = 4

Loop gets terminated as ‘i’ becomes 4.

The state of array after the loops are finished is as shown-



With each loop cycle,

The minimum element in unsorted sub-array is selected.

It is then placed at the correct location in the sorted sub-array until array A is completely sorted.

Time Complexity Analysis-

- Selection sort algorithm consists of two nested loops.
- Owing to the two nested loops, it has $O(n^2)$ time complexity

	Time Complexity
Best Case	n^2
Average Case	n^2
Worst Case	n^2

Important Notes-

- Selection sort is not a very efficient algorithm when data sets are large.
- This is indicated by the average and worst case complexities.
- Selection sort uses minimum number of swap operations O(n) among all the sorting algorithms.

Time Complexity:

To find the minimum element from the array of *N* elements, *N−1* comparisons are required. After putting the minimum element in its proper position, the size of an unsorted array reduces to *N−1* and then *N−2* comparisons are required to find the minimum in the unsorted array.

Therefore

$$(N-1) + (N-2) + ... + 1 = (N \cdot (N-1))/2$$

comparisons and *N* swaps result in the overall complexity of $O(N^2)$.

Heap Sort Algorithm

Heap Sort is a popular and efficient sorting algorithm in computer programming. Learning how to write the heap sort algorithm requires knowledge of two types of data structures - arrays and trees.

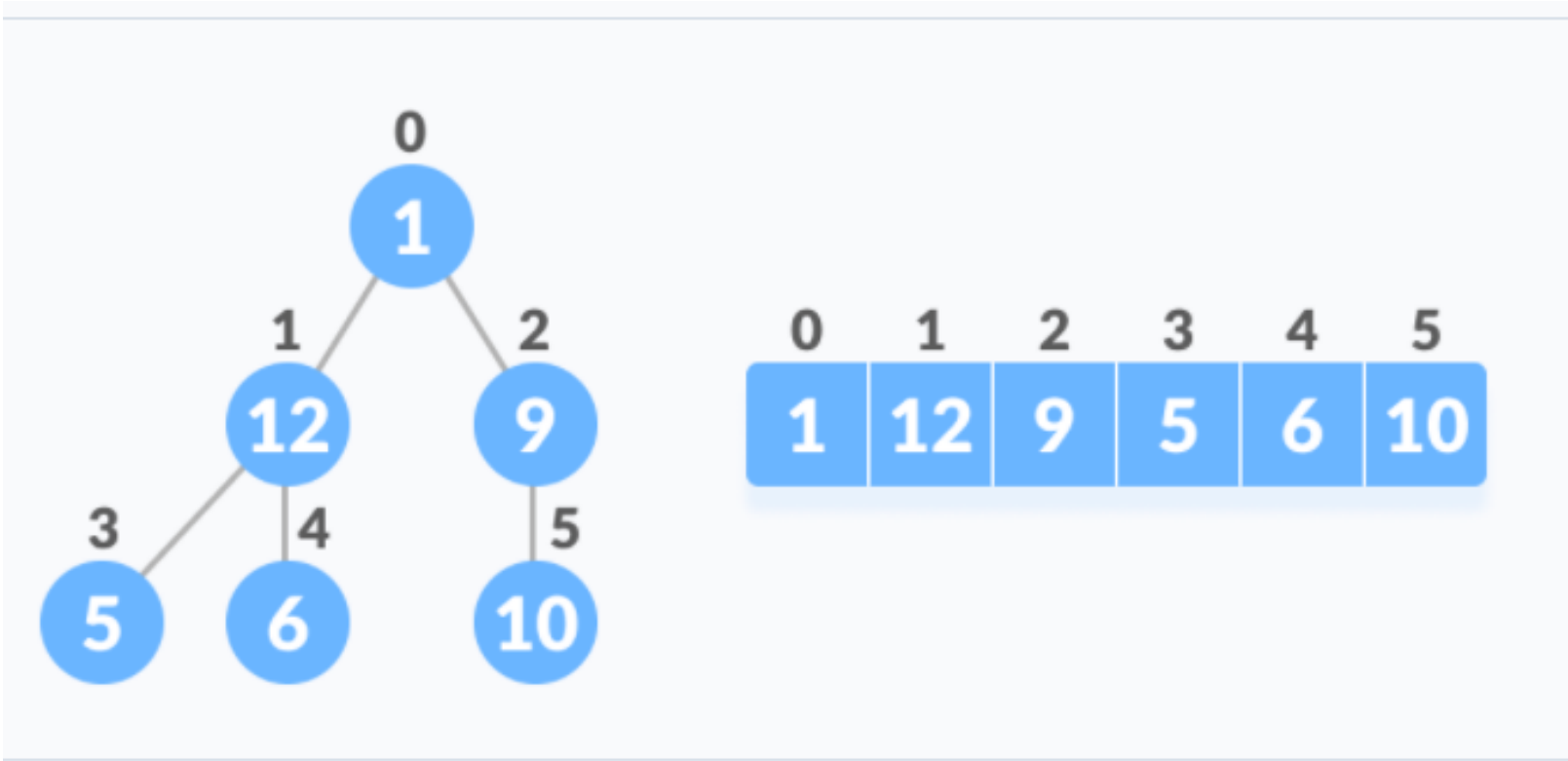
The initial set of numbers that we want to sort is stored in an array e.g. [10, 3, 76, 34, 23, 32] and after sorting, we get a sorted array [3,10,23,32,34,76]

Heap sort works by visualizing the elements of the array as a special kind of complete binary tree called a heap.

Relationship between Array Indexes and Tree Elements

A complete binary tree has an interesting property that we can use to find the children and parents of any node.

If the index of any element in the array is i, the element in the index 2i+1 will become the left child and element in 2i+2 index will become the right child. Also, the parent of any element at index i is given by the lower bound of (i-1)/2.



Relationship between array and heap indices

Let's test it out,

Left child of 1 (index 0)

= element in (2*0+1) index

= element in 1 index

= 12 Right child of 1

= element in (2*0+2) index

= element in 2 index

= 9 Similarly, Left child of 12 (index 1)

= element in (2*1+1) index

= element in 3 index

= 5 Right child of 12

= element in (2*1+2) index

= element in 4 index = 6

Let us also confirm that the rules hold for finding parent of any node

Parent of 9 (position 2)

= (2-1)/2

= ½ = 0.5 ~ 0 index

= 1

Parent of 12 (position 1)

= (1-1)/2

= 0 index

= 1

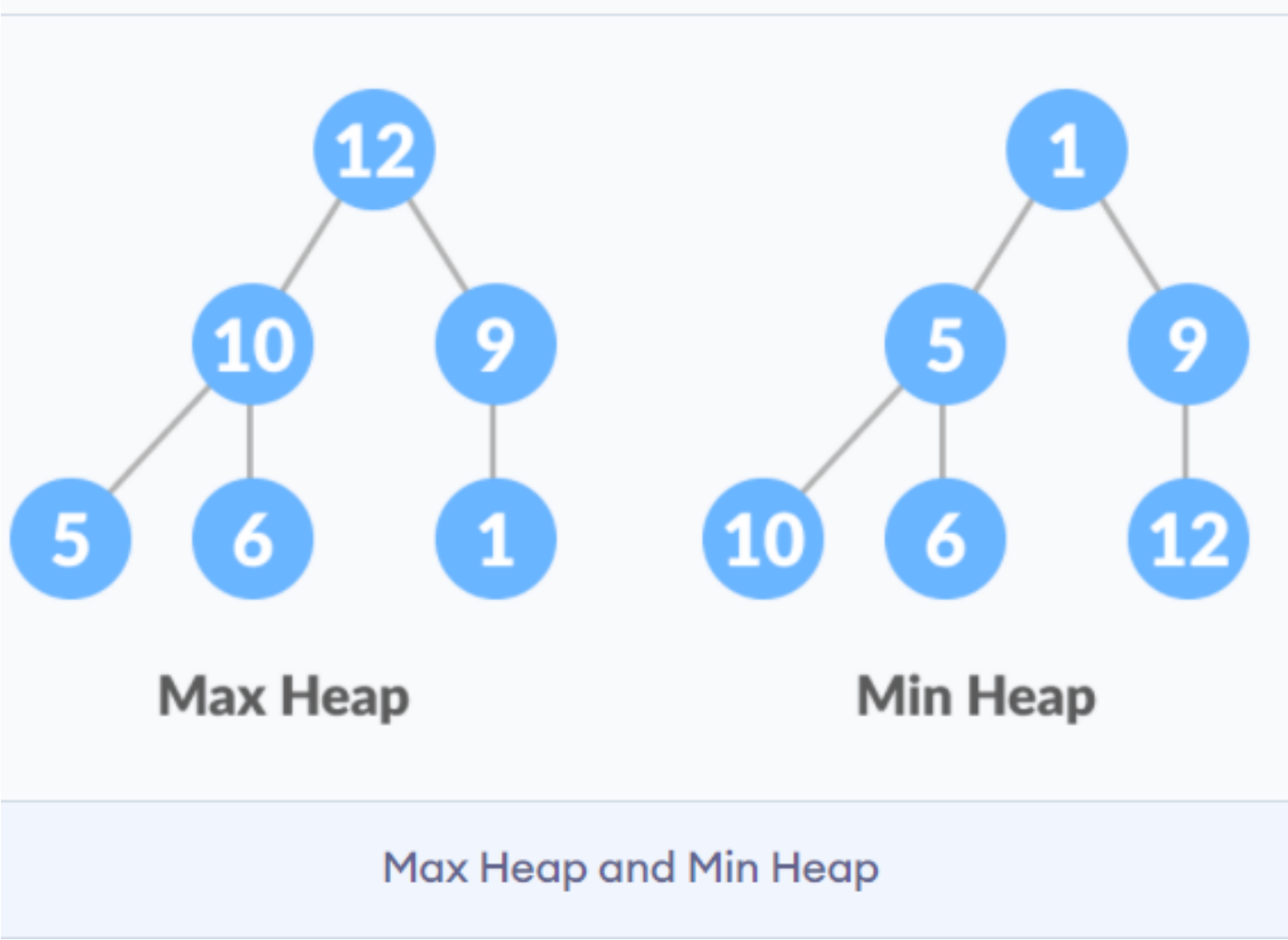
Understanding this mapping of array indexes to tree positions is critical to understanding how the Heap Data Structure works and how it is used to implement Heap Sort.

What is Heap Data Structure?

Heap is a special tree-based data structure. A binary tree is said to follow a heap data structure if it is a complete binary tree

All nodes in the tree follow the property that they are greater than their children i.e. the largest element is at the root and both its children and smaller than the root and so on. Such a heap is called a max-heap. If instead, all nodes are smaller than their children, it is called a min-heap

The following example diagram shows Max-Heap and Min-Heap.

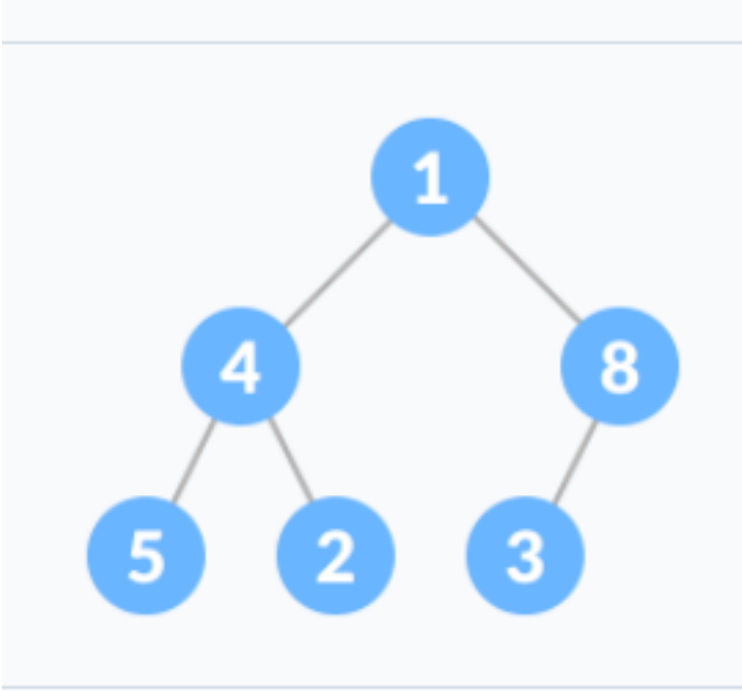


Max Heap and Min Heap

Heap Data Structure

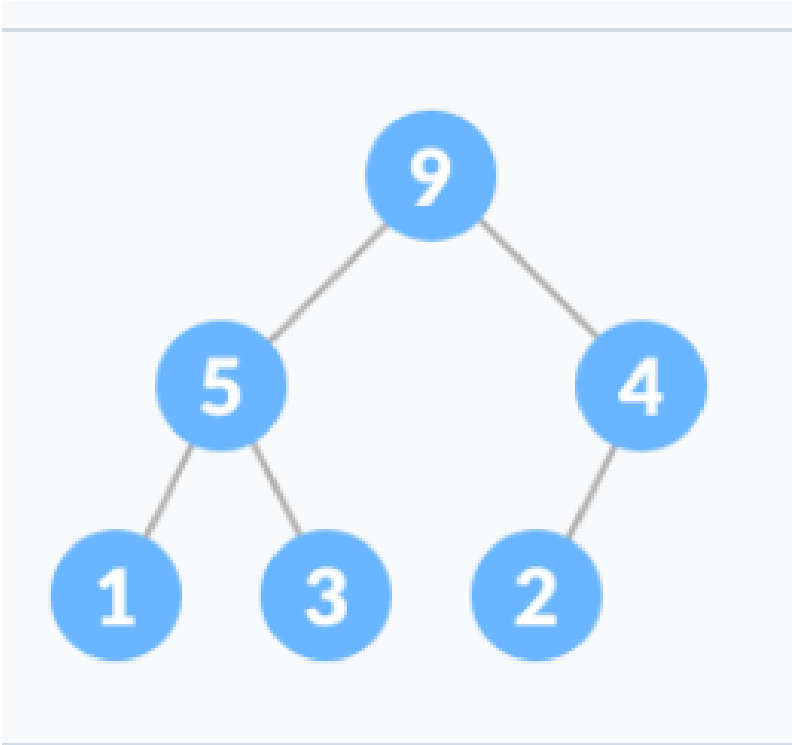
Heap data structure is a complete binary tree that satisfies **the heap property**. It is also called as **a binary heap**.

- A complete binary tree is a special binary tree in which every level, except possibly the last, is filled
- all the nodes are as far left as possible

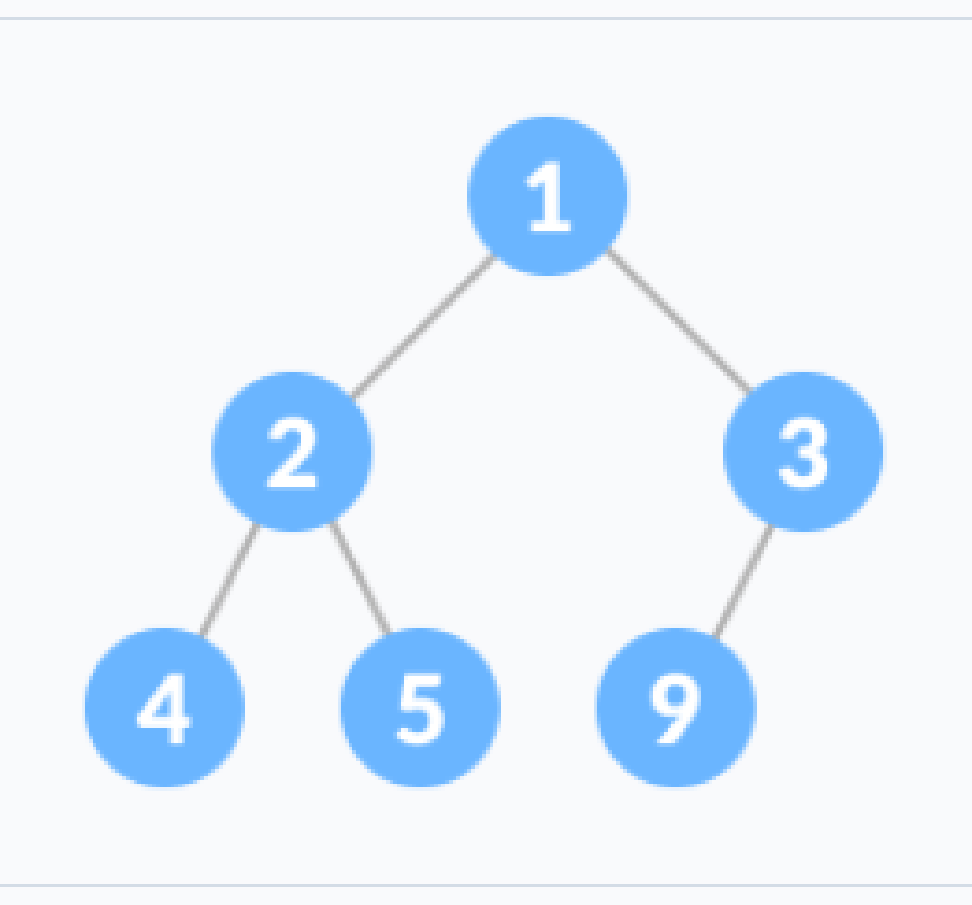


Heap Property is the property of a node in which

(for max heap) key of each node is always greater than its child node/s and the key of the root node is the largest among all other nodes;



(for min heap) key of each node is always smaller than the child node/s and the key of the root node is the smallest among all other nodes.



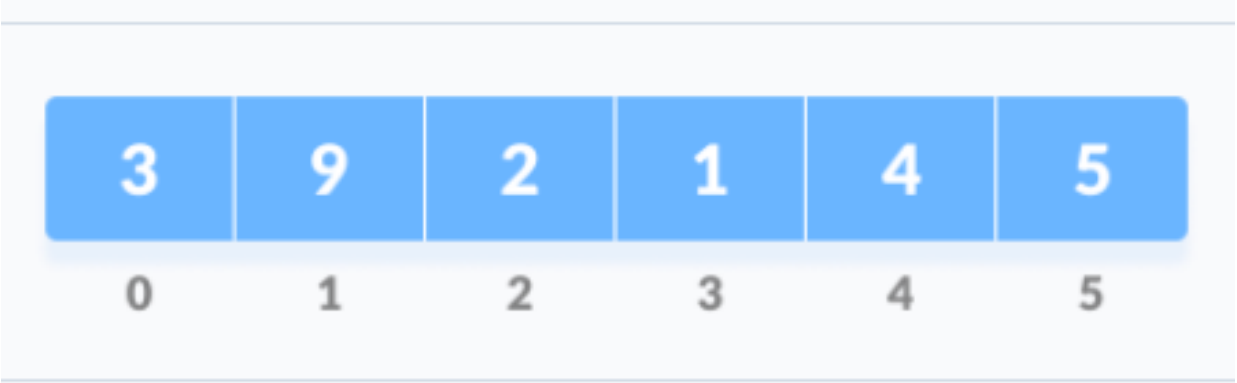
Heap Operations

Some of the important operations performed on a heap are described below along with their algorithms.

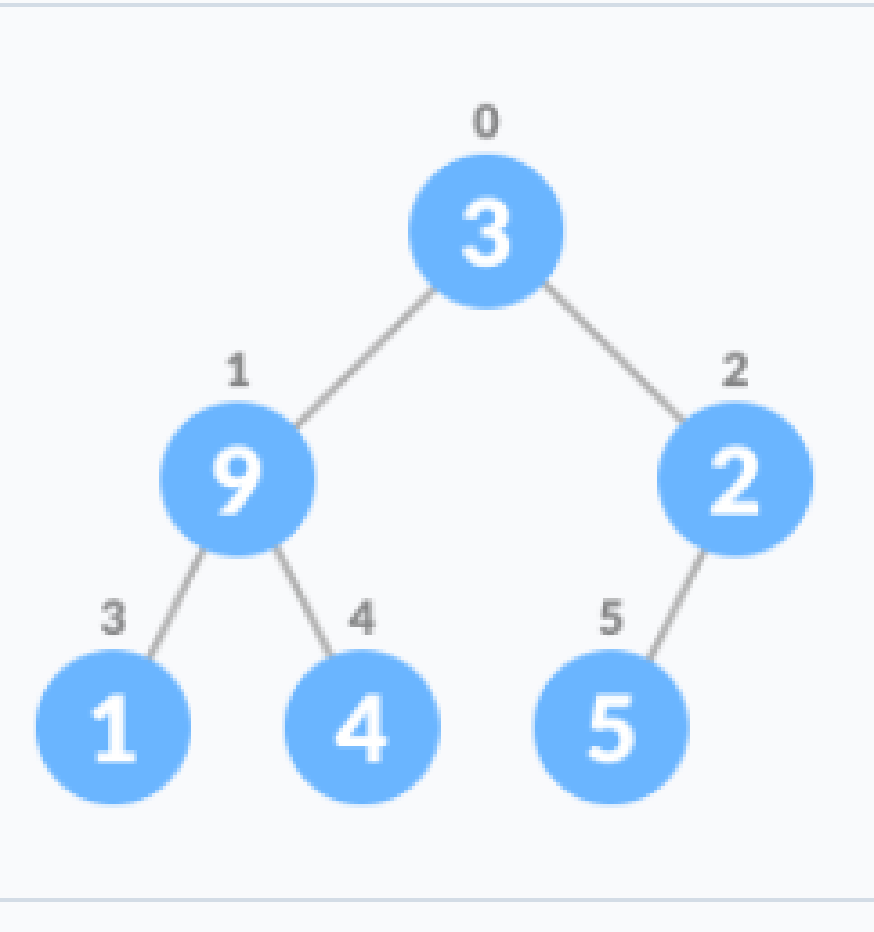
Heapify

Heapify is the process of creating a heap data structure from a binary tree. It is used to create a Min-Heap or a Max-Heap.

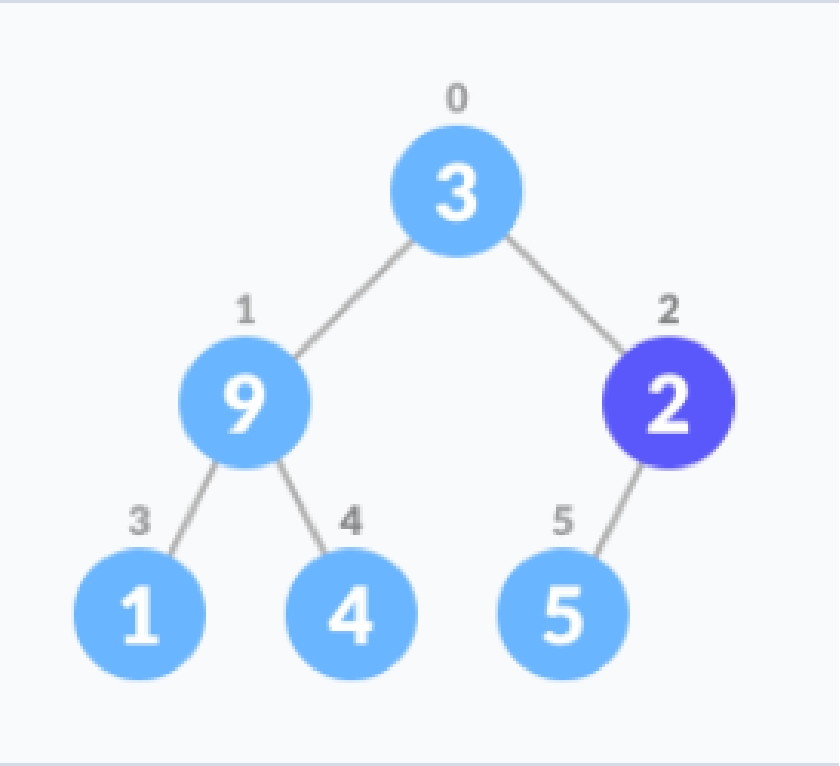
1. Let the input array be



2. Create a complete binary tree from the array



3. Start from the first index of non-leaf node whose index is given by n/2 – 1



4. Set current element i as largest.

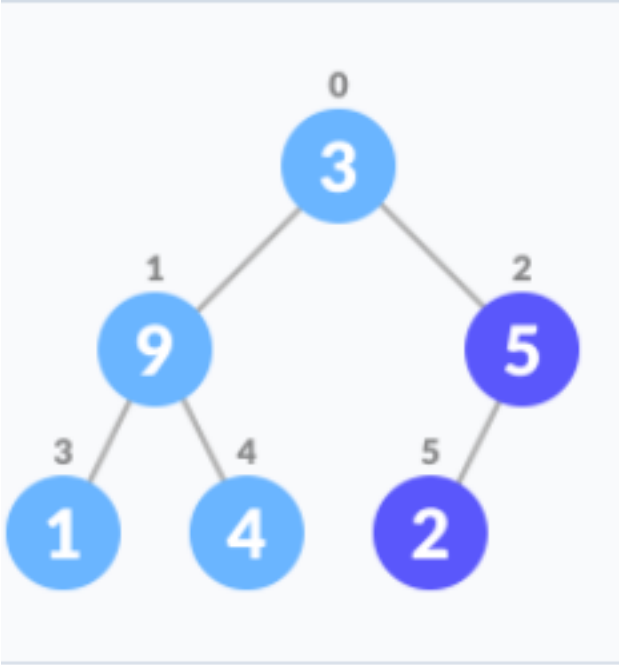
5. The index of left child is given by 2i + 1 and the right child is given by 2i + 2.

If leftChild is greater than currentElement (i.e. element at ith index), set leftChildIndex as largest.

If rightChild is greater than element in largest,

set rightChildIndex as largest.

6. Swap largest with currentElement



7. Repeat steps 3-7 until the subtrees are also heapified.

How to "heapify" a tree

Starting from a complete binary tree, we can modify it to become a Max-Heap by running a function called heapify on all the non-leaf elements of the heap.

Since heapify uses recursion, it can be difficult to grasp. So let's first think about how you would heapify a tree with just three elements.

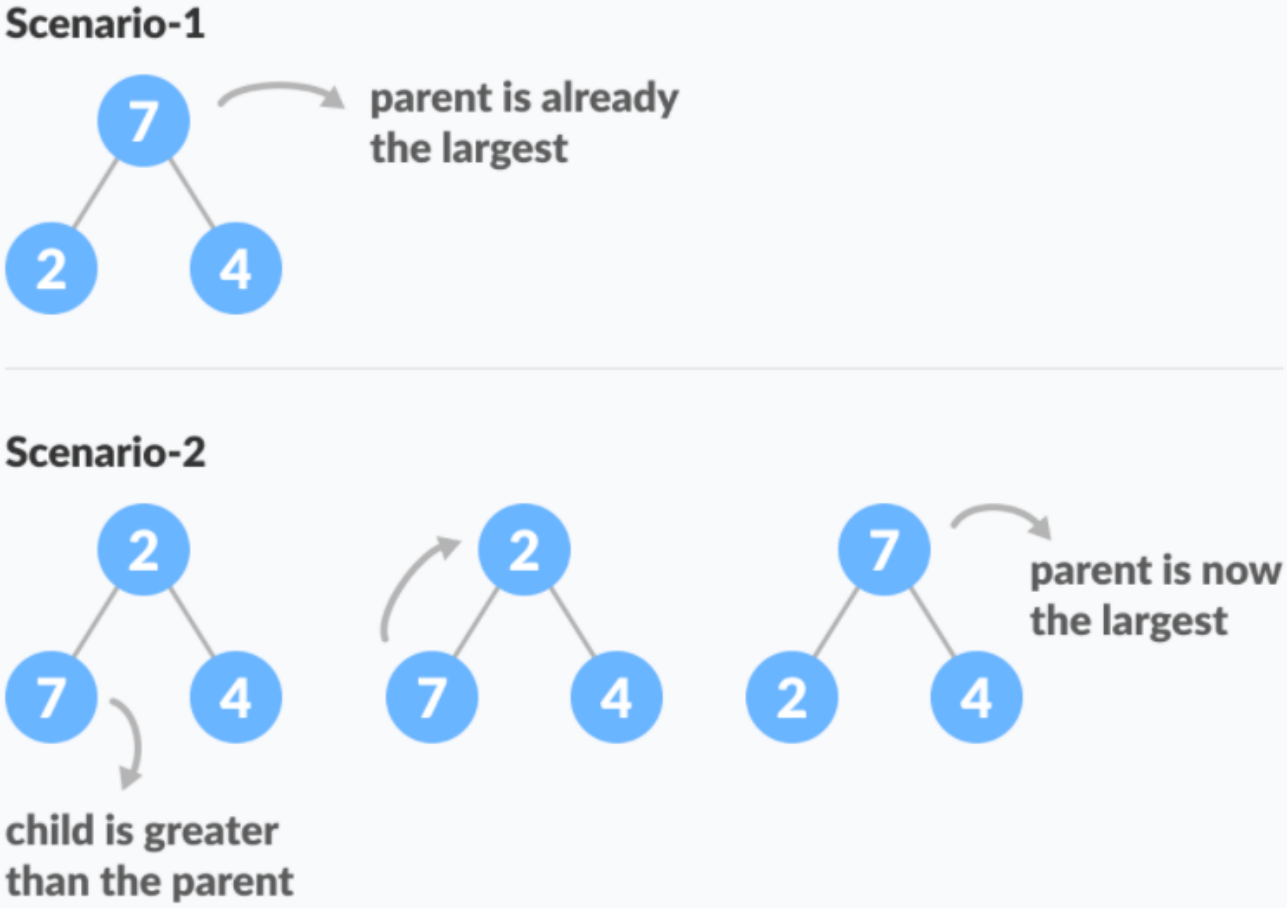
```
heapify(array)

Root = array[0]

Largest = largest( array[0] , array [2*0 + 1].array[2*0+2])

if(Root != Largest)

        Swap(Root, Largest)
```

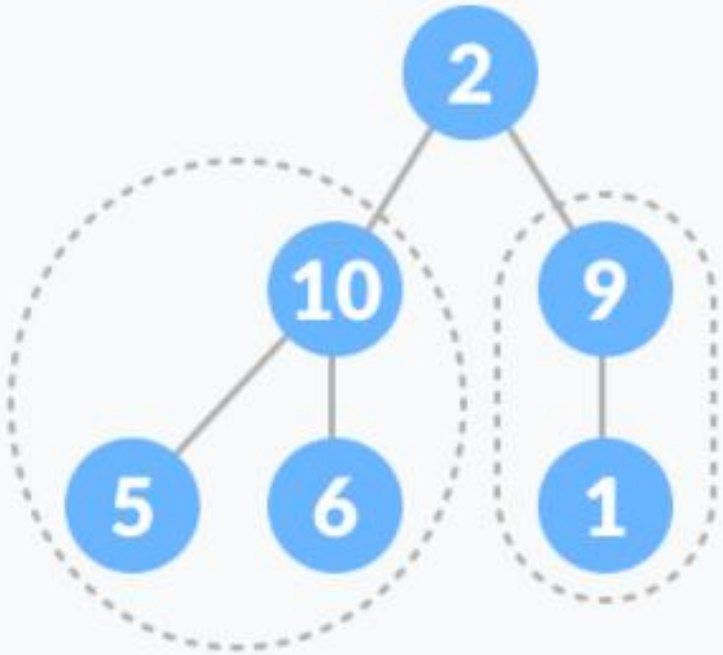


Heapify base cases

The example above shows two scenarios - one in which the root is the largest element and we don't need to do anything. And another in which the root had a larger element as a child and we needed to swap to maintain max-heap property.

If you're worked with recursive algorithms before, you've probably identified that this must be the base case.

Now let's think of another scenario in which there is more than one level.

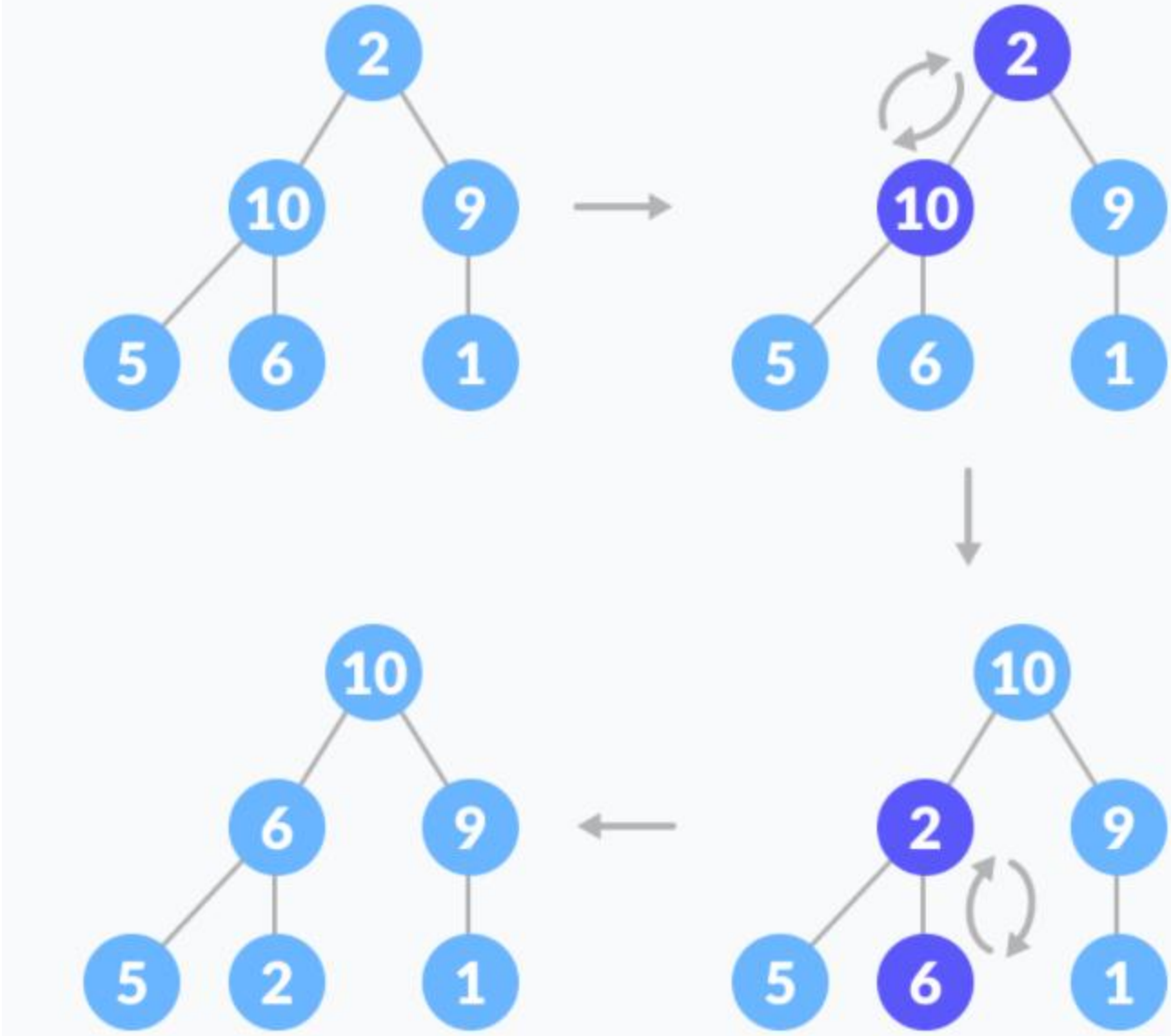


both subtrees of the root are already max-heaps

How to heapify root element when its subtrees are already max heaps

The top element isn't a max-heap but all the sub-trees are max-heaps.

To maintain the max-heap property for the entire tree, we will have to keep pushing 2 downwards until it reaches its correct position.



How to heapify root element when its subtrees are max-heaps

Thus, to maintain the max-heap property in a tree where both sub-trees are max-heaps, we need to run heapify on the root element repeatedly until it is larger than its children or it becomes a leaf node.

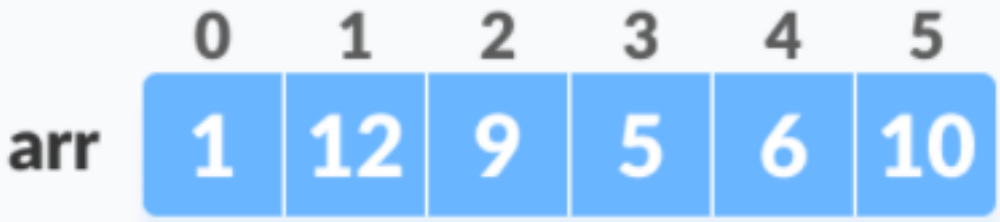
Build max-heap

To build a max-heap from any tree, we can thus start heapifying each sub-tree from the bottom up and end up with a max-heap after the function is applied to all the elements including the root element.

In the case of a complete tree, the first index of a non-leaf node is given by n/2 - 1. All other nodes after that are leaf-nodes and thus don't need to be heapified.

So, we can build a maximum heap as

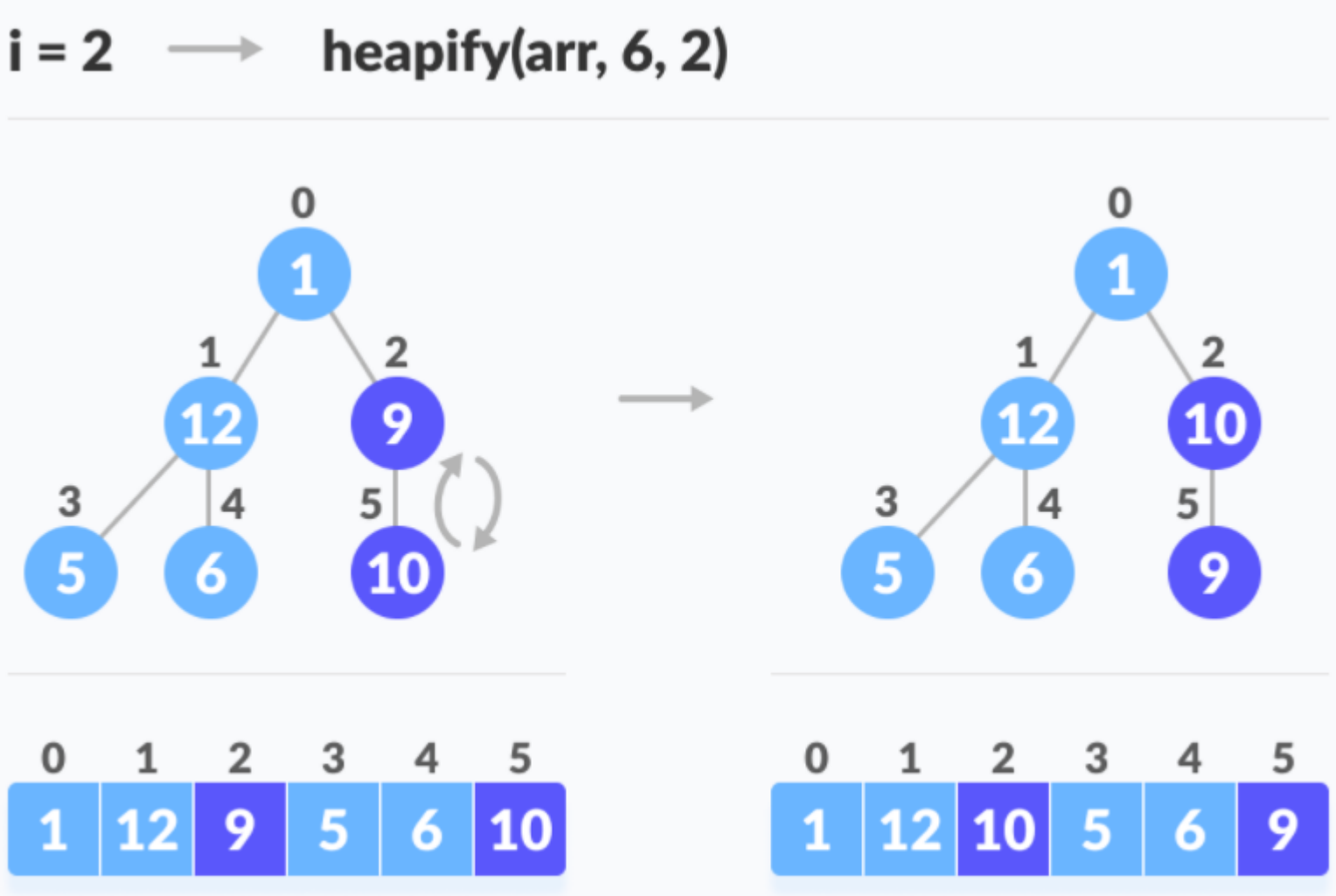
```
// Build heap (rearrange array)
for (int i = n / 2 - 1; i >= 0; i--)
heapify(arr, n, i);
```



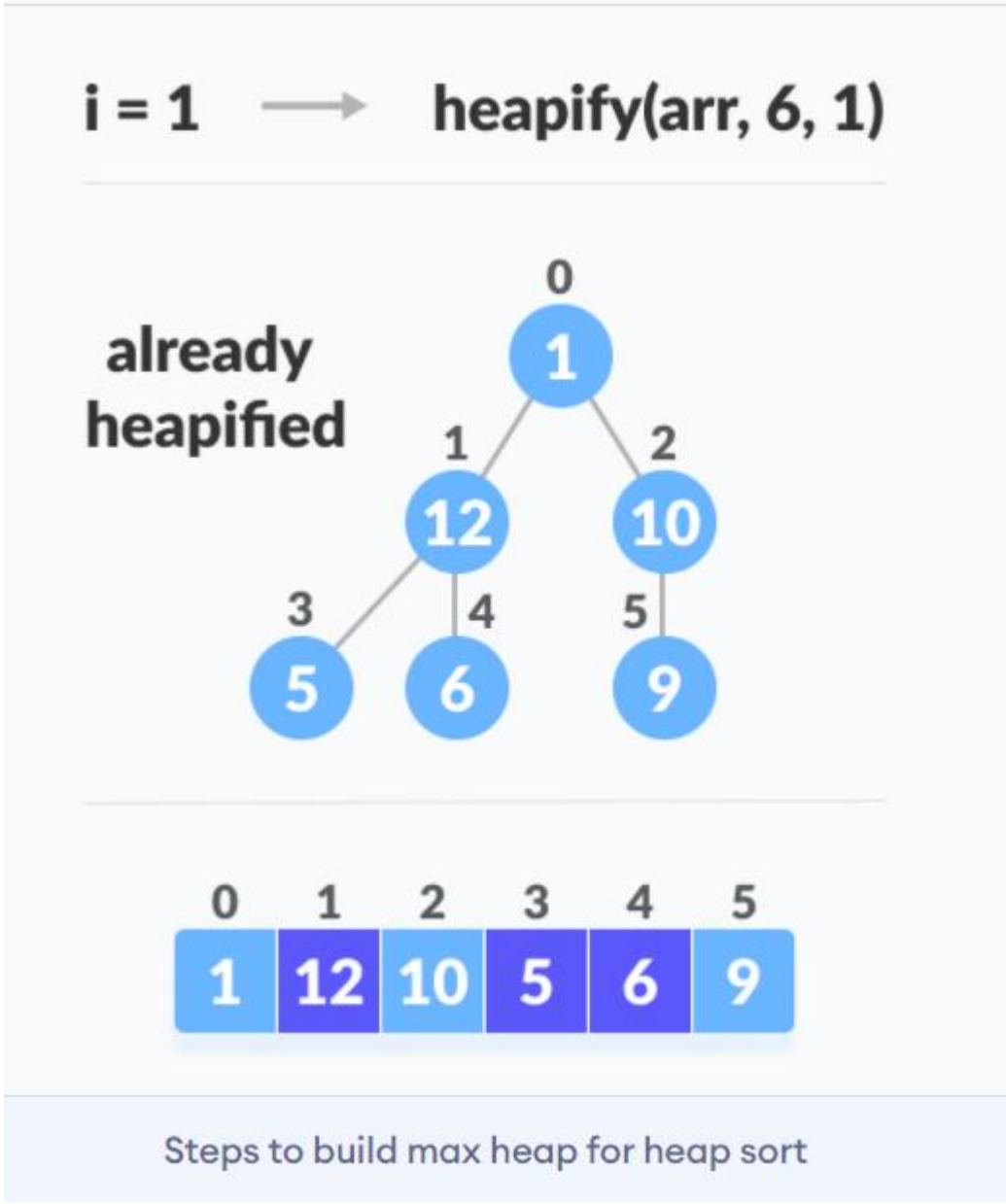
n = 6

i = 6/2 - 1 = 2 # loop runs from 2 to 0

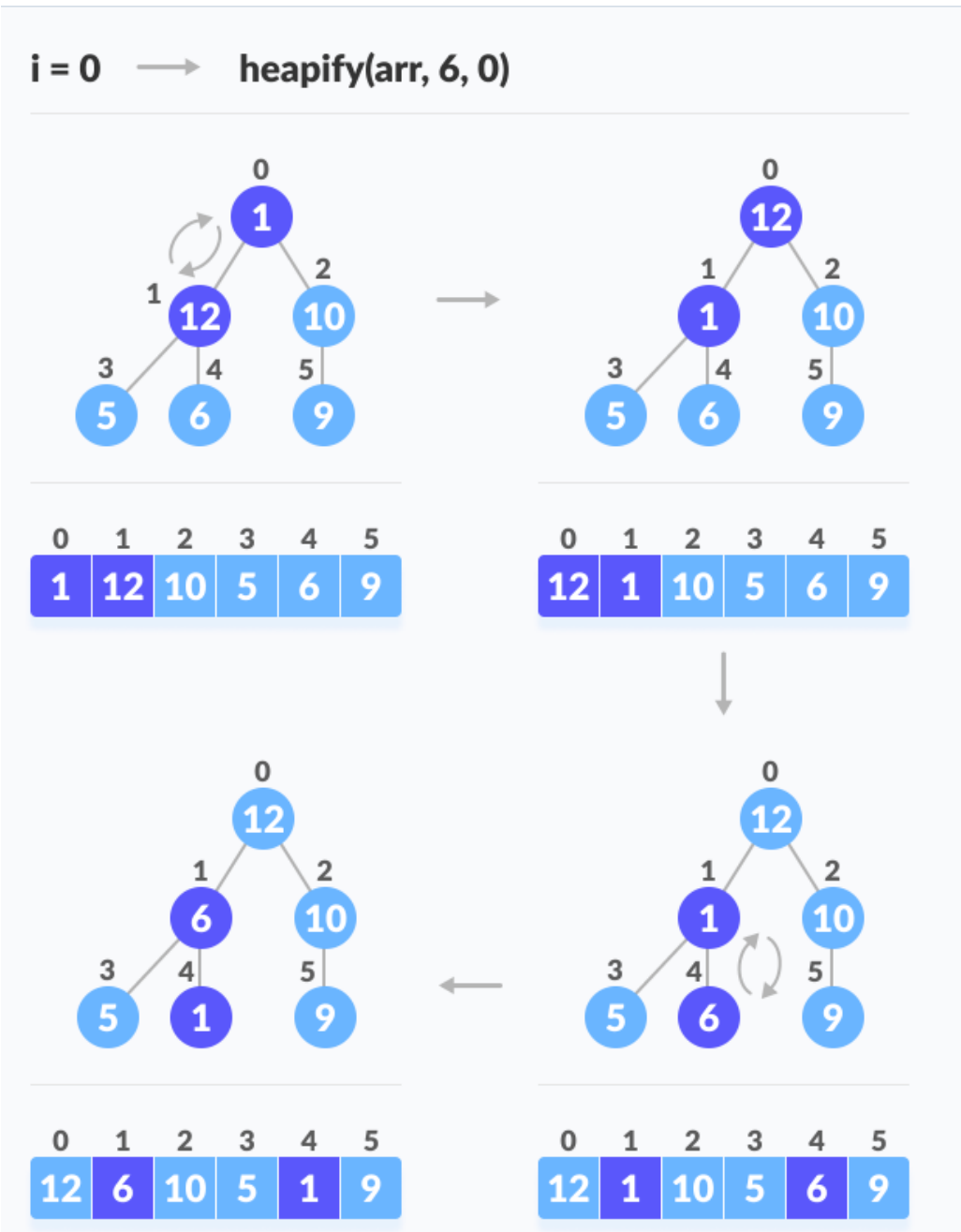
Create array and calculate i



Steps to build max heap for heap sort



Steps to build max heap for heap sort

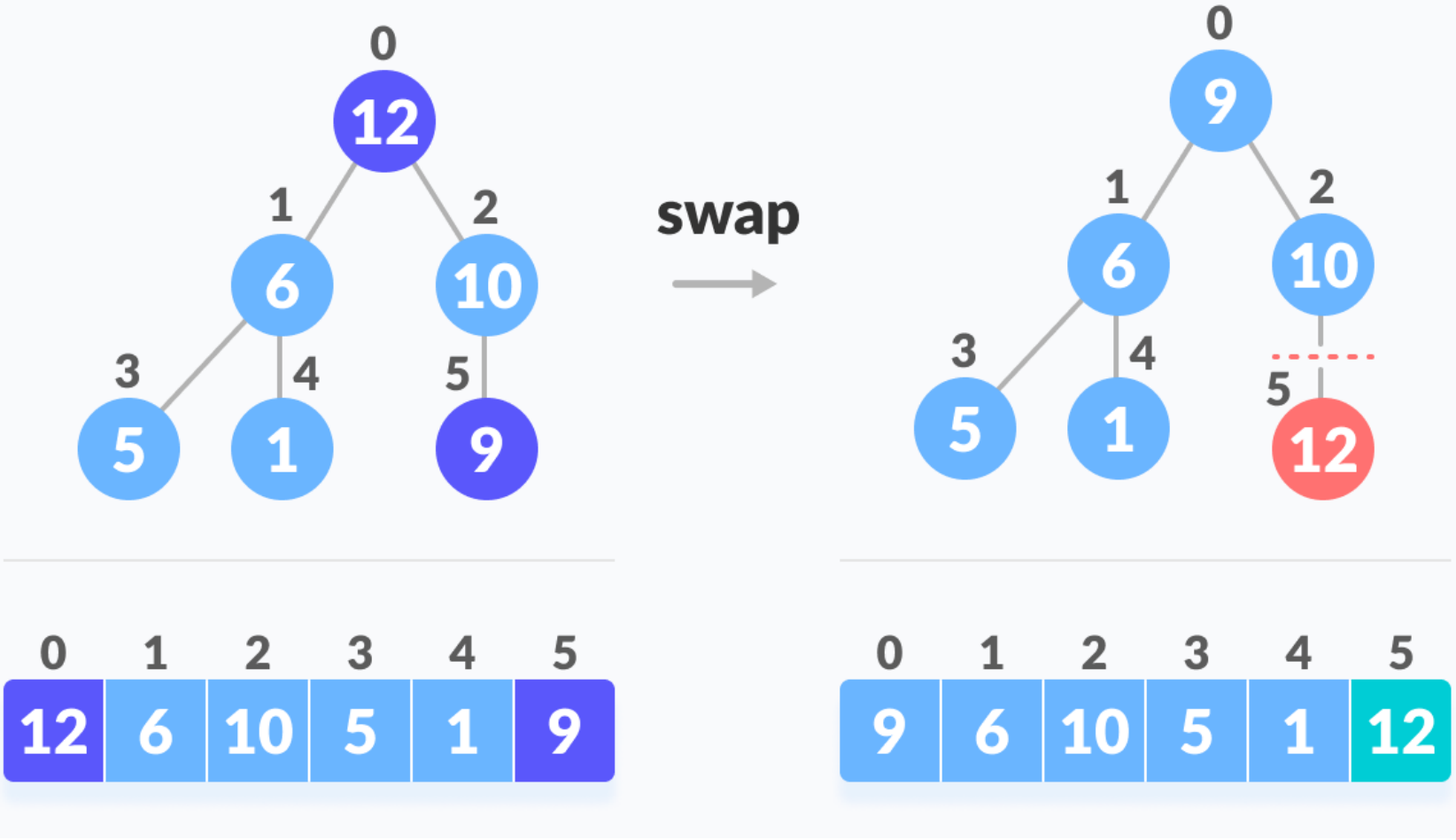


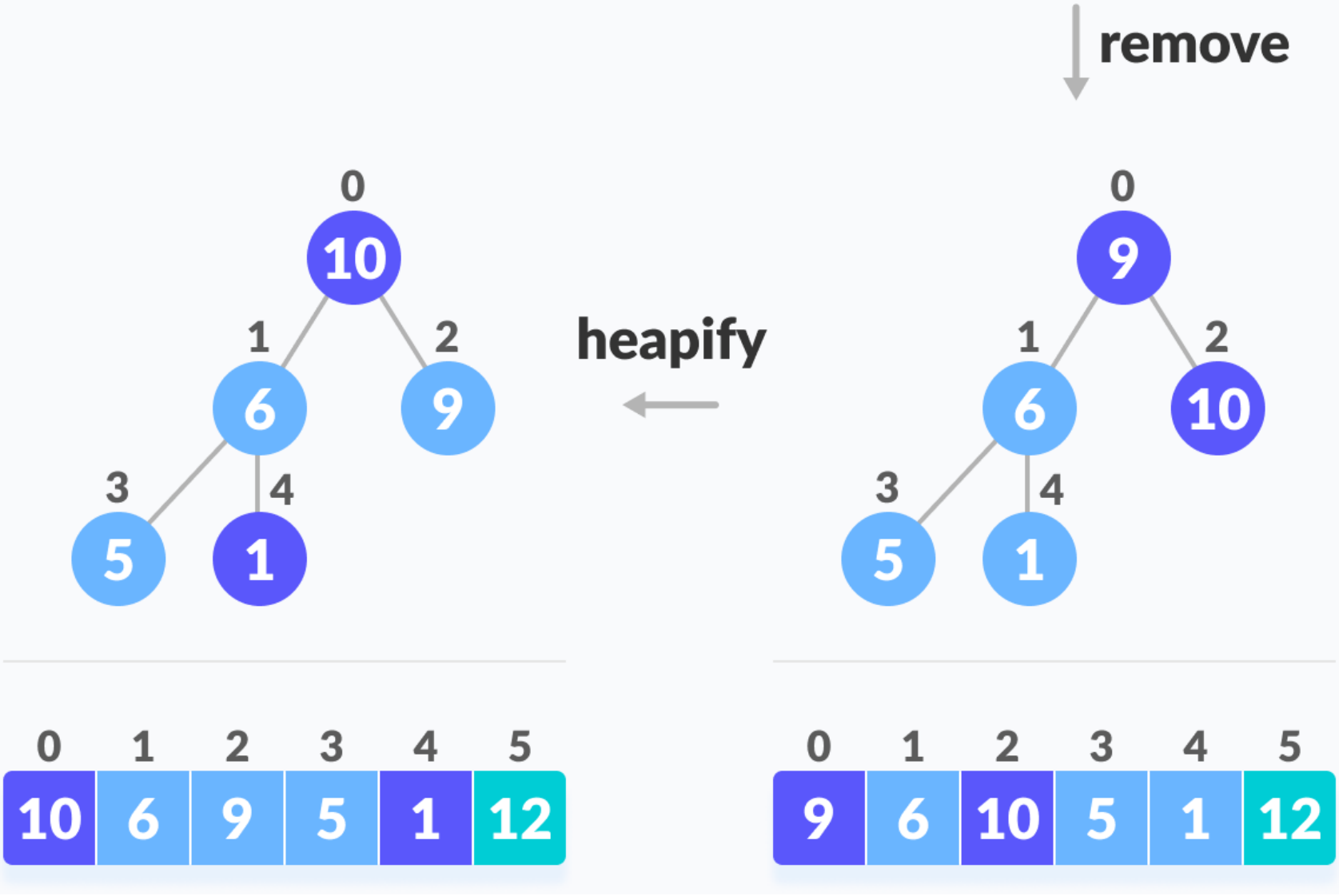
Steps to build max heap for heap sort

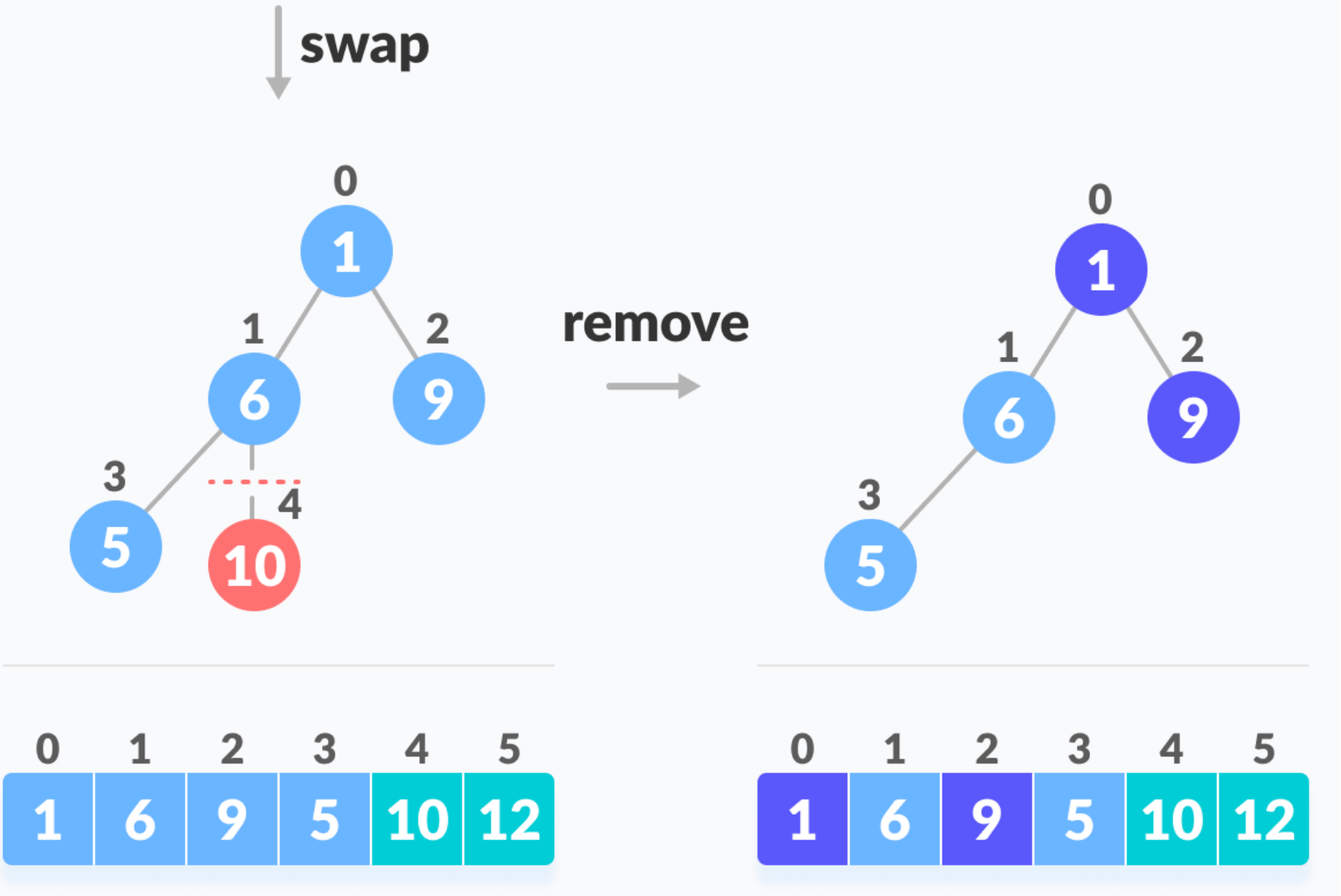
As shown in the above diagram, we start by heapifying the lowest smallest trees and gradually move up until we reach the root element.

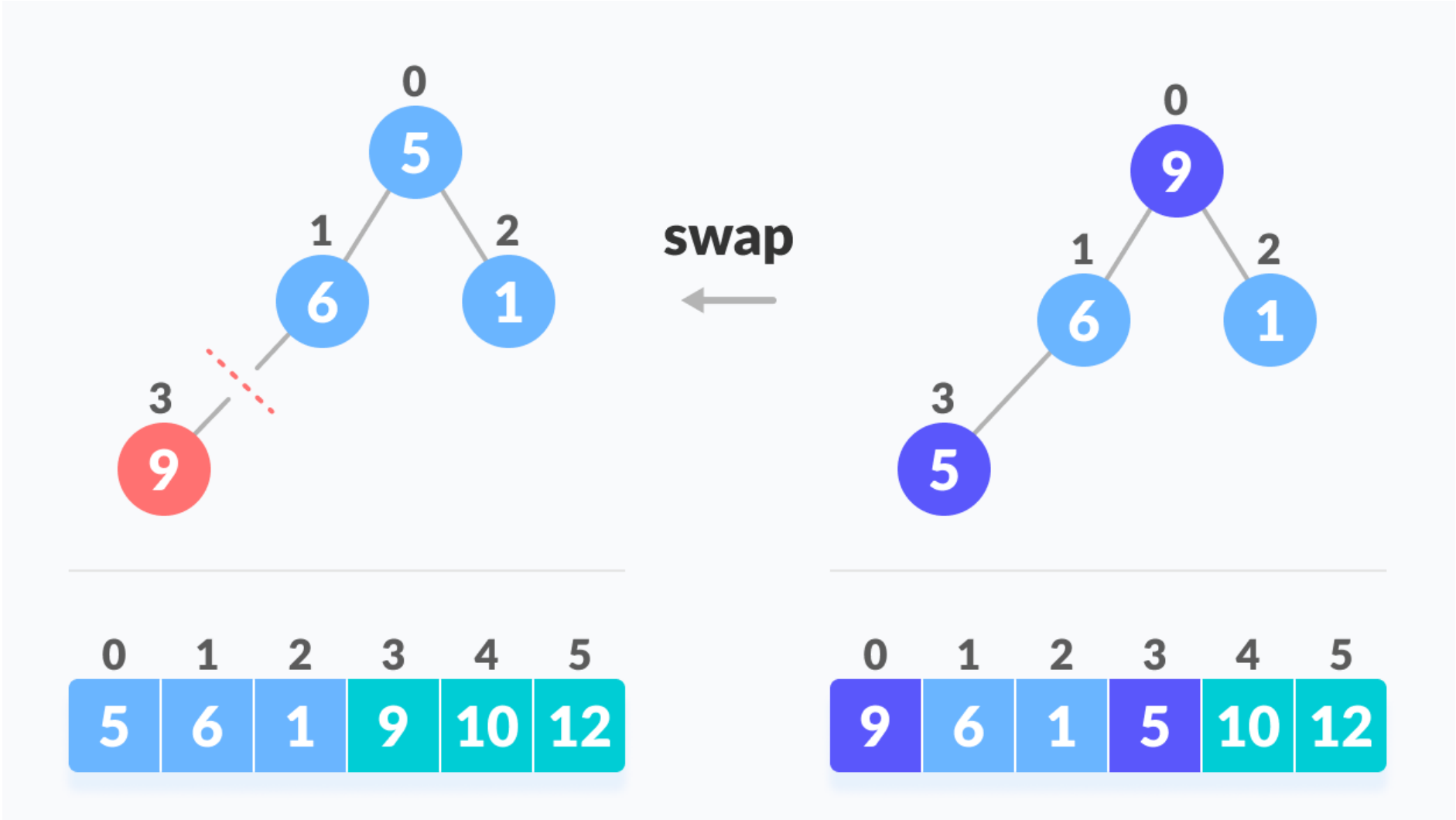
How Heap Sort Works?

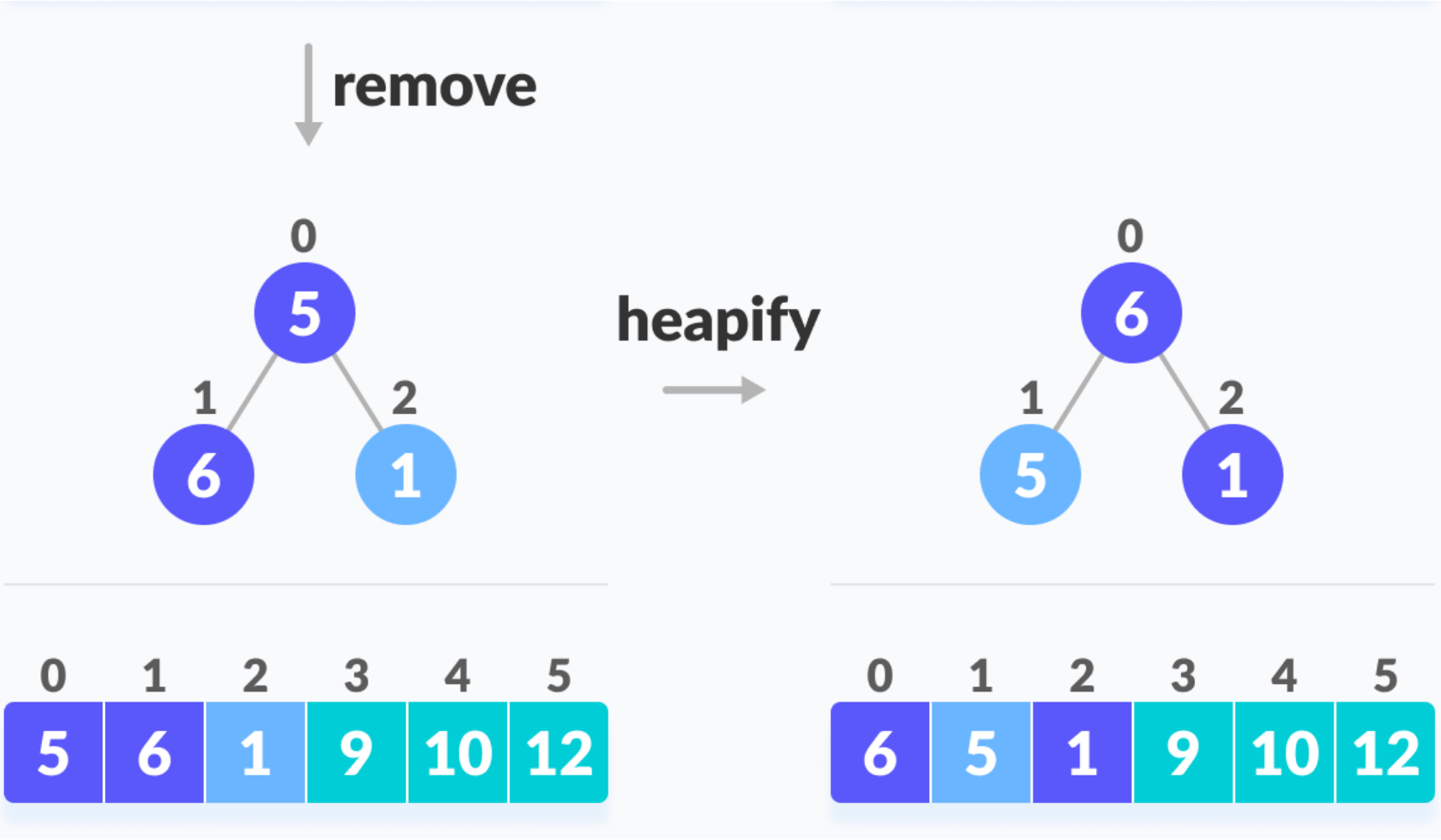
1. Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.
2. **Swap:** Remove the root element and put at the end of the array (nth position) Put the last item of the tree (heap) at the vacant place.
3. **Remove:** Reduce the size of the heap by 1.
4. **Heapify:** Heapify the root element again so that we have the highest element at root.
5. The process is repeated until all the items of the list are sorted.

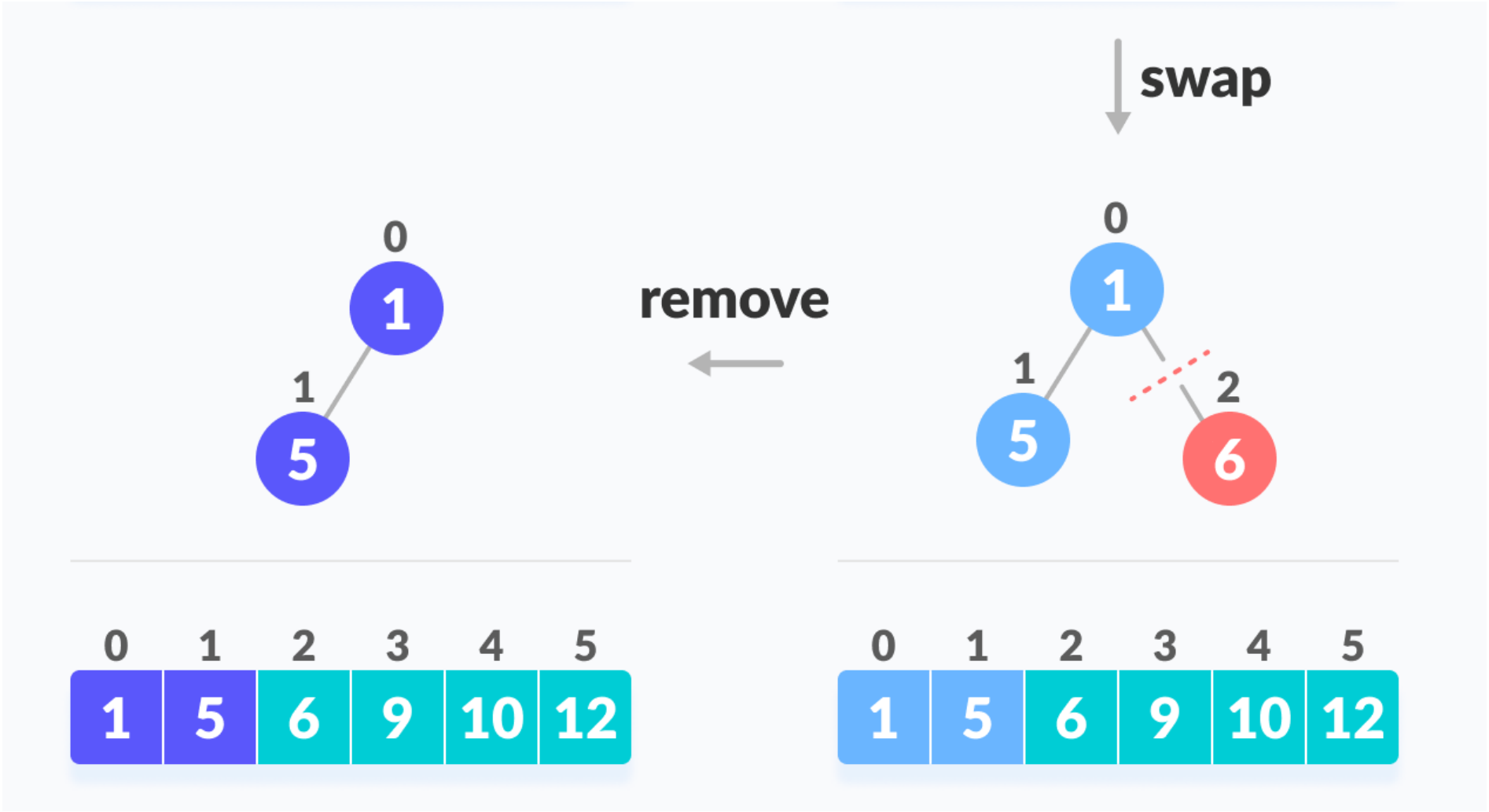

0
12
1
6
2
10
3
5
4
1
5
9
swap
0
9
1
6
2
10
3
5
4
1
5
12
0 1 2 3 4 5
12 6 10 5 1 9
0 1 2 3 4 5
9 6 10 5 1 12

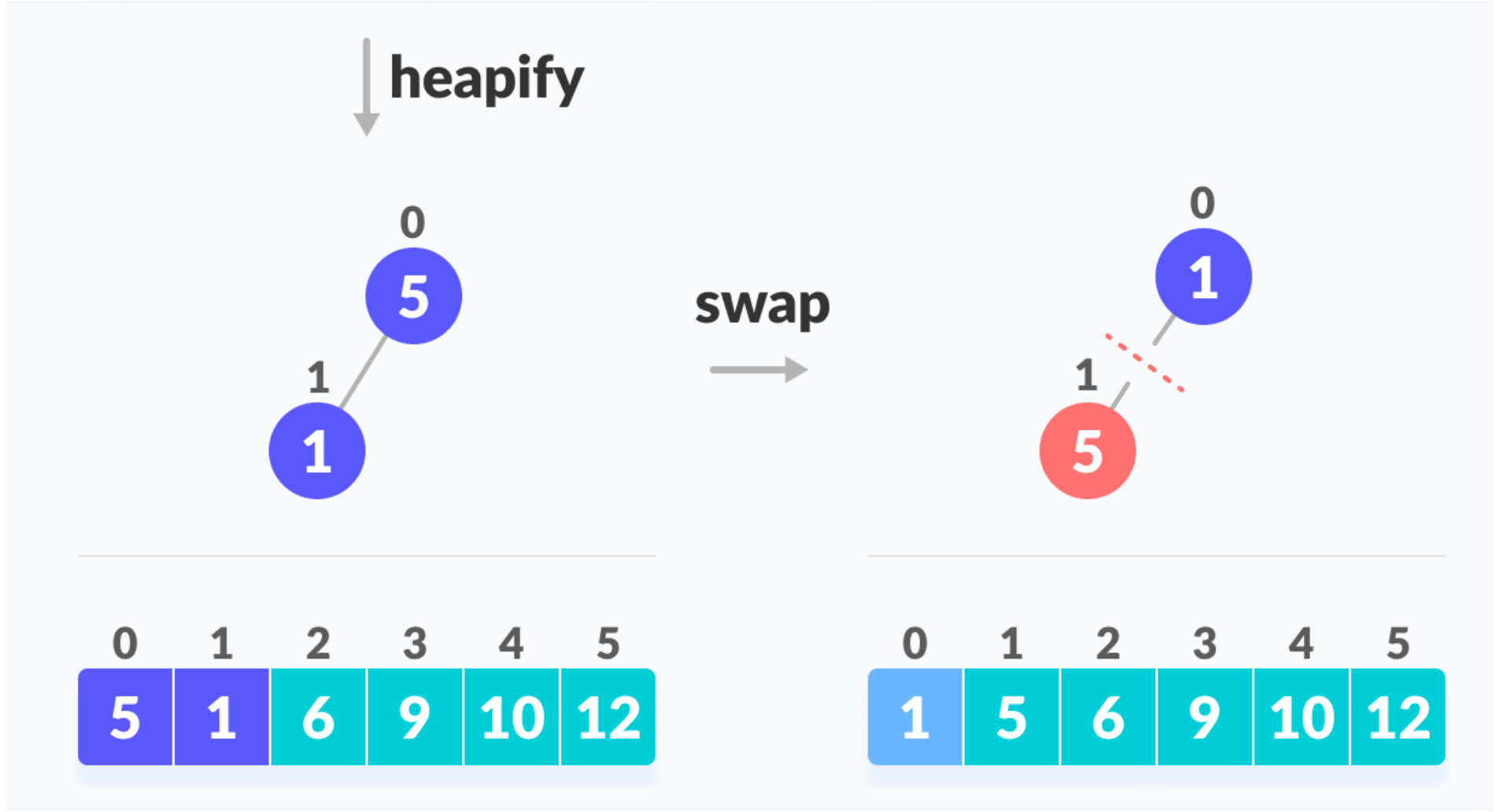

remove
0
10
1
6
2
9
3
5
4
1
heapify
0
9
1
6
2
10
3
5
4
1
0 1 2 3 4 5
10 6 9 5 1 12
0 1 2 3 4 5
9 6 10 5 1 12

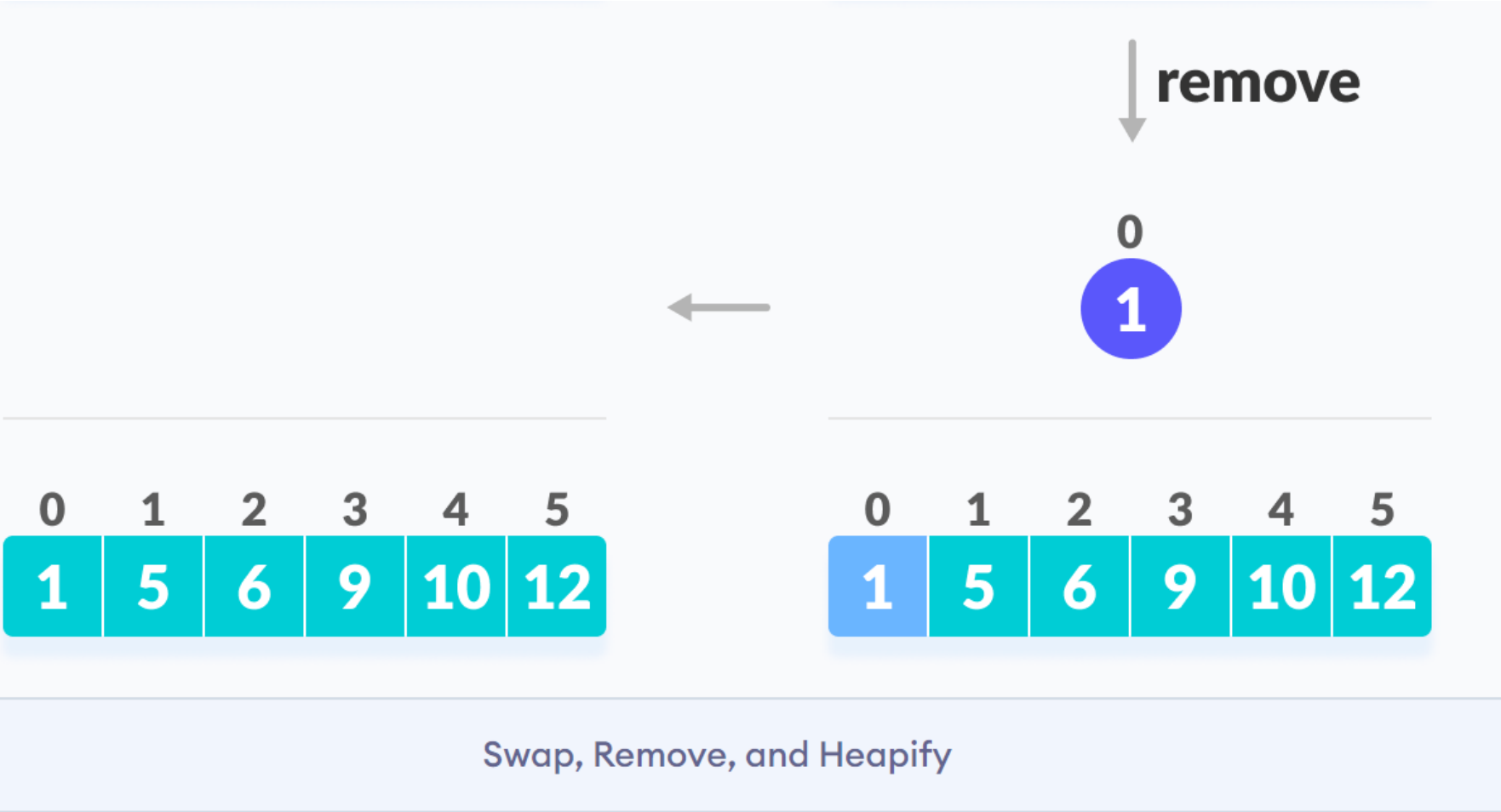

swap
0
1
1
6
2
9
3
5
4
10
remove
0
1
1
6
2
9
3
5
0 1 2 3 4 5
1 6 9 5 10 12
0 1 2 3 4 5
1 6 9 5 10 12


0
5
1
6
2
1
3
9
swap
0
9
1
6
2
1
3
5
0 1 2 3 4 5
5 6 1 9 10 12
0 1 2 3 4 5
9 6 1 5 10 12


remove
0
5
1
6
2
1
heapify
0
6
1
5
2
1
0 1 2 3 4 5
5 6 1 9 10 12
0 1 2 3 4 5
6 5 1 9 10 12


swap
remove
0
1
1
5
0
1
1
5
2
6
0 1 2 3 4 5
1 5 6 9 10 12
0 1 2 3 4 5
1 5 6 9 10 12


heapify
0
5
1
1
swap
0
1
1
5
0 1 2 3 4 5
5 1 6 9 10 12
0 1 2 3 4 5
1 5 6 9 10 12



Swap, Remove, and Heapify

The code below shows the operation.

```
// Heap sort
 for (int i = n - 1; i >= 0; i--)
{
        swap(&arr[0], &arr[i]);
        // Heapify root element to get highest
        //element at root again
        heapify(arr, i, 0);
}
```

Heap Sort Complexity

Heap Sort has O(nlog n) time complexities for all the cases (best case, average case, and worst case).

Let us understand the reason why. The height of a complete binary tree containing n elements is log n

As we have seen earlier, to fully heapify an element whose subtrees are already max-heaps, we need to keep comparing the element with its left and right children and pushing it downwards until it reaches a point where both its children are smaller than it.

In the worst case scenario, we will need to move an element from the root to the leaf node making a multiple of log(n) comparisons and swaps.

During the build_max_heap stage, we do that for n/2 elements so the worst case complexity of the build_heap step is n/2*log n ~ nlog n.

During the sorting step, we exchange the root element with the last element and heapify the root element. For each element, this again takes log n worst time because we might have to bring the element all the way from the root to the leaf. Since we repeat this n times, the heap_sort step is also nlog n.

Also since the build_max_heap and heap_sort steps are executed one after another, the algorithmic complexity is not multiplied and it remains in the order of nlog n.

Also it performs sorting in O(1) space complexity. Compared with Quick Sort, it has a better worst case (O(nlog n)). Quick Sort has complexity O(n^2) for worst case. But in other cases, Quick Sort is fast. Introsort is an alternative to heapsort that combines quicksort and heapsort to retain advantages of both: worst case speed of heapsort and average speed of quicksort.