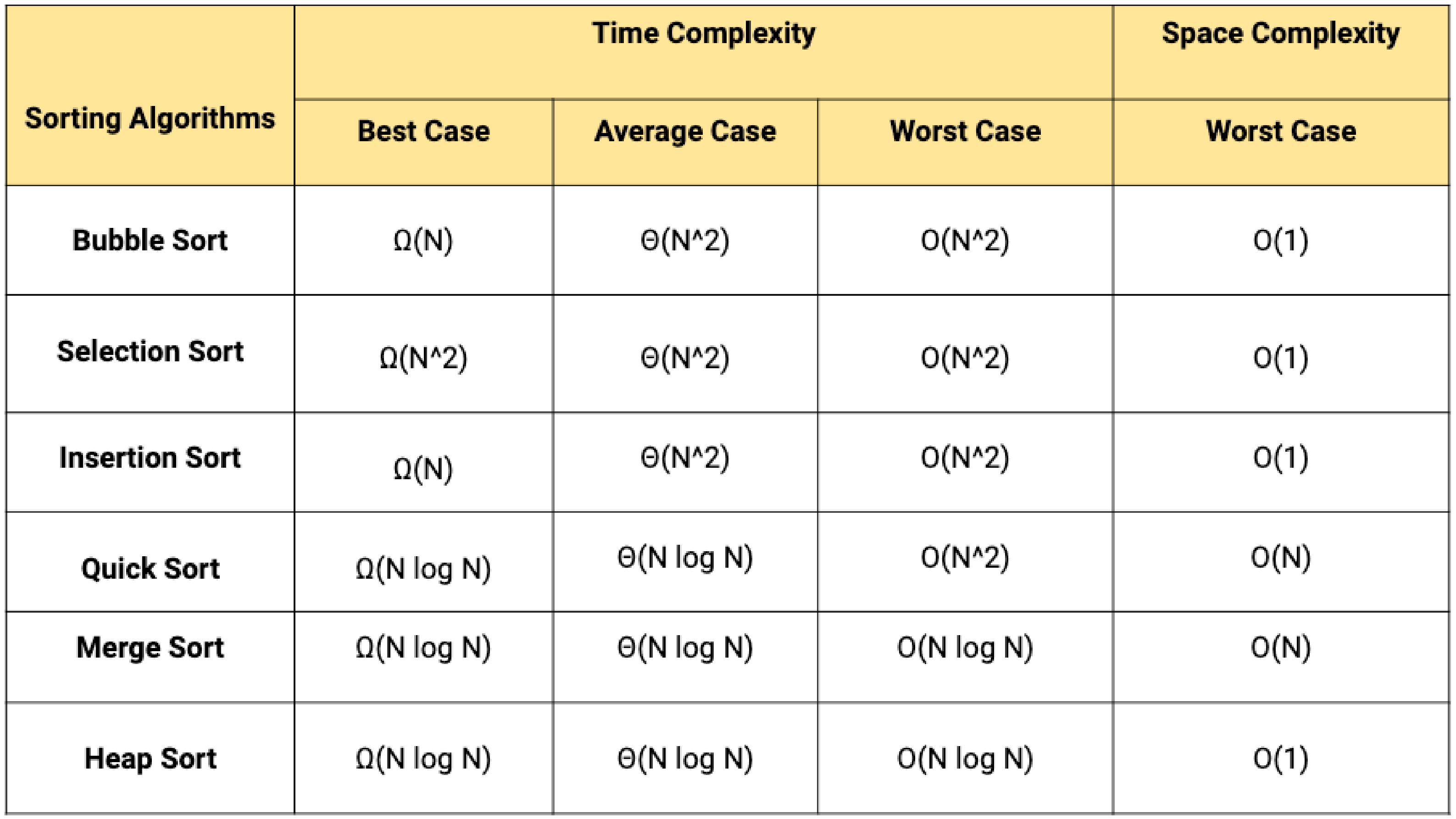
Data Structures and Algorithms

Trong-Hop Do
University of Information Technology, HCM city

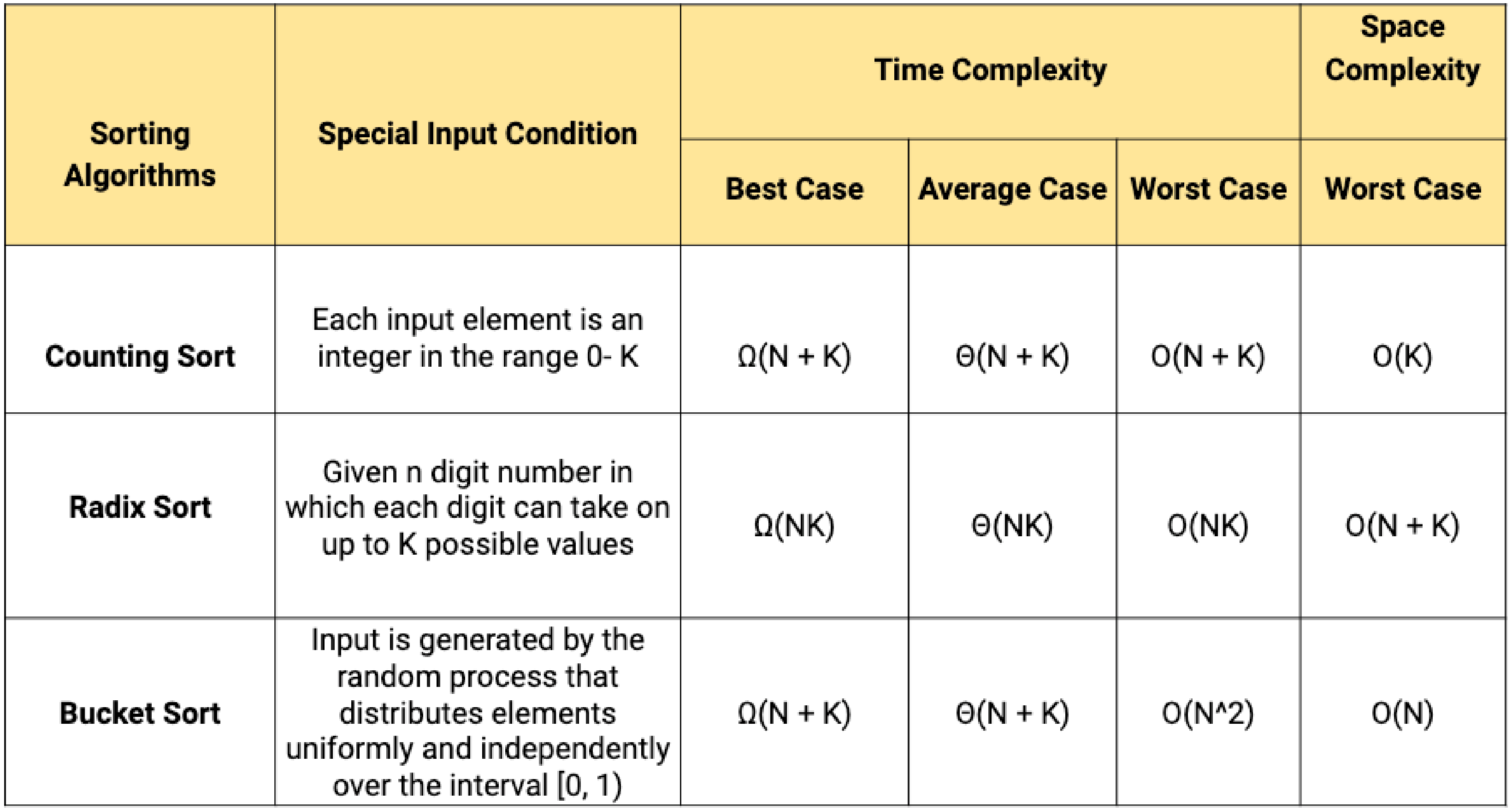
Sorting Algorithms

Comparison based sorting algorithms



Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Selection Sort	Ω(N^2)	Θ(N^2)	O(N^2)	O(1)
Insertion Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Quick Sort	Ω(N log N)	Θ(N log N)	O(N^2)	O(N)
Merge Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(N)
Heap Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(1)

Non-comparison based sorting algorithm



Sorting Algorithms	Special Input Condition	Time Complexity			Space Complexity
		Best Case	Average Case	Worst Case	Worst Case
Counting Sort	Each input element is an integer in the range 0- K	Ω(N + K)	Θ(N + K)	O(N + K)	O(K)
Radix Sort	Given n digit number in which each digit can take on up to K possible values	Ω(NK)	Θ(NK)	O(NK)	O(N + K)
Bucket Sort	Input is generated by the random process that distributes elements uniformly and independently over the interval [0, 1)	Ω(N + K)	Θ(N + K)	O(N^2)	O(N)

In-place and Stable sorting Algorithms

- A sorting algorithm is In-place if the algorithm does not use extra space for manipulating the input but may require a small though nonconstant extra space for its operation. Or we can say, a sorting algorithm sorts in-place if only a constant number of elements of the input array are ever stored outside the array.
- A sorting algorithm is stable if it does not change the order of elements with the same value.

- **Example of stable vs unstable:**

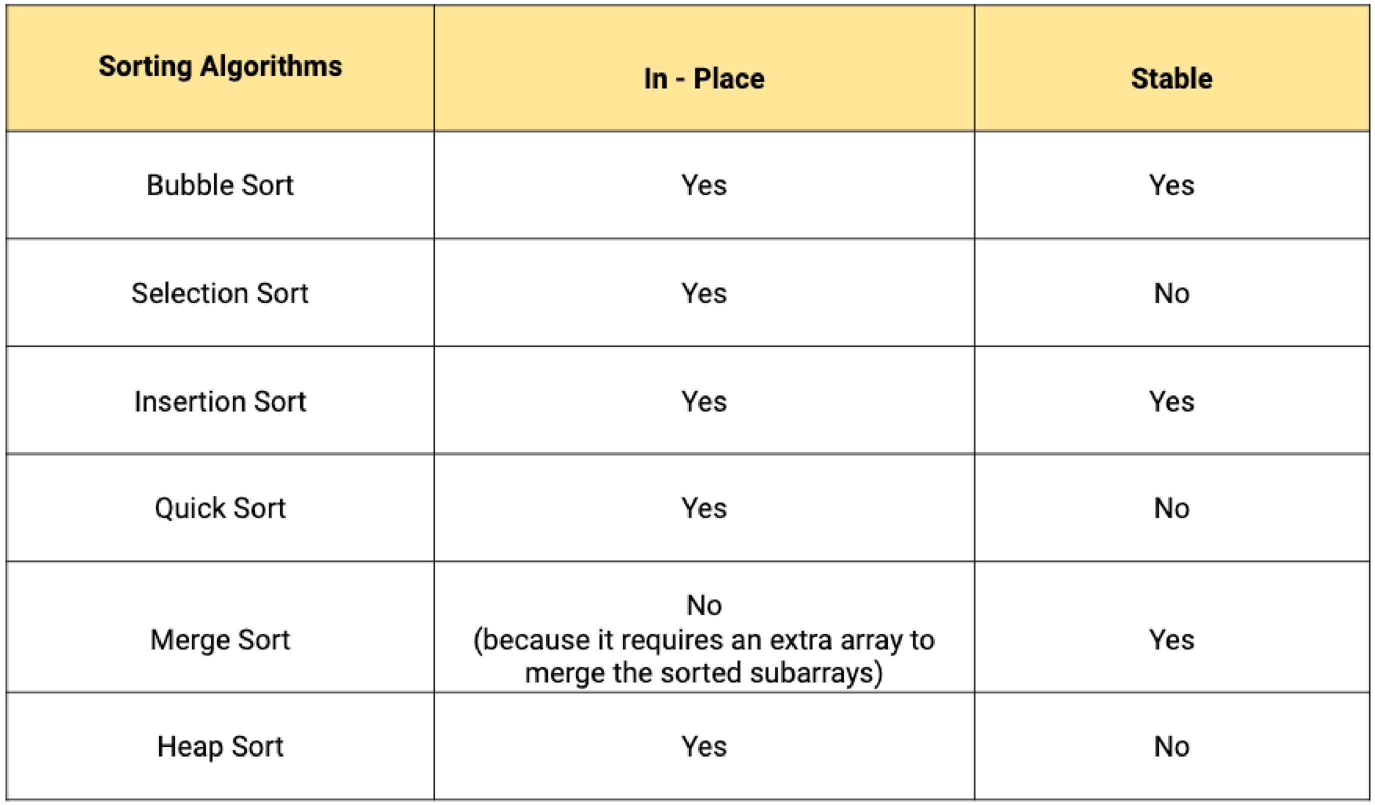
- INPUT: (4,5), (3, 2) (4, 3) (5,4) (6,4)

- OUTPUT1: (3, 2), (4, 5), (4,3), (5,4), (6,4)
- OUTPUT2: (3, 2), (4, 3), (4,5), (5,4), (6,4)

- The sorting algorithm which will produce the first output will be known as stable sorting algorithm because the original order of equal keys are maintained.

In-place and Stable sorting Algorithms

- A sorting algorithm is In-place if the algorithm does not use extra space for manipulating the input but may require a small though nonconstant extra space for its operation. Or we can say, a sorting algorithm sorts in-place if only a constant number of elements of the input array are ever stored outside the array.
- A sorting algorithm is stable if it does not change the order of elements with the same value.



Sorting Algorithms	**In - Place**	**Stable**
Bubble Sort	Yes	Yes
Selection Sort	Yes	No
Insertion Sort	Yes	Yes
Quick Sort	Yes	No
Merge Sort	No (because it requires an extra array to merge the sorted subarrays)	Yes
Heap Sort	Yes	No

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Bubble Sort

First Pass:
(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Bubble Sort

Second Pass:
(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Bubble Sort

Third Pass:
(**1** **2** 4 5 8) –> (**1** **2** 4 5 8)
(1 **2** **4** 5 8) –> (1 **2** **4** 5 8)
(1 2 **4** **5** 8) –> (1 2 **4** **5** 8)
(1 2 4 **5** **8**) –> (1 2 4 **5** **8**)

Bubble Sort

i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i = 1	0	3	1	5	8	2	4	7	**9**
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 2	0	1	3	5	2	4	7	**8**	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 3	0	1	3	2	4	5	**7**		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 4	0	1	2	3	4	**5**			
	1	1	2	3	4				
	2	1	2	3	4				
i = 5	0	1	2	3	**4**				
	1	1	2	3					
i = 6	0	1	2	**3**					
		1	**2**						

Bubble Sort

```
void bubbleSort(int arr[], int n)
{
   int i, j;
   for (i = 0; i < n-1; i++)

       // Last i elements are already in place
       for (j = 0; j < n-i-1; j++)
           if (arr[j] > arr[j+1])
              swap(&arr[j], &arr[j+1]);
}
```

The above function always runs O(n^2) time even if the array is sorted. It can be optimized by stopping the algorithm if inner loop didn’t cause any swap.

Bubble Sort

```
// An optimized version of Bubble Sort
void bubbleSort(int arr[], int n)
{
   int i, j;
   bool swapped;
   for (i = 0; i < n-1; i++)
   {
     swapped = false;
     for (j = 0; j < n-i-1; j++)
     {
        if (arr[j] > arr[j+1])
        {
           swap(&arr[j], &arr[j+1]);
           swapped = true;
        }
     }
     // IF no two elements were swapped by inner loop, then break
     if (swapped == false)   break;
   }
}
```

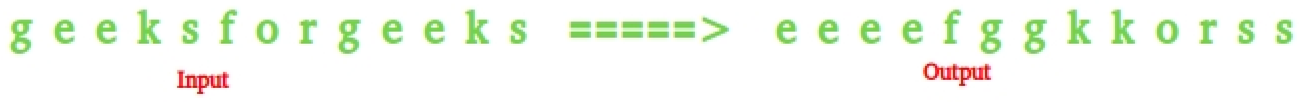
Bubble Sort

- **Worst and Average Case Time Complexity:** $O(n^2)$. Worst case occurs when array is reverse sorted.
- **Best Case Time Complexity:** O(n). Best case occurs when array is already sorted.
- **Auxiliary Space:** O(1)
- **Boundary Cases:** Bubble sort takes minimum time (Order of n) when elements are already sorted.

Sorting algorithms

- A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure.

- **For example**: The below list of characters is sorted in increasing order of their ASCII values. That is, the character with lesser ASCII value will be placed first than the character with higher ASCII value.



Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.

2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Selection Sort

```
arr[] = 64 25 12 22 11

// Find the min in arr[0...4] and place it at beginning of arr[0...4]
64 25 12 22 11  →   11 25 12 22 64

// Find the min in arr[1...4] and place it at beginning of arr[1...4]
11 25 12 22 64  →   11 12 25 22 64

// Find the min in arr[2...4] and place it at beginning of arr[2...4]
11 12 25 22 64  →   11 12 22 25 64

// Find the min in arr[3...4] and place it at beginning of arr[3...4]
11 12 22 25 64  →   11 12 22 25 64
```

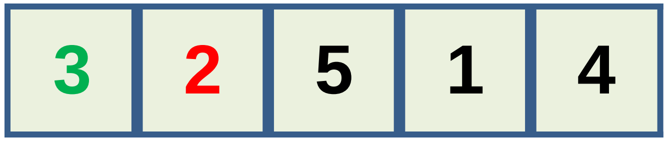
Selection Sort

```
void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}
```

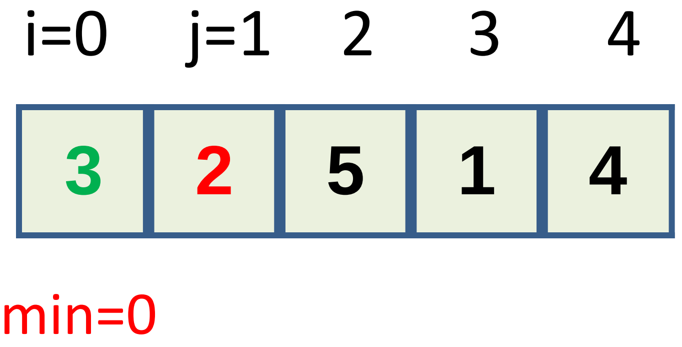
```
void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
          if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}
```

Selection Sort

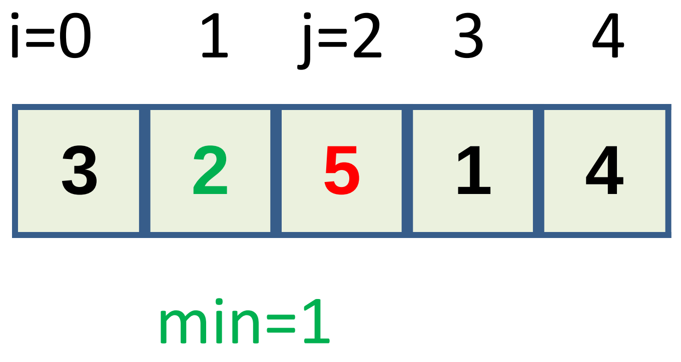
- A={3,2,5,1,4}
- Sorting order is ascending



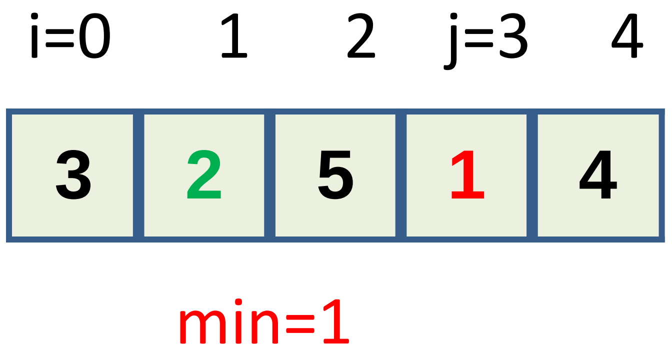
Selection Sort



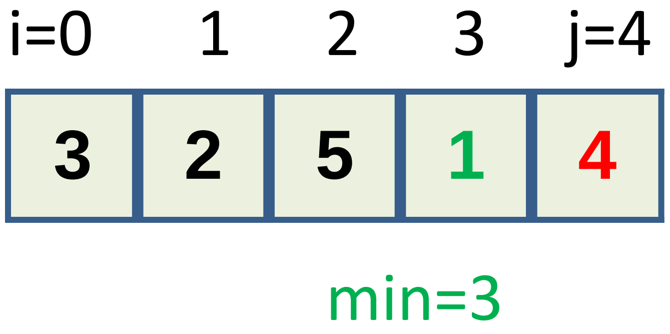
Selection Sort



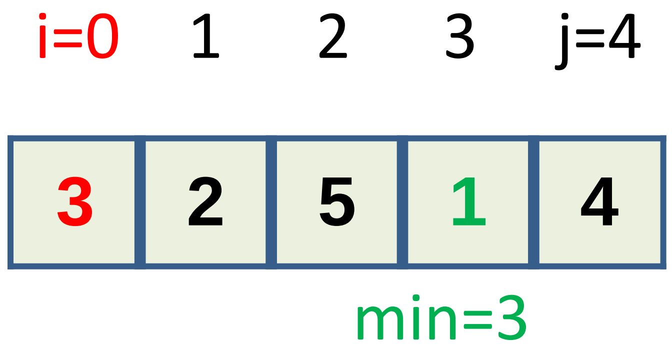
Selection Sort



Selection Sort

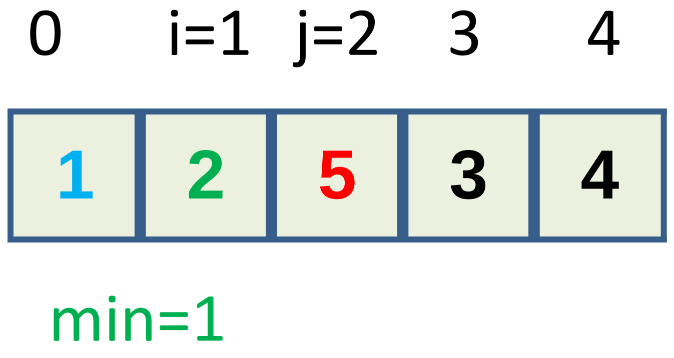


Selection Sort

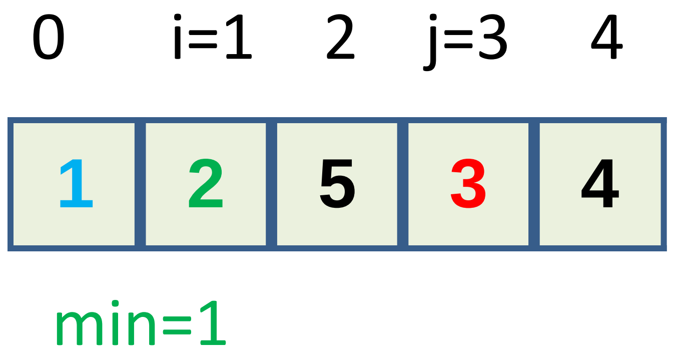


Swap A[i] and A[min]

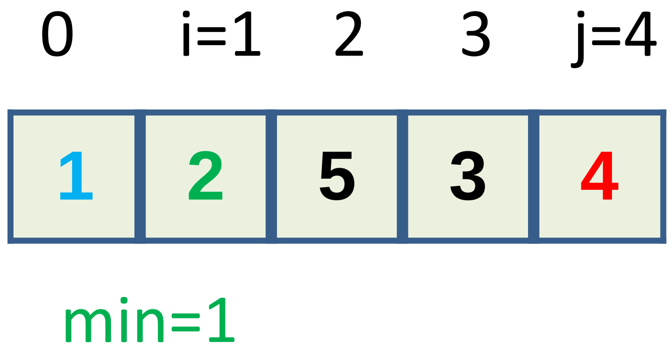
Selection Sort



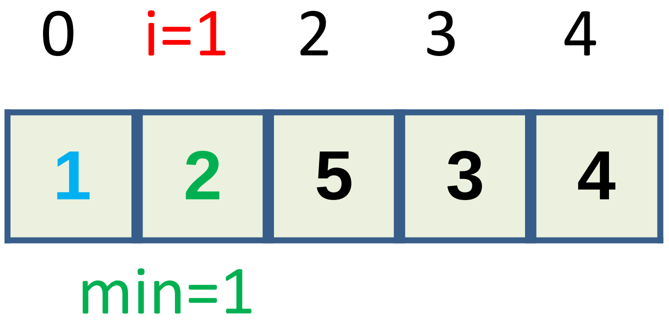
Selection Sort



Selection Sort

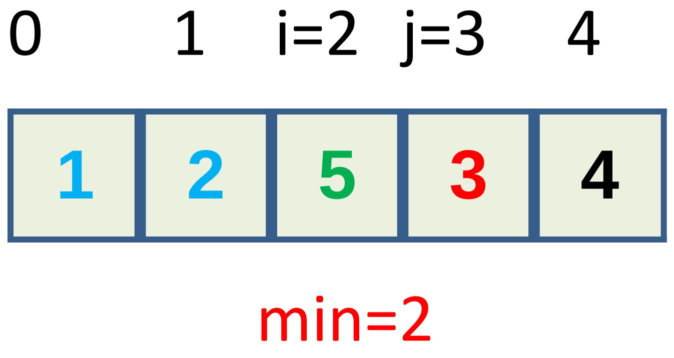


Selection Sort

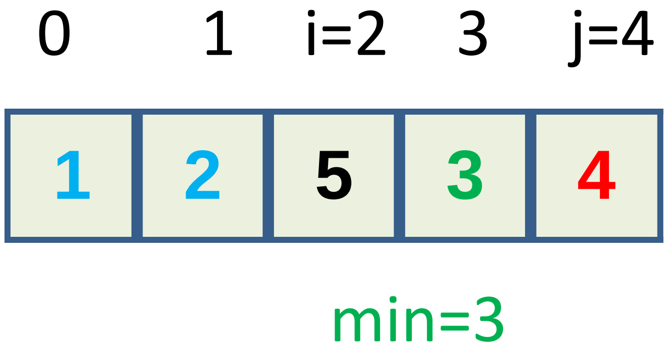


Swap A[i] and A[min]

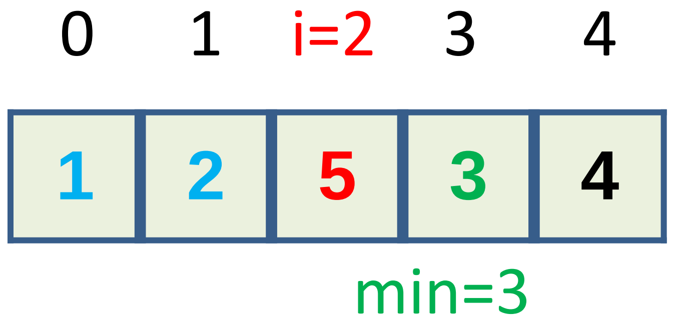
Selection Sort



Selection Sort

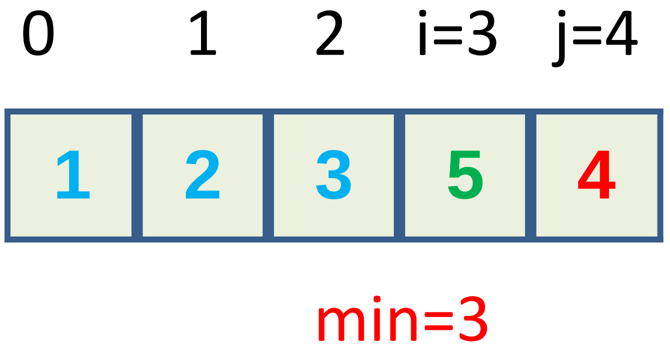


Selection Sort

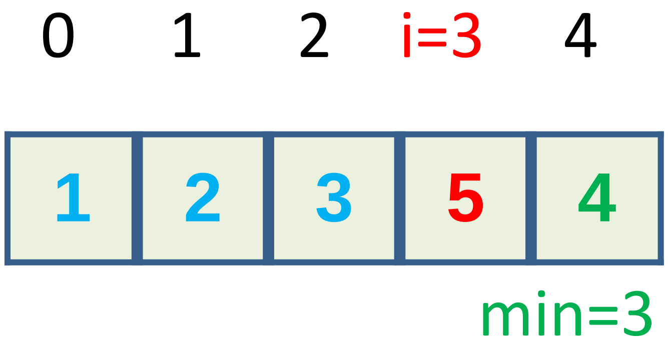


Swap A[i] and A[min]

Selection Sort



Selection Sort



Swap A[i] and A[min]

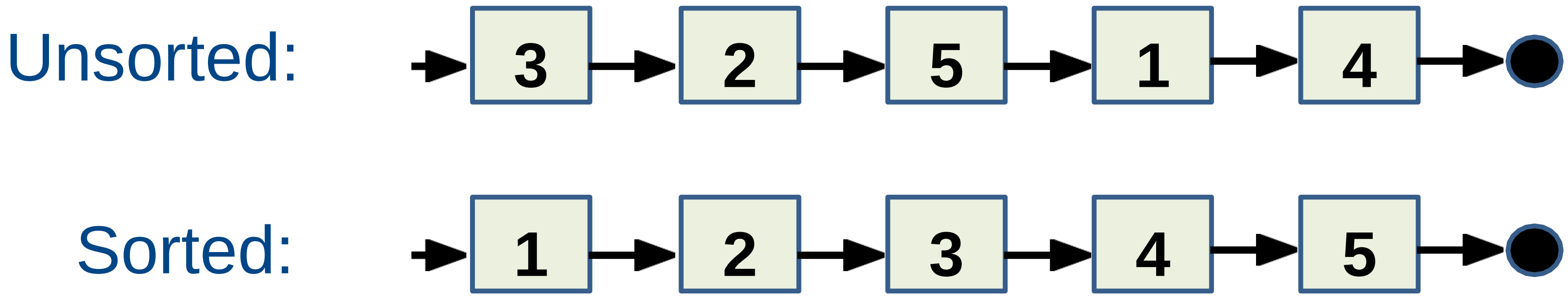
Selection Sort

0	1	2	3	4
1	**2**	**3**	**4**	**5**

End

Selection Sort

Selection sort for linked list



Selection Sort

```
list = 64 25 12 22 11

// Find the min in list(0...4) and place it at beginning
11 25 12 22 64

// Find the min in list(1...4) and place it at beginning of
list(1...4)
11 12 25 22 64

// Find the min in list(2...4) and place it at beginning of
list(2...4)
11 12 22 25 64

// Find the min in list(3...4) and place it at beginning of
list(3...4)
11 12 22 25 64
```

Selection Sort

- The swapping required can be done in two ways:
  - By swapping the data parts of the nodes.
  - By swapping the complete nodes.
- Second implementation is generally used when elements of the list are some kind of records because in such a case data swapping becomes tedious and expensive due to the presence of a large number of data elements.

Selection Sort

```
//Implementation Method 1
void selectionSort(List A) {
     Node *min, *i, *j;
     i = A.pHead;
     while (i) {
          min = i; j = i->pNext;
          while (j) {
               if (j->info < min->info) min = j;  j = j->pNext;
          }
     swap(i->info, min->info);
     i = i->pNext;
     }
}
```

Selection Sort

Method 2:

Data swapping is no doubt easier to implement and understand, but in some cases(as one mentioned above), it isn’t desirable. While doing swapping of the next parts of two nodes, four cases are needed to be taken into consideration :

- Nodes are adjacent and the first node is the starting node.
- Nodes are adjacent and the first node isn’t the starting node.
- Nodes aren’t adjacent and the first node is the starting node.
- Nodes aren’t adjacent and the first node isn’t the starting node.

Selection Sort

```
//Implementation Method 2
void selectionSort(List &A) {
      Node *qmin, *i, *j, *h; h = NULL; i = A.pHead;
      while (i->pNext) {
            qmin = h; j = i;
            while (j->pNext) {
                  if (j->pNext->info < i->info) {qmin = j; i = qmin->pNext;}
                  j = j->pNext;
            }
            addAfter(A, removeAfter(A,qmin), h);
            if (!h) h = A.pHead;
            else h = h->pNext;
            i = h->pNext;
      }
}
```

Selection Sort

- Time Complexity: O(n2) as there are two nested loops.
- Auxiliary Space: O(1)
- The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

Selection Sort

- Suppose the array is
  5 2 3 8 4 5 6
  Let's distinguish the two 5's as 5(a) and 5(b) .
- So our array is:
  5(a) 3 4 5(b) 2 6 8
- After iteration 1:
  2 will be swapped with the element in 1st position:
- So our array becomes:
  2 3 4 5(b) 5(a) 6 8
- Since now our array is in sorted order and we clearly see that 5(a) comes before 5(b) in initial array but not in the sorted array.

- Therefore, selection sort is unstable.

Insertion Sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Algorithm

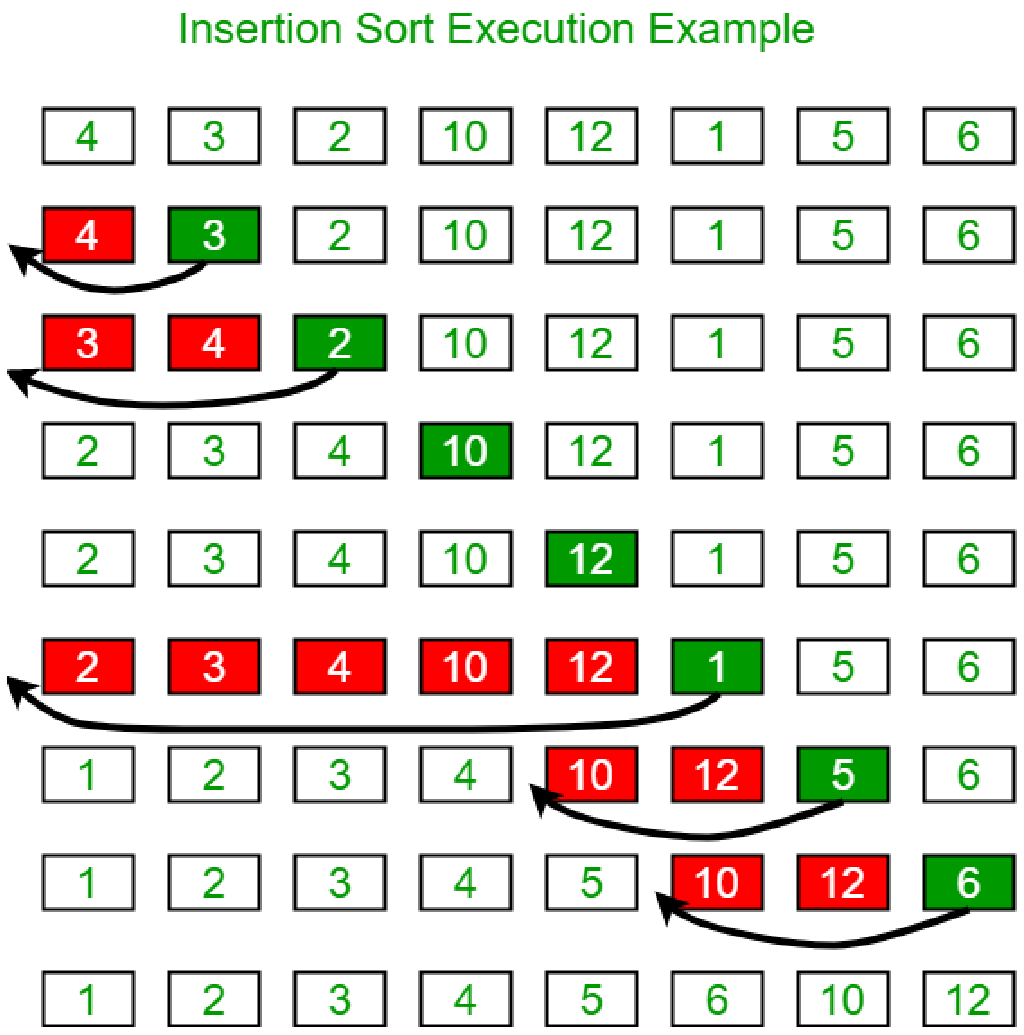
To sort an array of size n in ascending order:

1: Iterate from arr[1] to arr[n] over the array.

2: Compare the current element (key) to its predecessor.

3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Insertion Sort



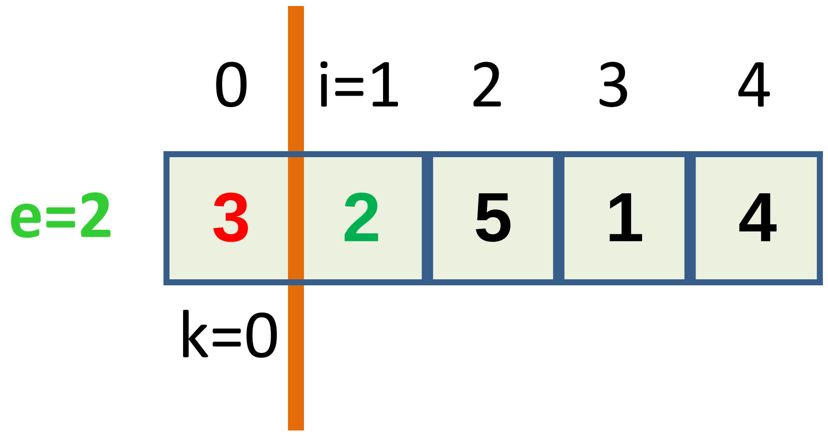
Insertion Sort

```
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

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

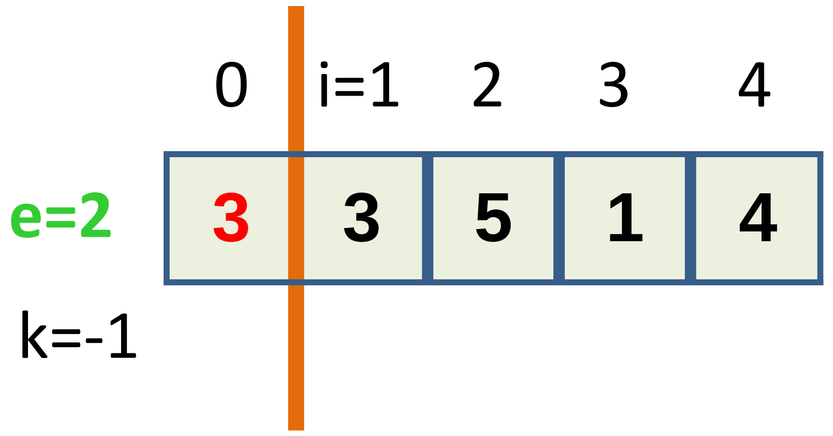
Insertion Sort

A={3,2,5,1,4}

Sorting order is ascending

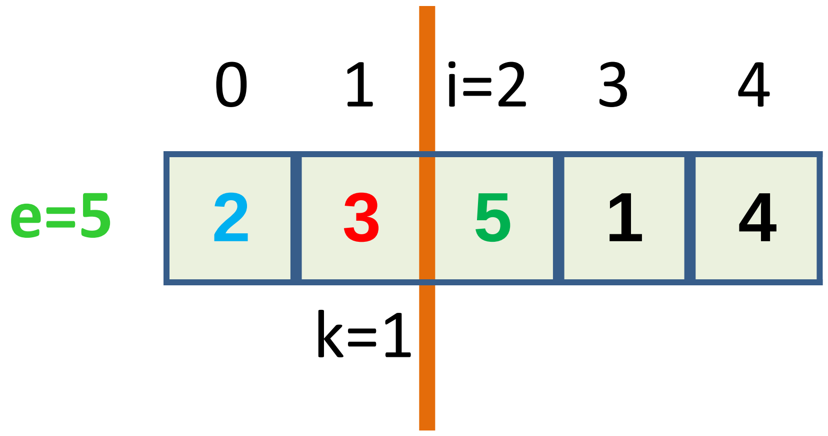


Insertion Sort



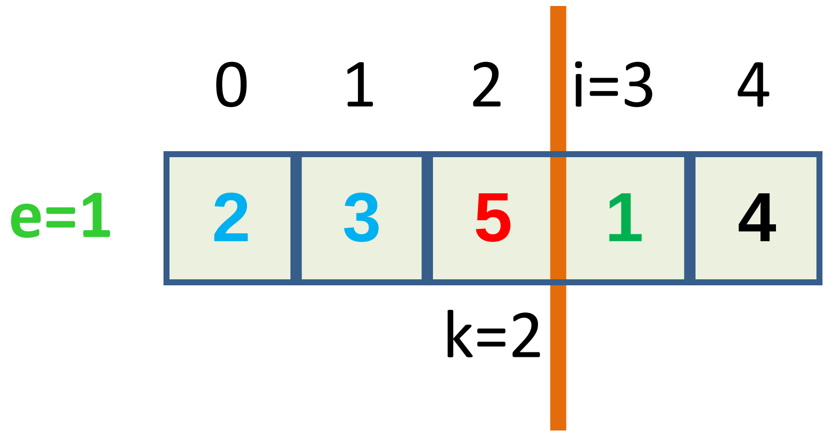
Đưa phần tử e vào vị trí k+1

Insertion Sort

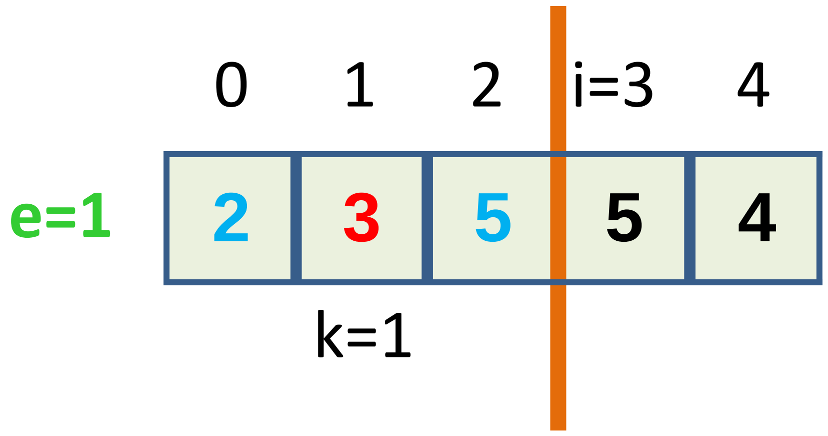


Đưa phần tử e vào vị trí k+1

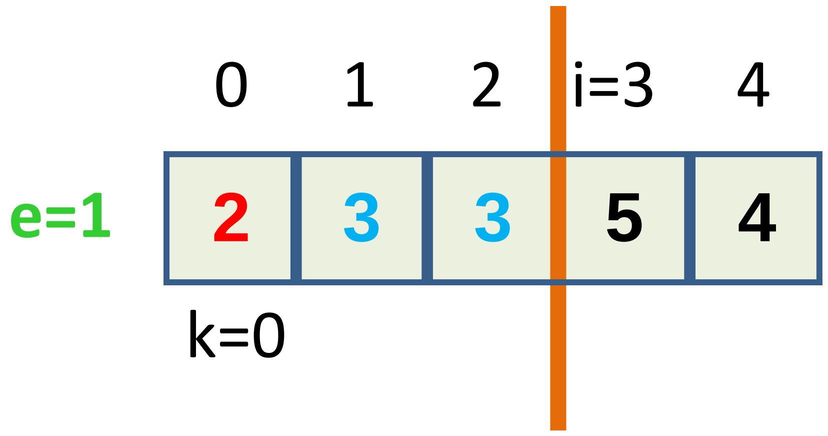
Insertion Sort



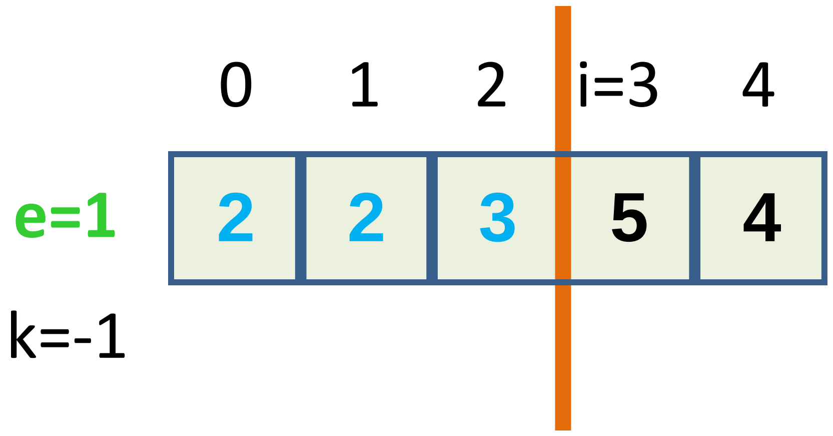
Insertion Sort



Insertion Sort

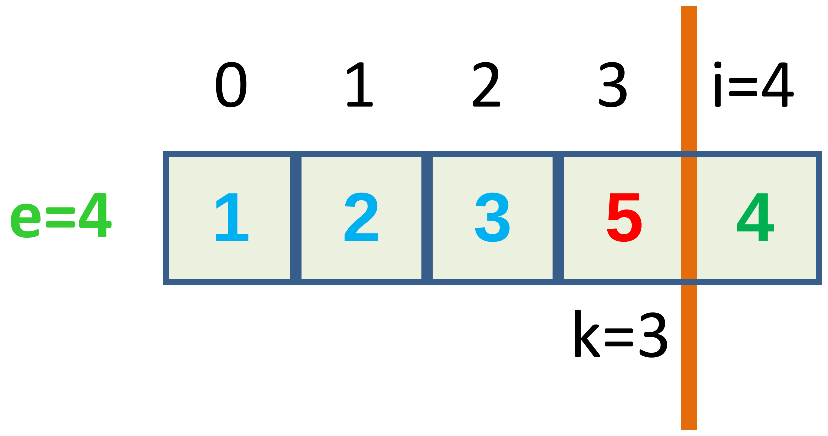


Insertion Sort

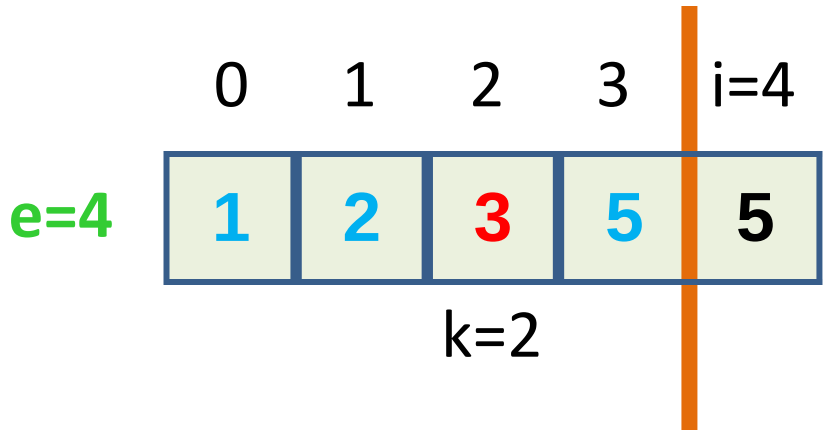


Đưa phần tử e vào vị trí k+1

Insertion Sort

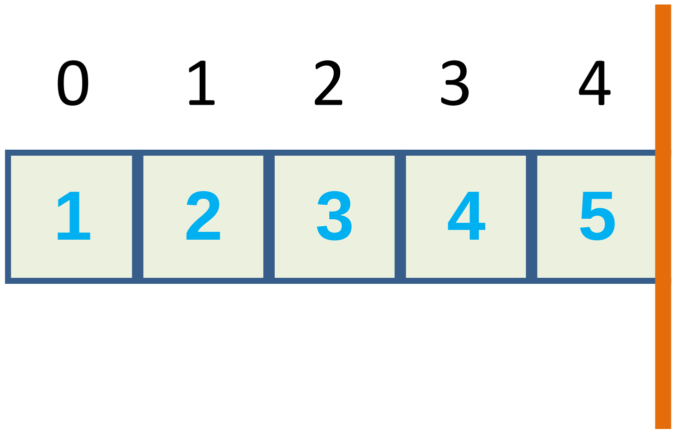


Insertion Sort



Đưa phần tử e vào vị trí k+1

Insertion Sort



Insertion Sort

- Insertion Sort for Singly Linked List

1) Create an empty sorted (or result) list

2) Traverse the given list, do following for every node.

… Insert current node in sorted way in sorted or result list.

3) Change head of given linked list to head of sorted (or result) list.

Insertion Sort

```
void insertionSort(List &A) {
     Node *i = A.pHead, *k, *e;
     while (i->pNext) {
          q = NULL; k = A.pHead;  e=removeAfter(A,i);
          while (k != i->pNext) {
               if (!(k->info < e->info)) break;
               q = k; k = q->pNext;
          }
          addAfter(A, e, q);
          if (i->pNext == e) i = i->pNext;
     }
}
```

Insertion Sort

- **Time Complexity:** O(n*2)
- **Auxiliary Space:** O(1)
- **Boundary Cases**: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

Binary Insertion Sort

- We can use binary search to reduce the number of comparisons in normal insertion sort. Binary Insertion Sort uses binary search to find the proper location to insert the selected item at each iteration.
- In normal insertion sort, it takes O(n) comparisons (at nth iteration) in the worst case. We can reduce it to O(log n) by using binary search.

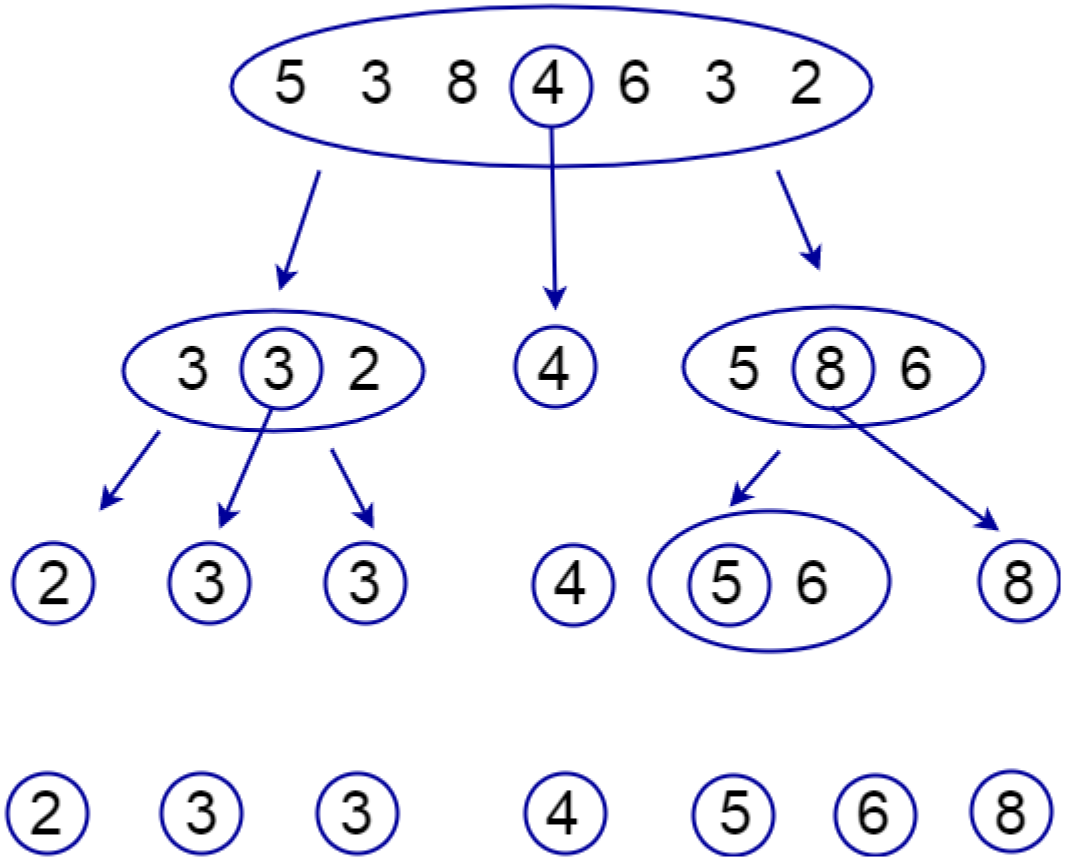
Binary Insertion Sort

```
int binarySearch(int a[], int item, int low, int high)
{
    if (high <= low)
        return (item > a[low]) ? (low + 1) : low;

    int mid = (low + high) / 2;

    if (item == a[mid])
        return mid + 1;

    if (item > a[mid])
        return binarySearch(a, item, mid + 1, high);
    return binarySearch(a, item, low, mid - 1);
}
```

Binary Insertion Sort

The number of comparisons drops to $O(n \log n)$, but the number of shifts in the array stays $O(n^2)$, which now dominates the run time. So the total algorithm stays $O(n^2)$.

Quick sort



Quick sort

Principle of quick sort:

- If the array A was sorted, given any A_i:
  - $A_j < A_i \ \forall \ j < i$
  - $A_i > A_i \ \forall \ j > i$
  - $A_0, \ldots, A_{i-1}$ and $A_{i+1}, \ldots, A_{n-1}$ are sorted

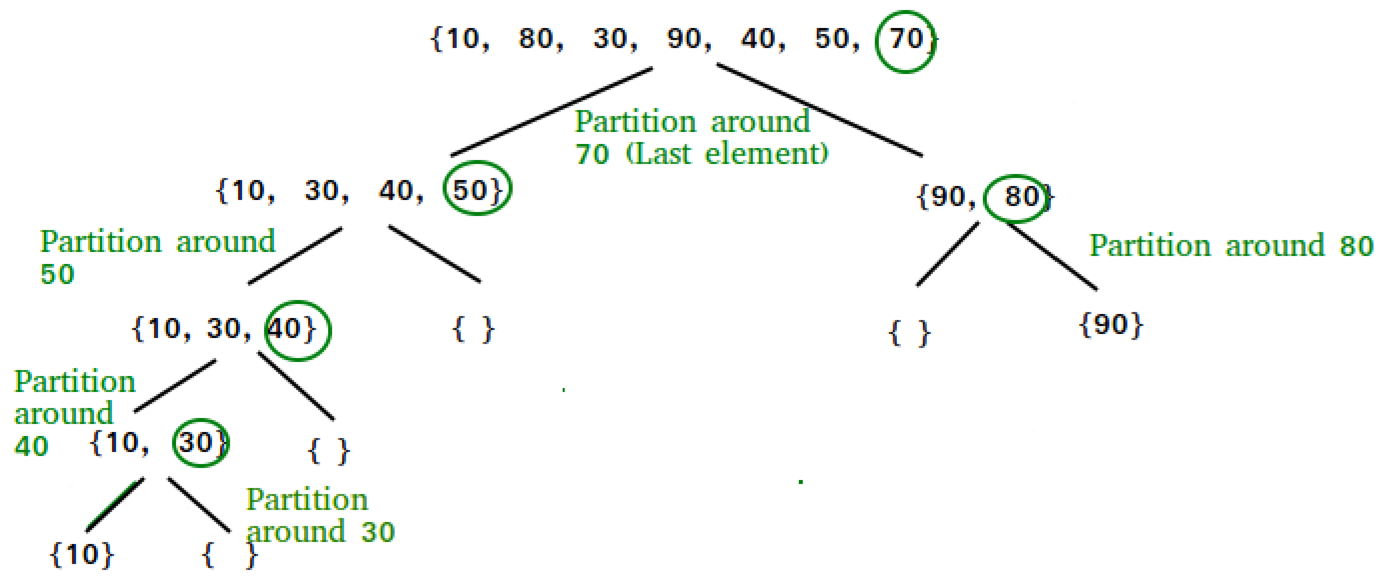
Quick sort

- QuickSort is a Divide and Conquer algorithm
- If A has 0 or 1 element, A has been sorted
- Select a pivot element x
- Partition A into two sub-array:
    - First sub-array contains all A_i with $A_i < x$
    - Second sub-array contains all A_j with $A_j \geq x$
- The sub-arrays are then sorted recursively

Quick sort

```
if |A| < 2 then return end if
x ← A[0], A1 ← {}, A2 ← {}
for i ← 0 to n-1 do
if A[i] < x then A1 ← A1 ∪ {A[i]}
else A2←A2 ∪ {A[i]} end if
end for

quickSort(A1), quickSort(A2)
A ← A1 ∪ {x} ∪ A2
```

Quick sort



{10, 80, 30, 90, 40, 50, 70}
Partition around 70 (Last element)
{10, 30, 40, 50}
{90, 80}
Partition around 50
Partition around 80
{10, 30, 40}
{ }
{ }
{90}
Partition around 40
{10, 30}
{ }
Partition around 30
{10}
{ }

Quick sort

There are many different versions of quickSort that pick pivot in different ways

- Always pick first element as pivot.
- Always pick last element as pivot (illustrated in the previous slide)
- Pick a random element as pivot.
- Pick median as pivot.

Quick sort

```
void quickSort(List &A) {
        if (!A.pHead) return;
        List A1, A2, B;
        createList(B); createList(A1); createList(A2);
        int x=removeHead(A);
        while (A.pHead) {
                t=removeHead(A);
                if (t<x) addHead(A1,createNode(t));
                else if(t>x) addHead(A2,createNode(t));
                        else addHead(B,createNode(t));
        }
        quickSort(A1); quickSort(A2);
        if (A2.pHead) {
                B.pTail->pNext = A2.pHead;  B.pTail = A2.pTail;
        }
        if (A1.pHead) {
                A1.pTail->pNext = B.pHead;  B.pHead =
A1.pHead;
        }
        A = B;
}
```

Quick sort

- Implementing quick sort in the original method requires two array A1, A2
- Solution:
  - Mark the sub-array that needs to be sorted using index b, e: $\mathrm{A}_{\mathrm{b}}, \dots, \mathrm{A}_{\mathrm{e}}$
  - This approach does not require mearging two sub array after sorting

Quick sort

```
void quickSort(int *a, int b, int e) {
        if (b >= e) return;
        int x = a[0], i = b, j = e;
        while(i < j) {
                while (a[i] < x) i++;
                while (a[j] > x) j--;
                if (i < j) {swap(a[i], a[j]); i++; j--; }
        }
        quickSort(a, b, j); quickSort(a, i,e);
}
```

Quick sort

b = 0, e = 4, x = 3

i=0	1	2	3	4	5	j=6
3	**2**	**5**	**1**	**4**	**7**	**6**

Quick sort

b = 0, e = 4, x = 3

i=0	1	2	j=3	4	5	6
3	**2**	**5**	**1**	**4**	**7**	**6**

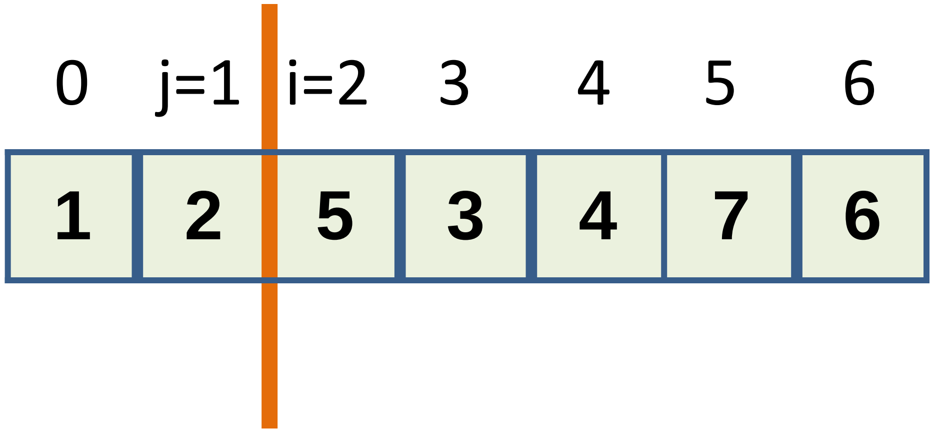
Quick sort

b = 0, e = 4, x = 3

0	i=1	j=2	3	4	5	6
1	**2**	**5**	**3**	**4**	**7**	**6**

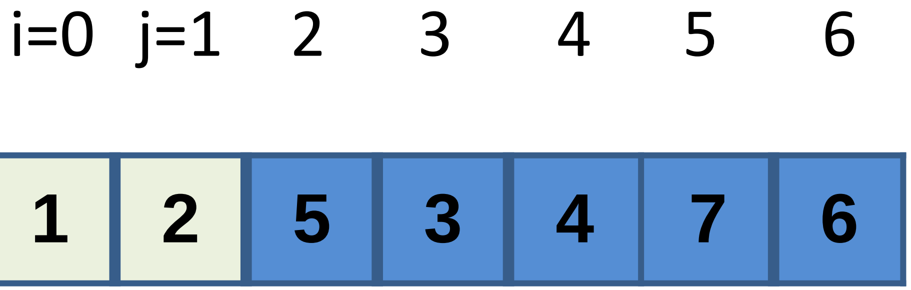
Quick sort

- b = 0, e = 4, x = 3
- Apply quick sort on two sub-array



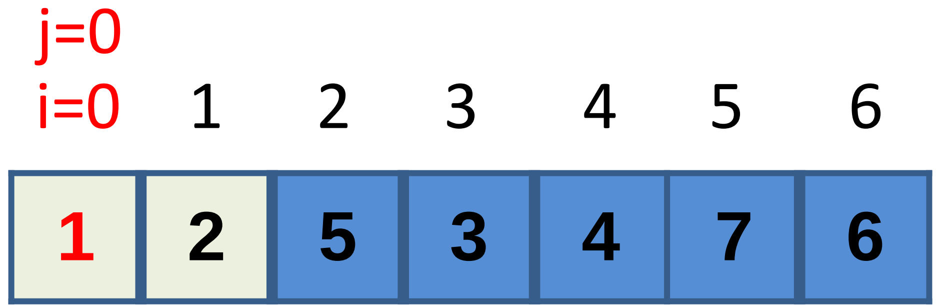
Quick sort

b = 0, e = 1, x = 1



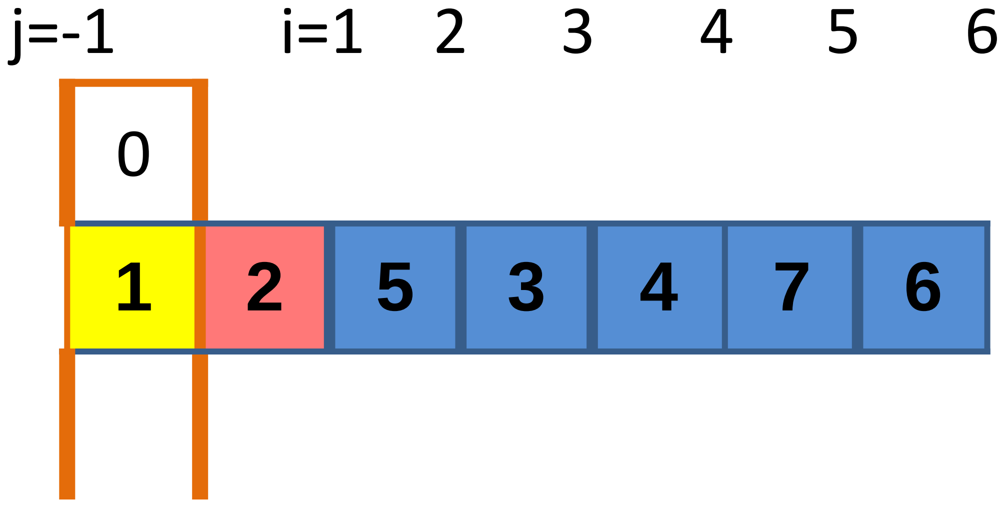
Quick sort

b = 0, e = 1, x = 1



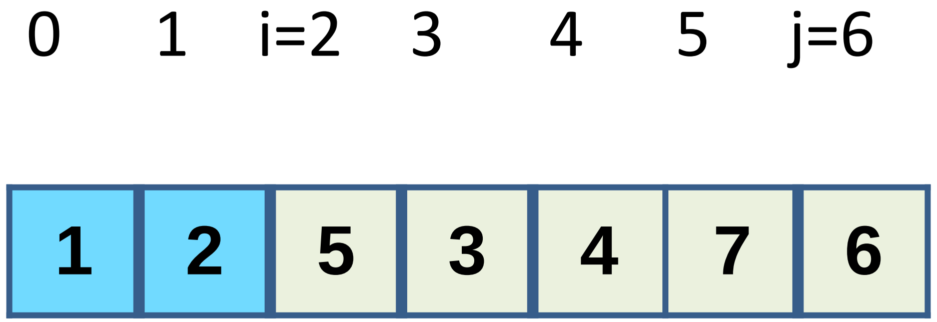
Quick sort

- b = 0, e = 1, x = 1
- □ Sort two sub-array
  - j=-1



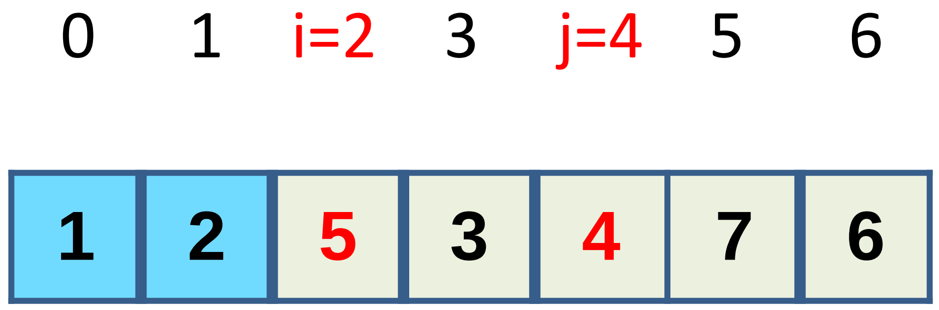
Quick sort

b = 2, e = 6, x = 5



Quick sort

b = 2, e = 6, x = 5



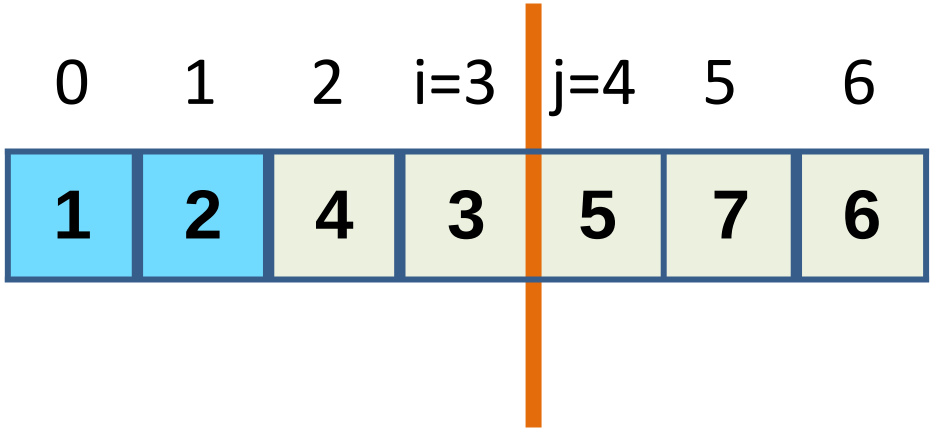
Quick sort

- x = 5

			j=3			
0	1	2	i=3	4	5	6
1	**2**	**4**	**3**	**5**	**7**	**6**

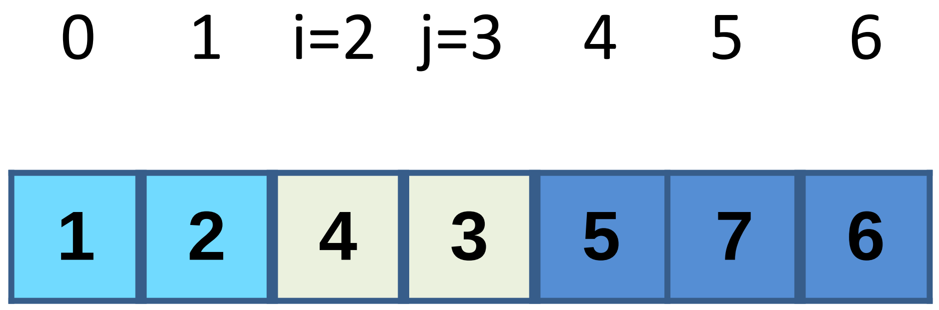
Quick sort

- b = 2, e = 6, x = 5
- □ Sort two sub-array



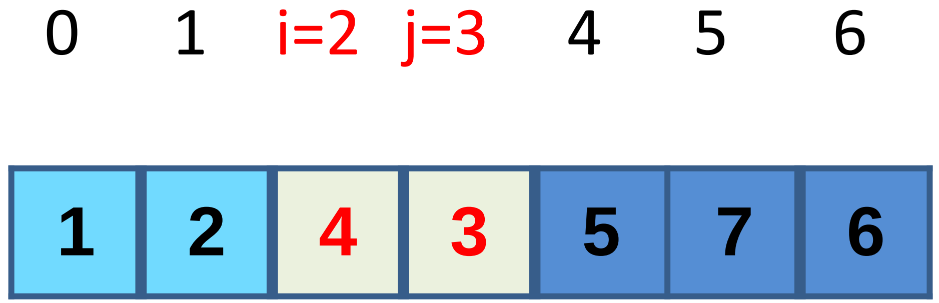
Quick sort

b = 2, e = 3, x = 4



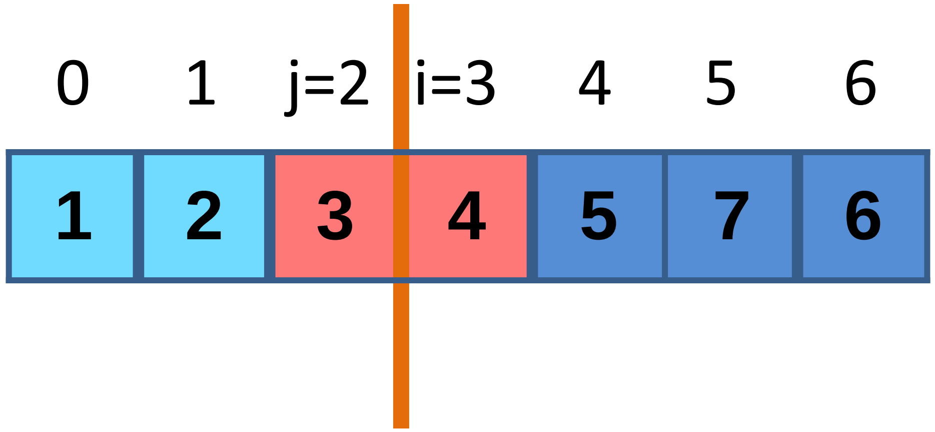
Quick sort

b = 2, e = 3, x = 4



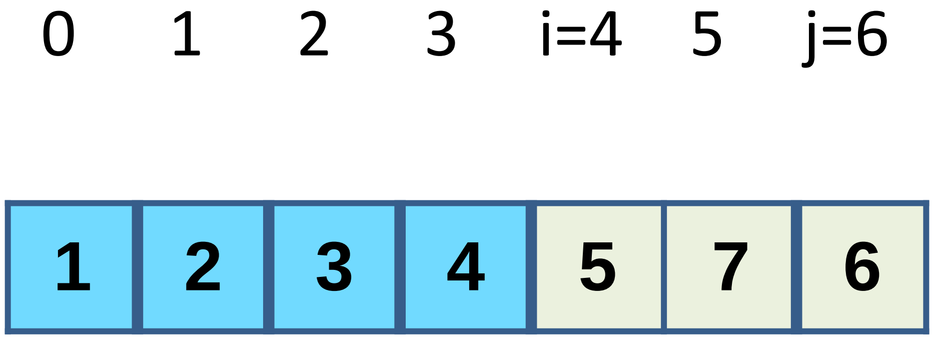
Quick sort

- b = 2, e = 3, x = 4
- □ sort two sub array



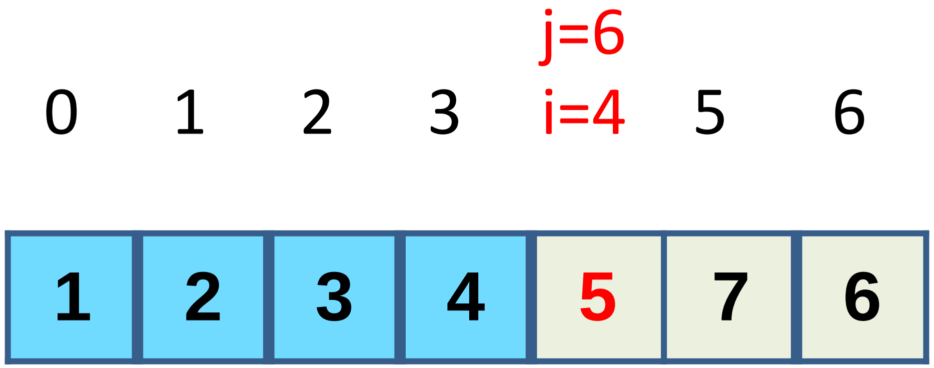
Quick sort

b = 4, e = 6, x = 5



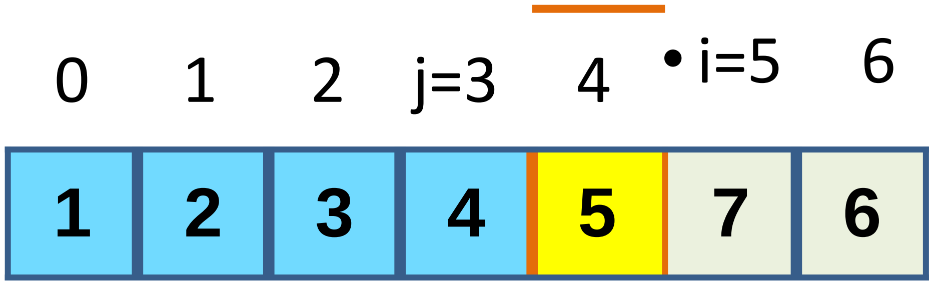
Quick sort

b = 4, e = 6, x = 5



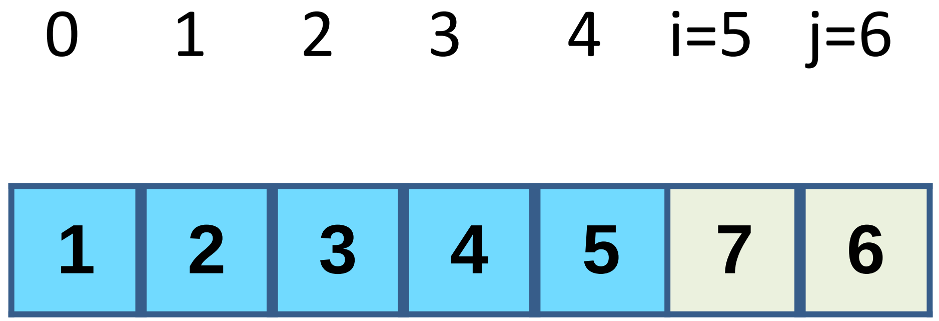
Quick sort

- b = 4, e = 6, x = 5



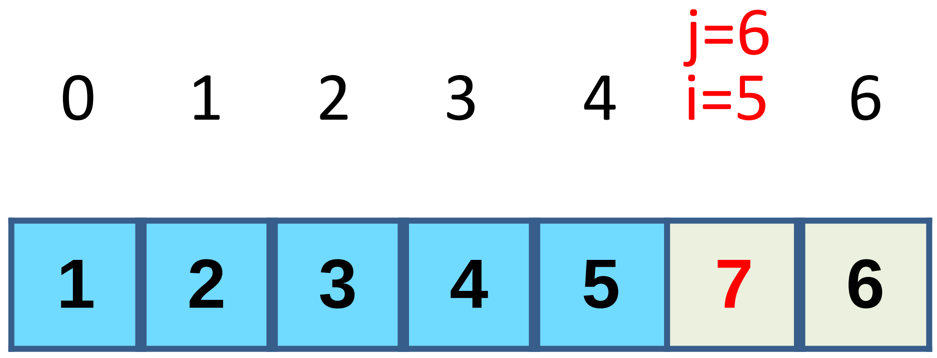
Quick sort

b = 5, e = 6, x = 7



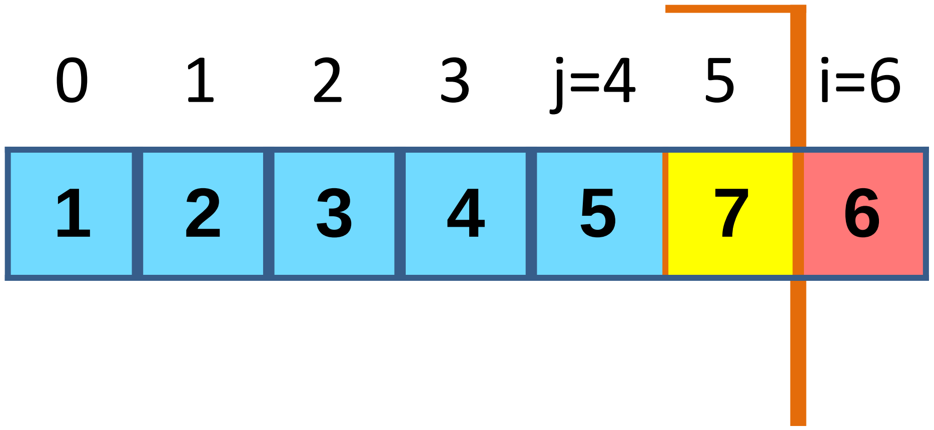
Quick sort

b = 5, e = 6, x = 7



Quick sort

- b = 5, e = 6, x = 7
- □ Sorting two sub-array



Quick sort

b = 0, e = 6,

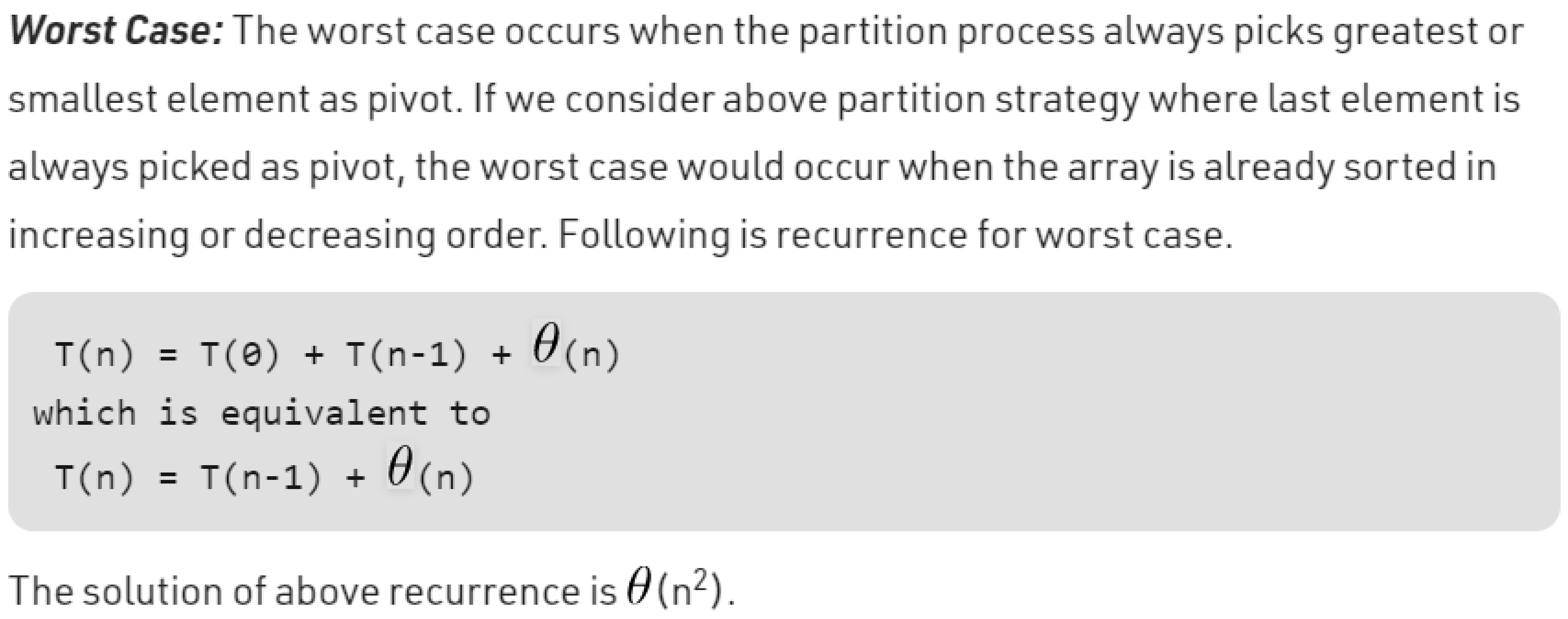
- Result

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

Quick sort

- Time taken by QuickSort in general can be written as following.
- $T(n) = T(k) + T(n-k-1) + \theta(n)$
- The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.
- The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

Quick sort

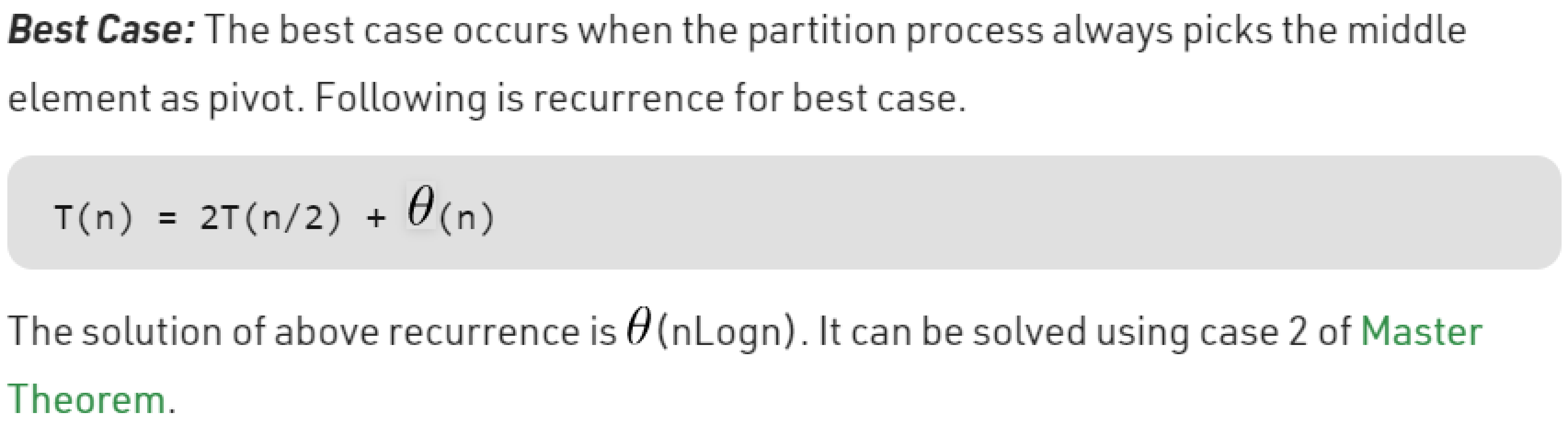


Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

```
 T(n) = T(0) + T(n-1) + θ(n)
which is equivalent to
 T(n) = T(n-1) + θ(n)
```

The solution of above recurrence is $\theta(n^2)$.

Quick sort



Best Case: The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

```
T(n) = 2T(n/2) + θ(n)
```

The solution of above recurrence is θ(nLogn). It can be solved using case 2 of Master Theorem.

Quick sort

Average Case:

To do average case analysis, we need to consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy.

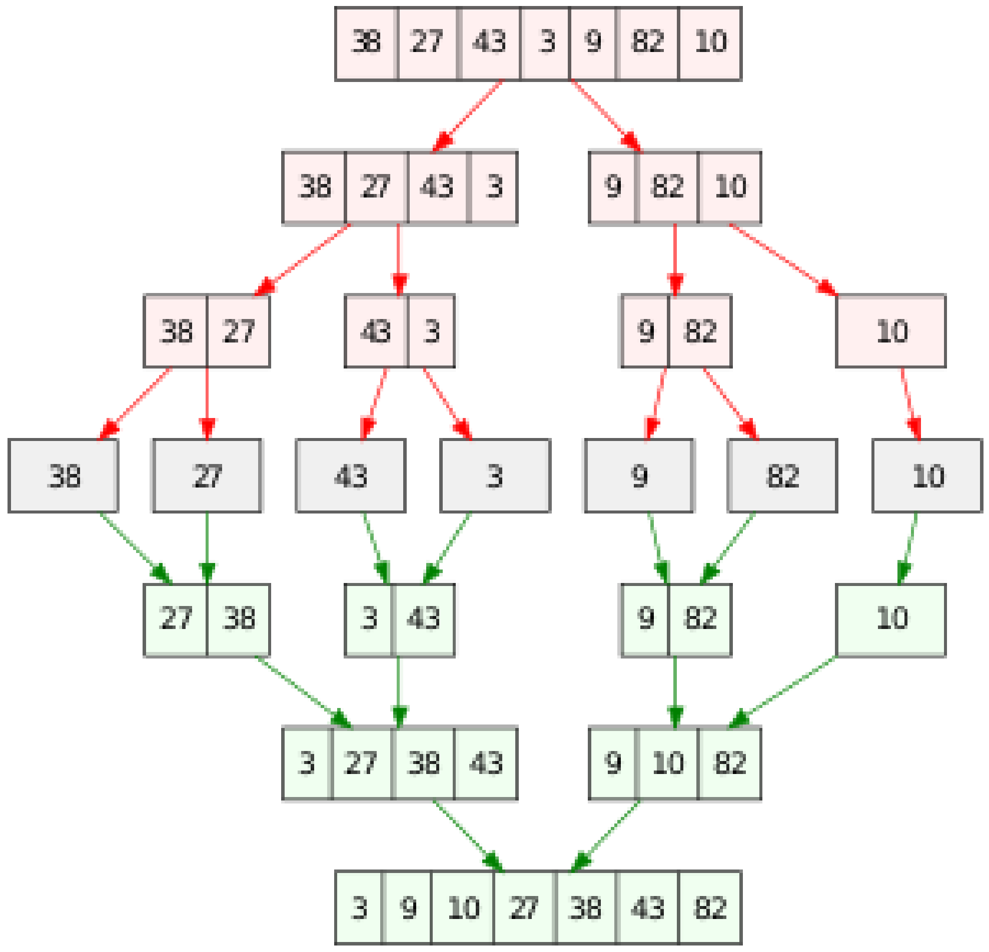
We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

```
T(n) = T(n/9) + T(9n/10) + θ(n)
```

Solution of above recurrence is also O(nLogn)

Although the worst case time complexity of QuickSort is O(n^2) which is more than many other sorting algorithms like Merge Sort and Heap Sort, QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

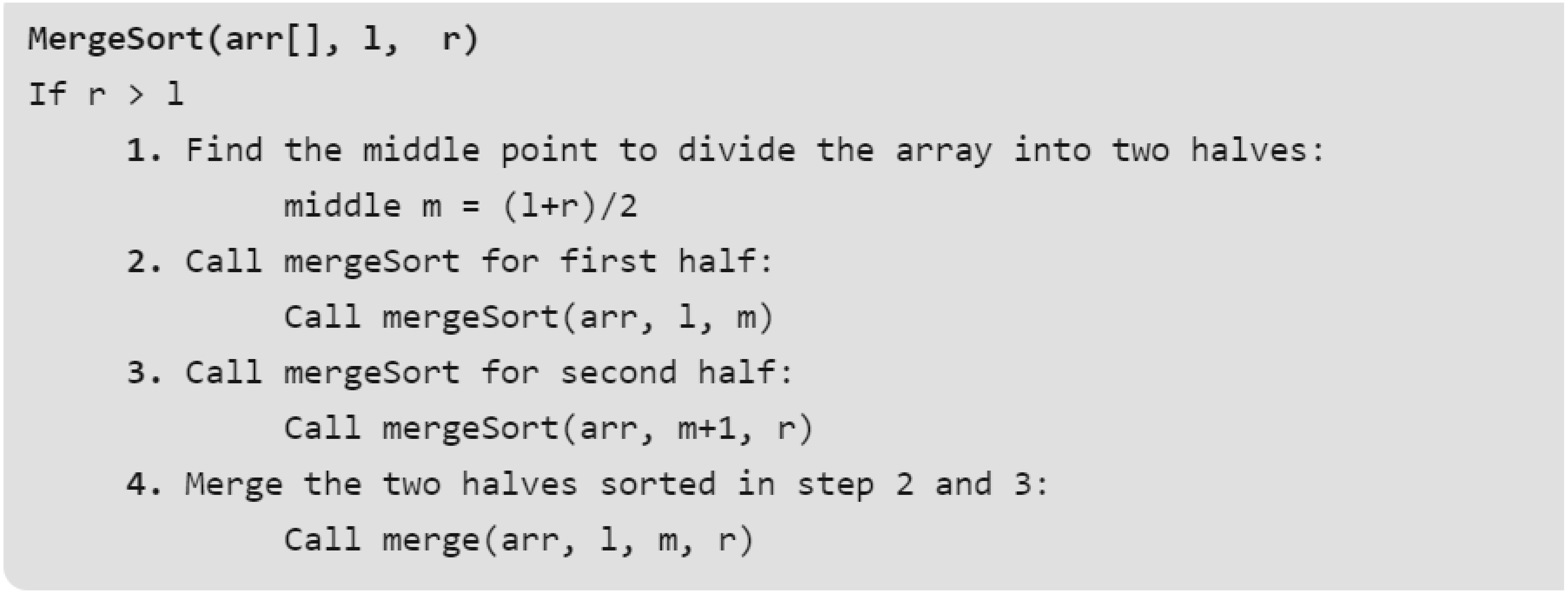
Merge sort


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Merge sort

- Like QuickSort, Merge Sort is a Divide and Conquer algorithm.
- It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.
- The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

Merge sort



```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = (l+r)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

Merge sort

```
if |A| < 2 then
    return
end if
partition(A,A1,A2)
mergeSort(A1)
mergeSort(A2)
merge(A1,A2,A)
```

Merge sort

Algorithm: (merge(A1 ,A2 ,A))

```
A ← {}
while |A1|>0 and |A2|>0 do
        if A1[0] < A2[0] then x ← A1[0], A1\{x}
        else x ← A2[0], A2\{x} end if
        A ← A ∪ {x}
end while

while |A1|>0 do
x ← A1[0], A1\{x} A ← A ∪ {x}  end while
while |A2|>0 do
x ← A2[0], A2\{x} A ← A ∪ {x}  end while
```

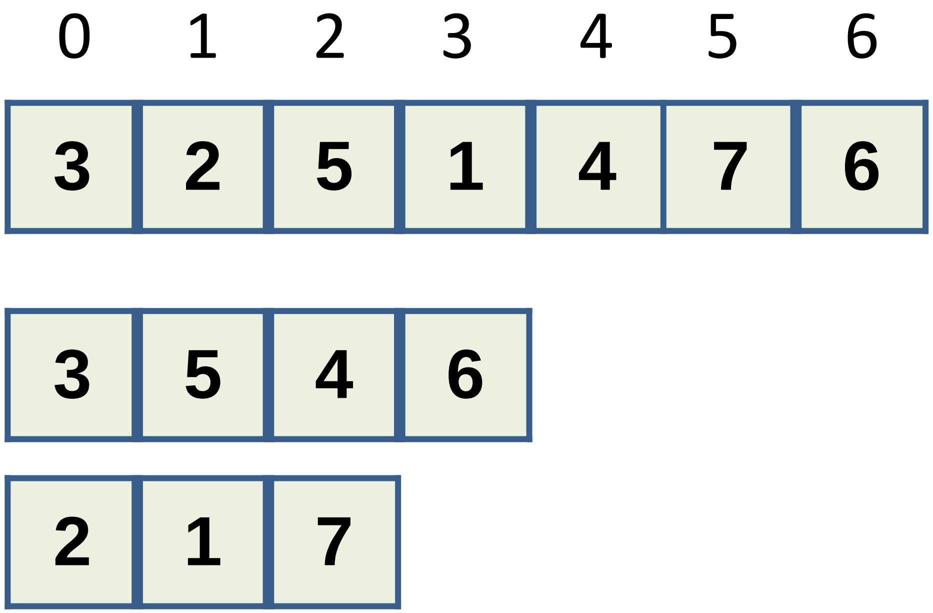
Merge sort

Sort A

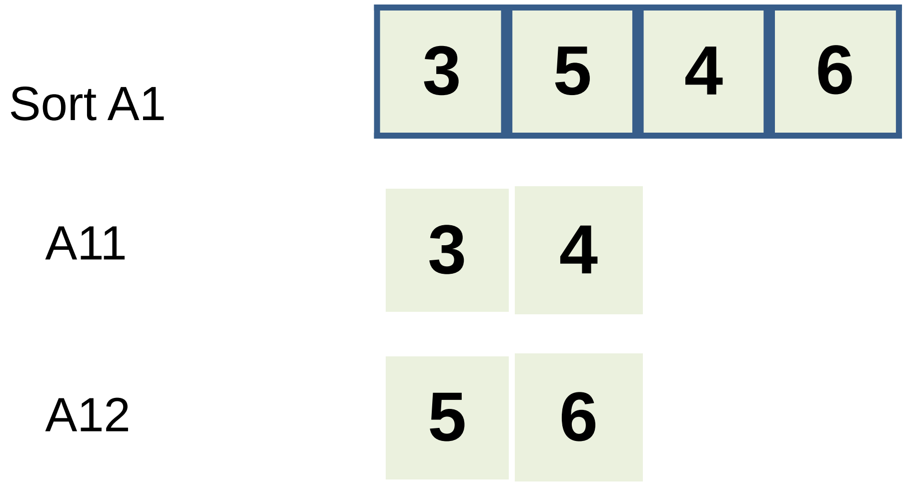
A

A_1

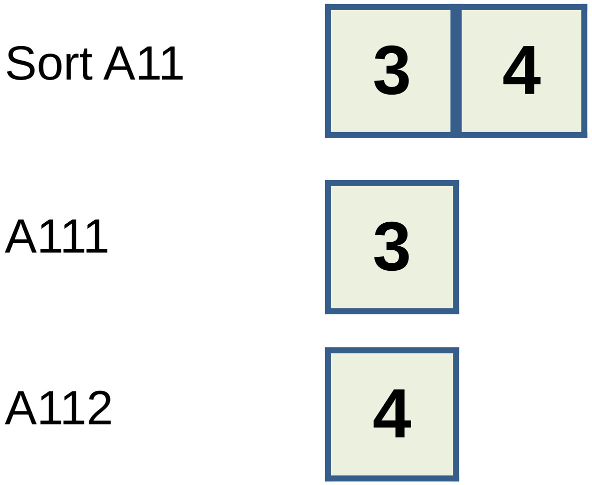
A



Merge sort

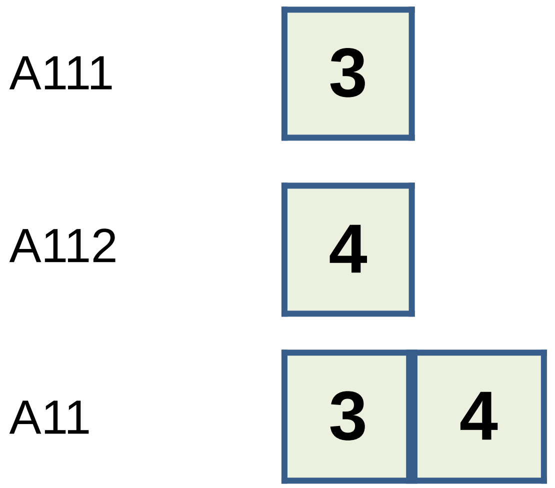


Merge sort

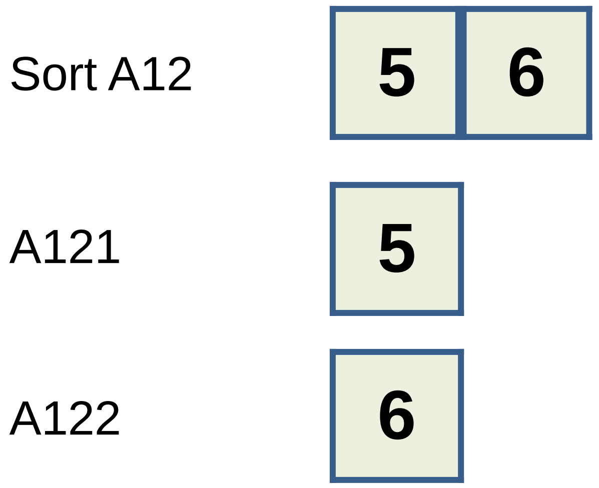


Merge sort

Merge A11



Merge sort



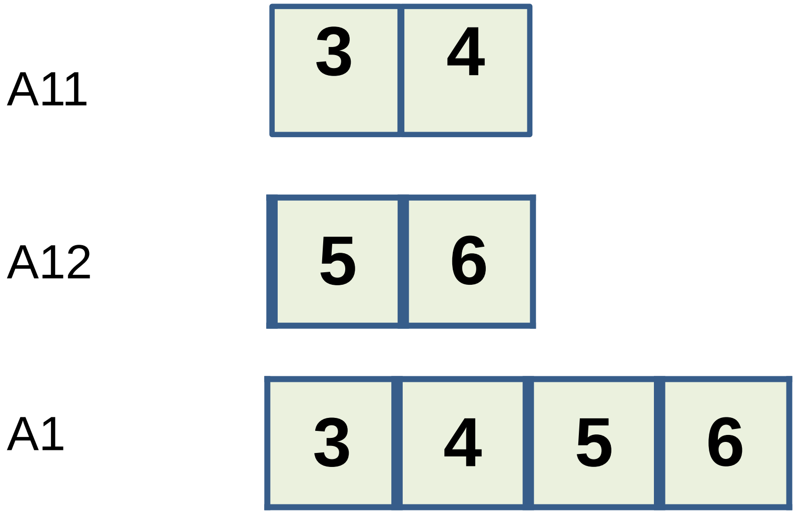
Merge sort

Sort A12

A121	5	
A122	6	
A12	5	6

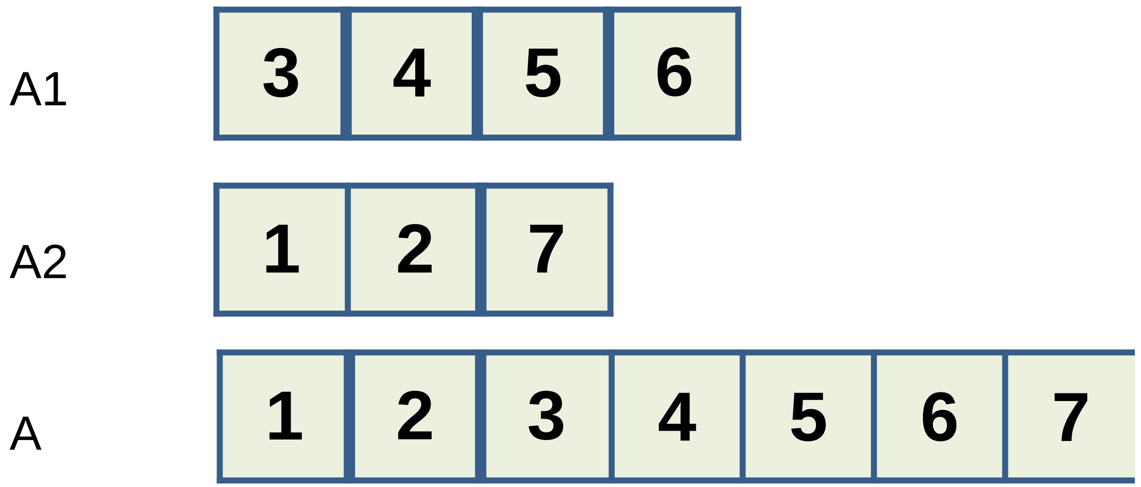
Merge sort

Merge A1



Merge sort

Merge A



Merge sort

```
void mergeSort(List &A) {
        if (!A.pHead) return;
        List A1, A2;
        partition(A,A1,A2);
        mergeSort(A1);
        mergeSort(A2);
        merge(A1, A2, A);
}
```

Merge sort

```
void partition(List &A,List &A1,List &A2){
        int lane = 0;
        createList(A1); createList(A2);
        while (A.pHead) {
                Node *p = a.pHead;
                A.pHead = p->pNext; p->pNext = NULL;
                if (lane) addTail(A2, p)
                else addTail(A1, p);
                lane = !lane;
        }
}
```

Merge sort

```
void merge(List &A1,List &A2,List &A){
        Node *p;
        while(A1.pHead && A2.pHead) {
                if (A1.pHead->info < A2.pHead->info){
                        p = A1.pHead; A1.pHead = p->pNext;
                }
                else {
                p=A2.pHead; A2.pHead=p->pNext;
                }
        p->pNext = NULL;
        addTail(A, p);
        }
```

Merge sort

```
    while (A1.pHead) {
        p = A1.pHead; A1.pHead = p->pNext;
        p->pNext = NULL;
        addTail(A, p);
    }
    while (A2.pHead) {
        p = A2.pHead; A2.pHead = p->pNext;
        p->pNext = NULL;
        addTail(A, p);
    }
}
```

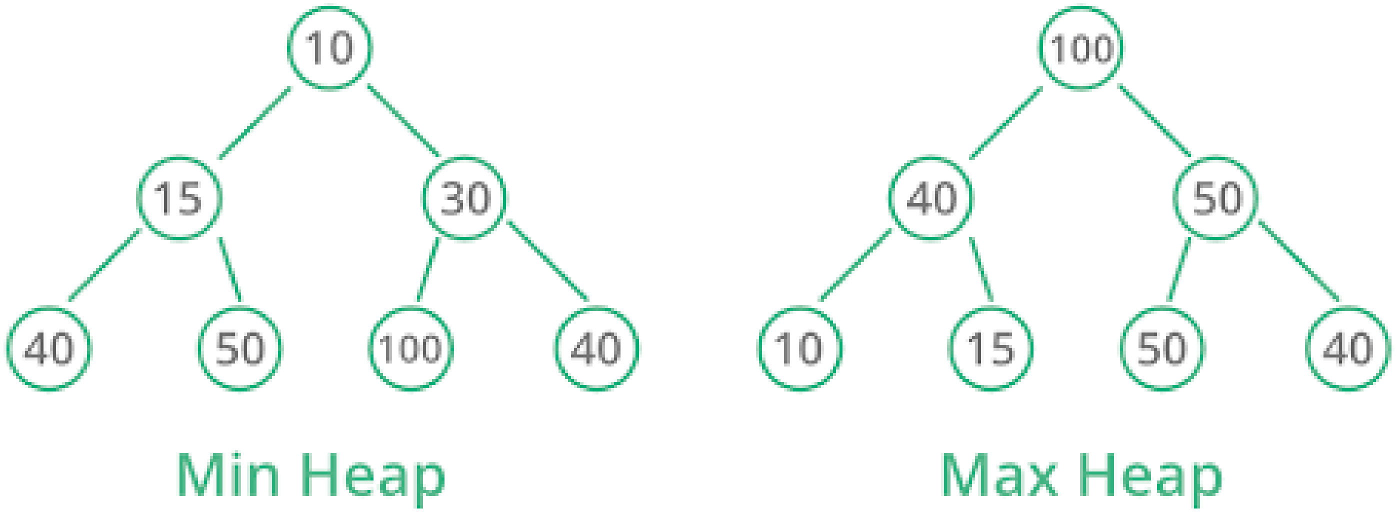
Merge sort

Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$T(n) = 2T(n/2) + \theta(n)$

Heap sort

- Binary Heap



Heap sort

Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.

2. At this point, the largest item is stored at the root of the heap.

Replace it with the last item of the heap followed by reducing the size

of heap by 1. Finally, heapify the root of the tree.

3. Repeat step 2 while size of heap is greater than 1.

Heap sort (how to build the heap)

Input data: 4, 10, 3, 5, 1

```
       4(0)
      /  \
   10(1)   3(2)
   /  \
 5(3)   1(4)
```

The numbers in bracket represent the indices in the array representation of data.

Heap sort

Applying heapify procedure to index 1:

```
      4(0)
     /  \
  10(1)   3(2)
  /  \
5(3)   1(4)
```

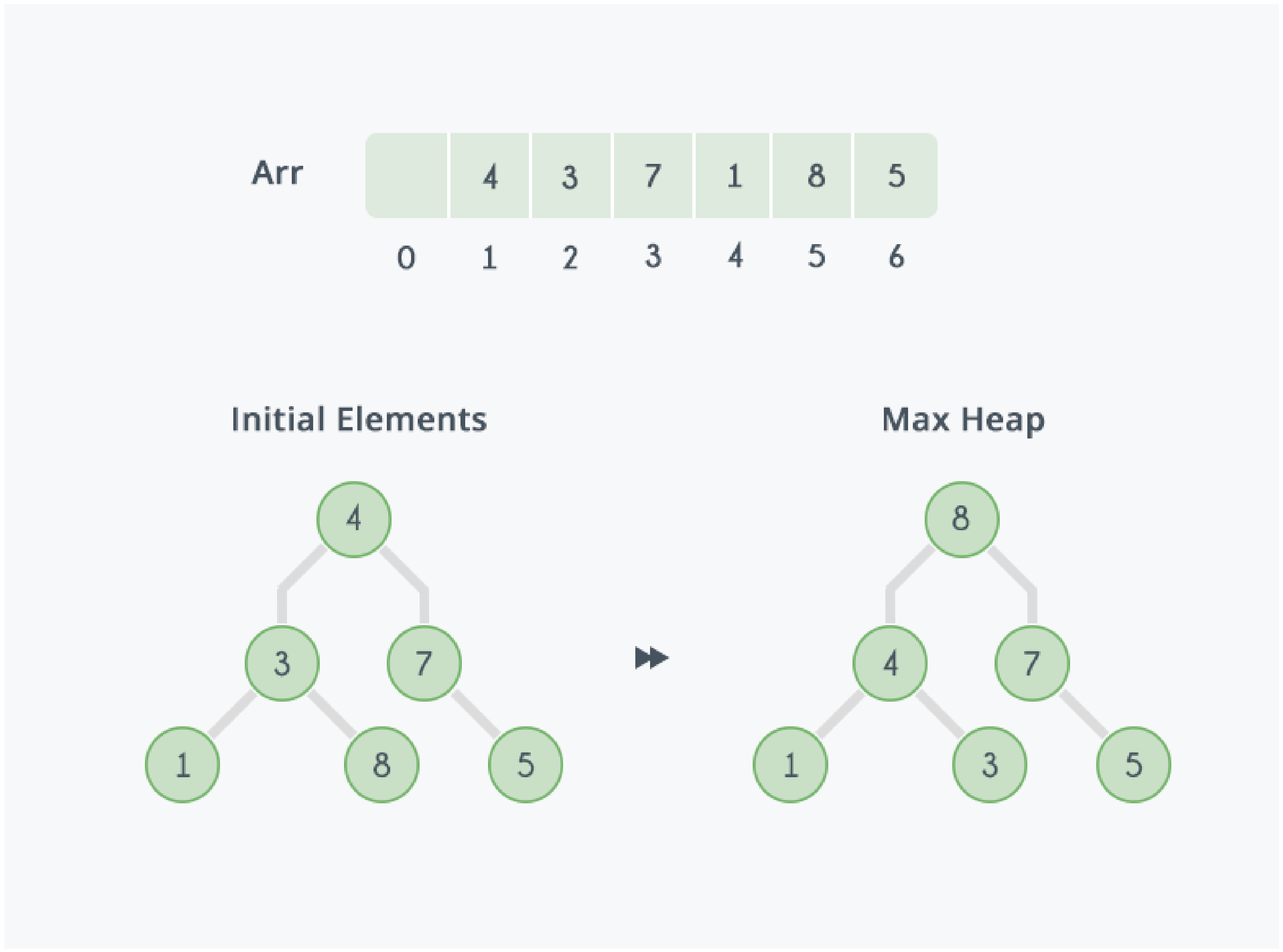
Heap sort

Applying heapify procedure to index 0:

```
      10(0)
      /  \
    5(1)  3(2)
   /   \
 4(3)   1(4)
```

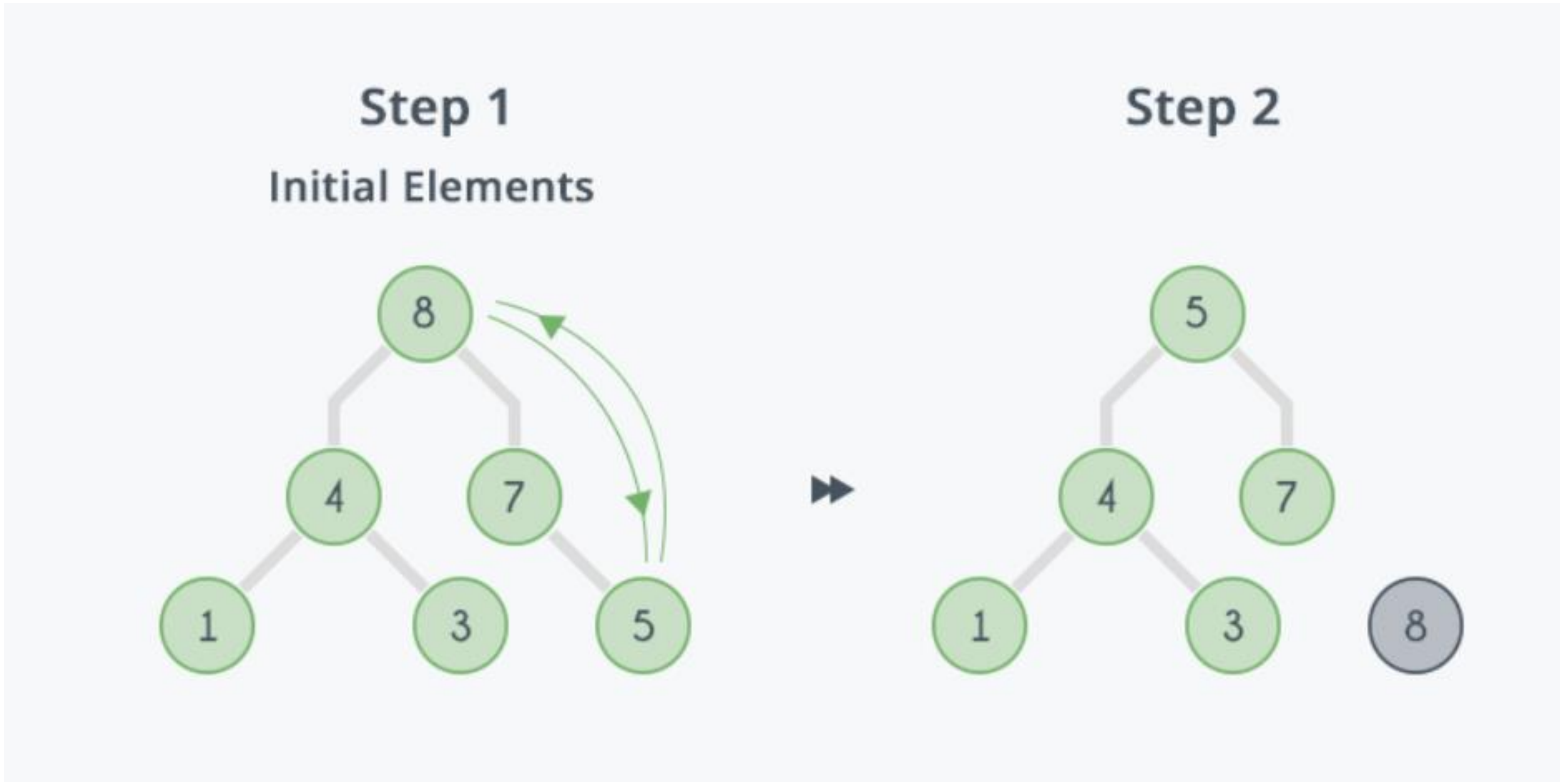
The heapify procedure calls itself recursively to build heap
in top down manner.

Heap sort

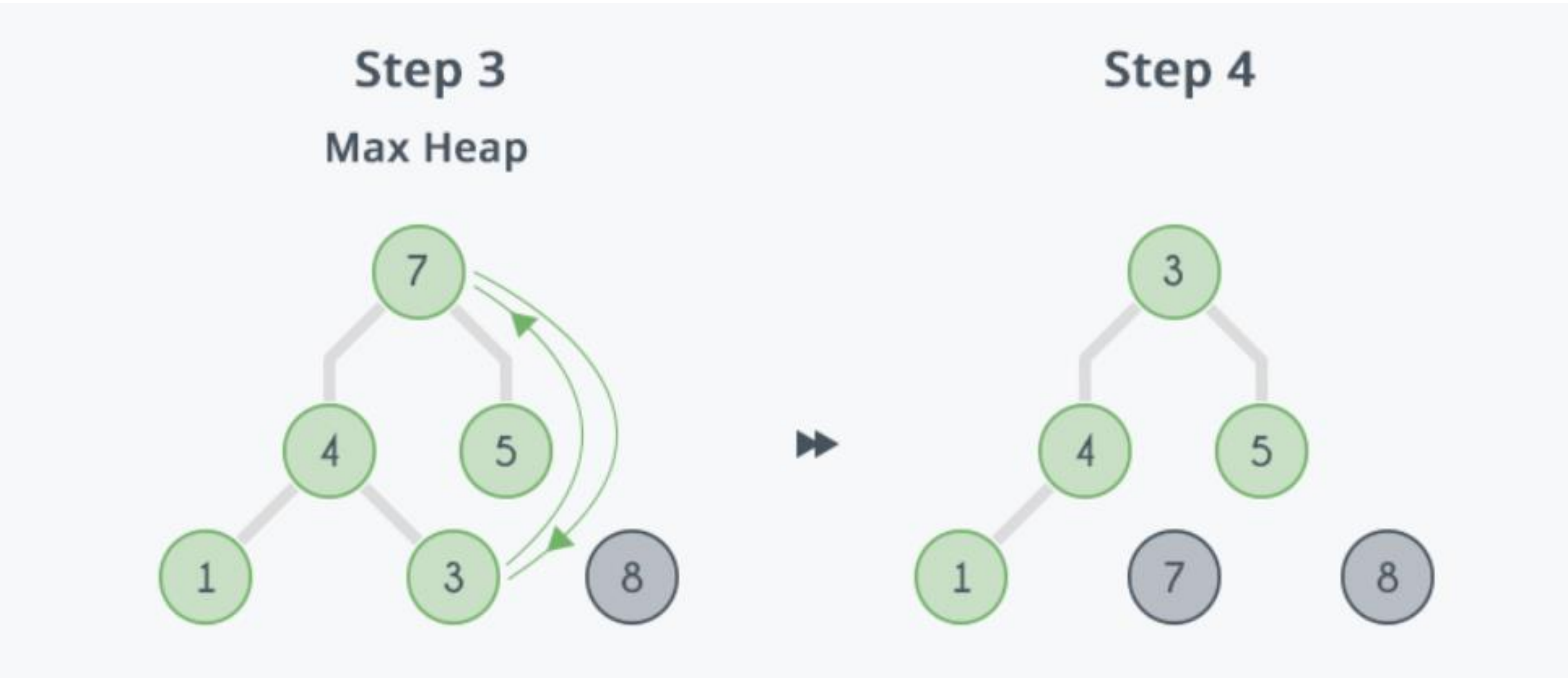


Arr
4
3
7
1
8
5
0
1
2
3
4
5
6
Initial Elements
4
3
7
1
8
5
Max Heap
8
4
7
1
3
5

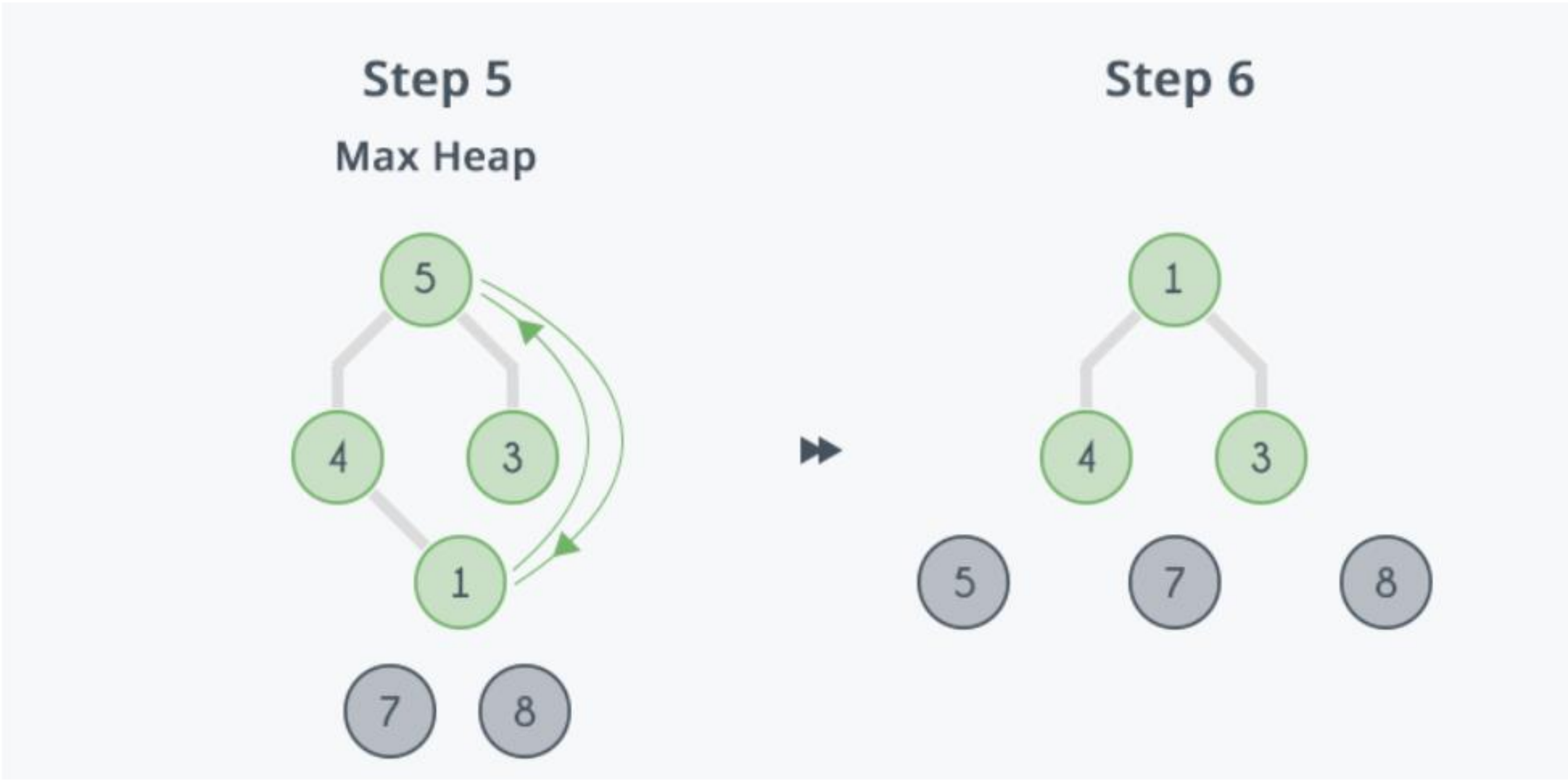
Heap sort



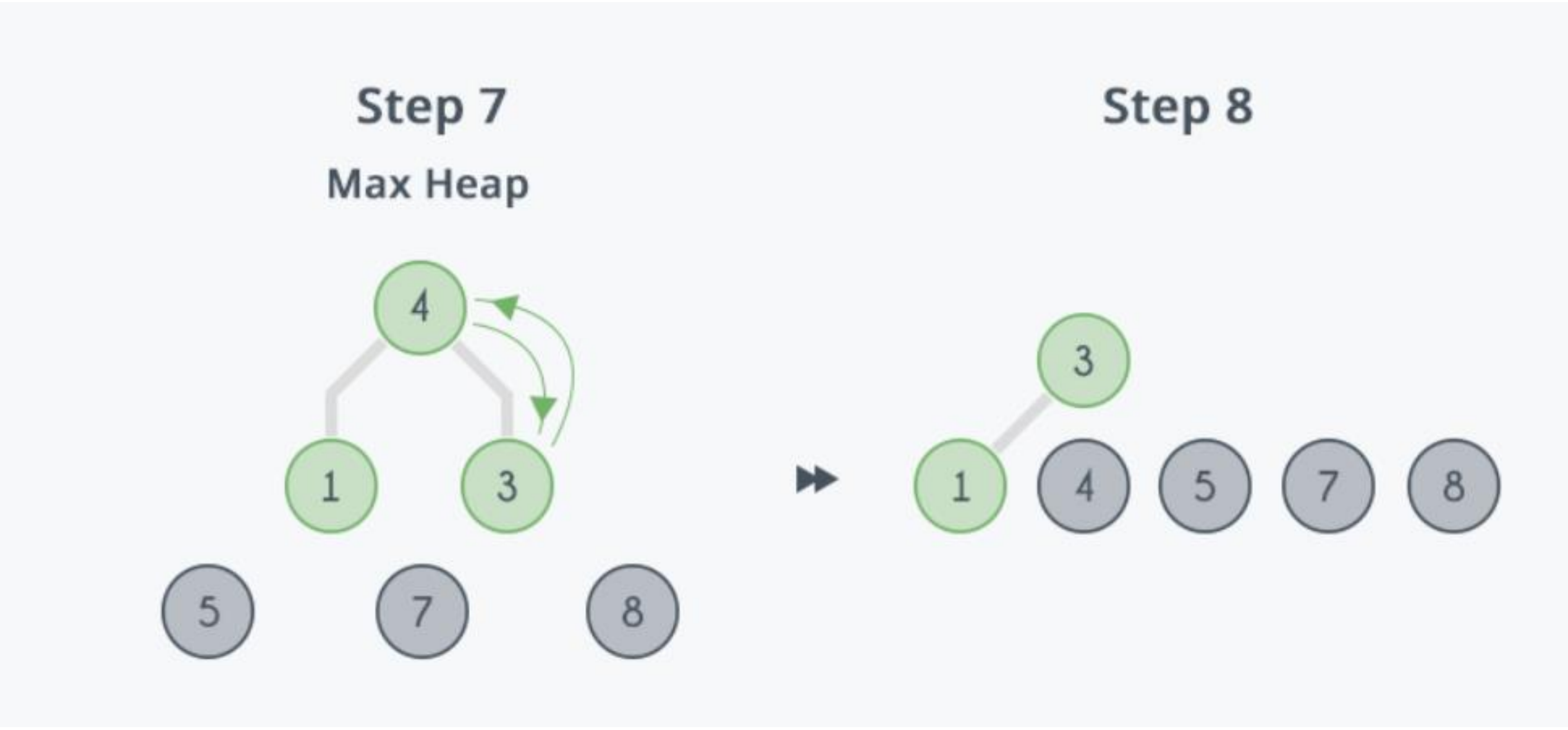
Heap sort



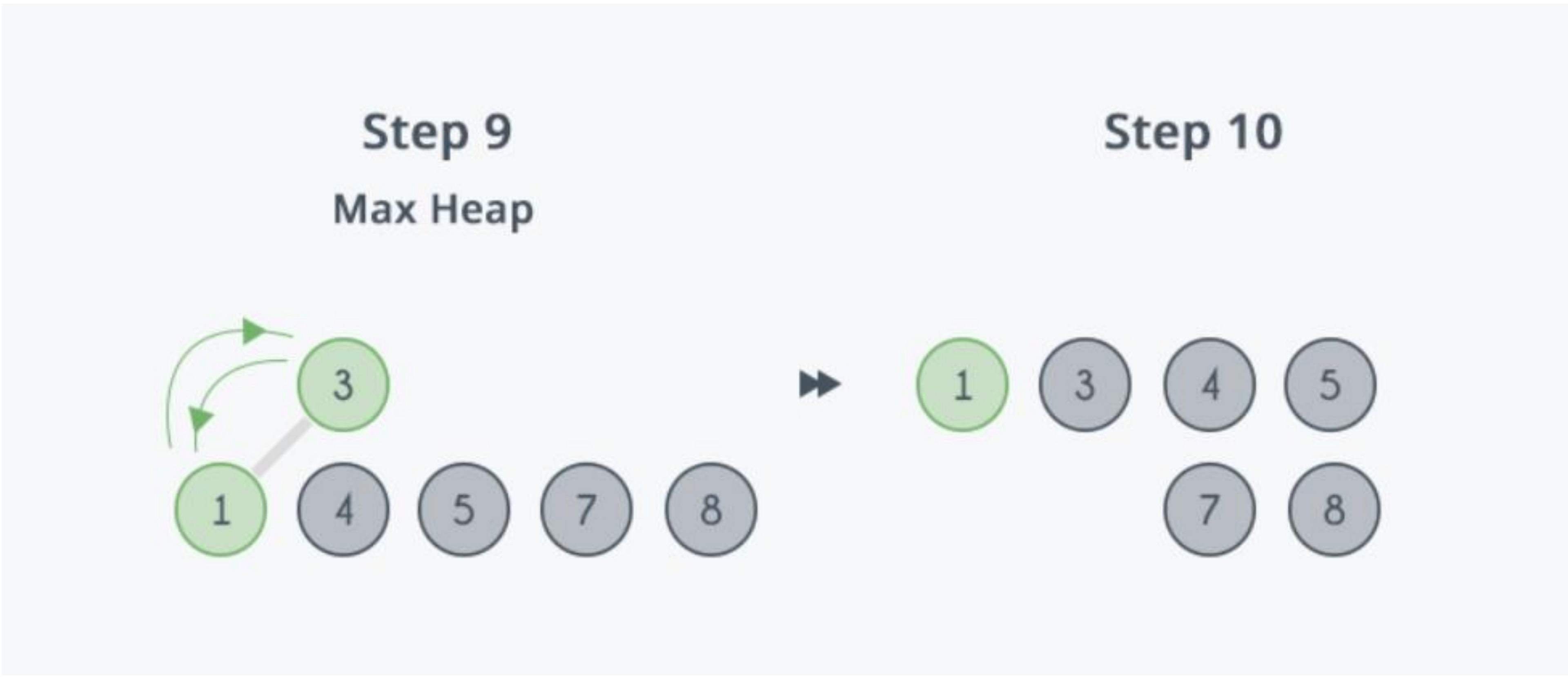
Heap sort



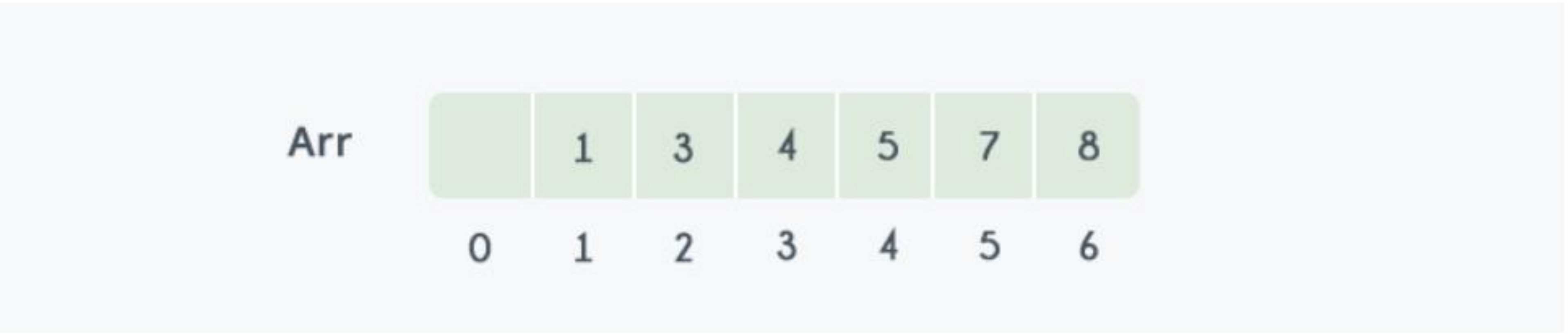
Heap sort



Heap sort



Heap sort



Heap Sort

```
void heapify(int *a, int k, int n) {
   int j = 2*k+1;
   while (j < n) {
         if (j + 1 < n)
            if (a[j] < a[j + 1]) j = j + 1;
         if (a[k] >= a[j]) return;
         swap(a[k], a[j]);
         k = j; j = 2*k + 1;
   }
}
```

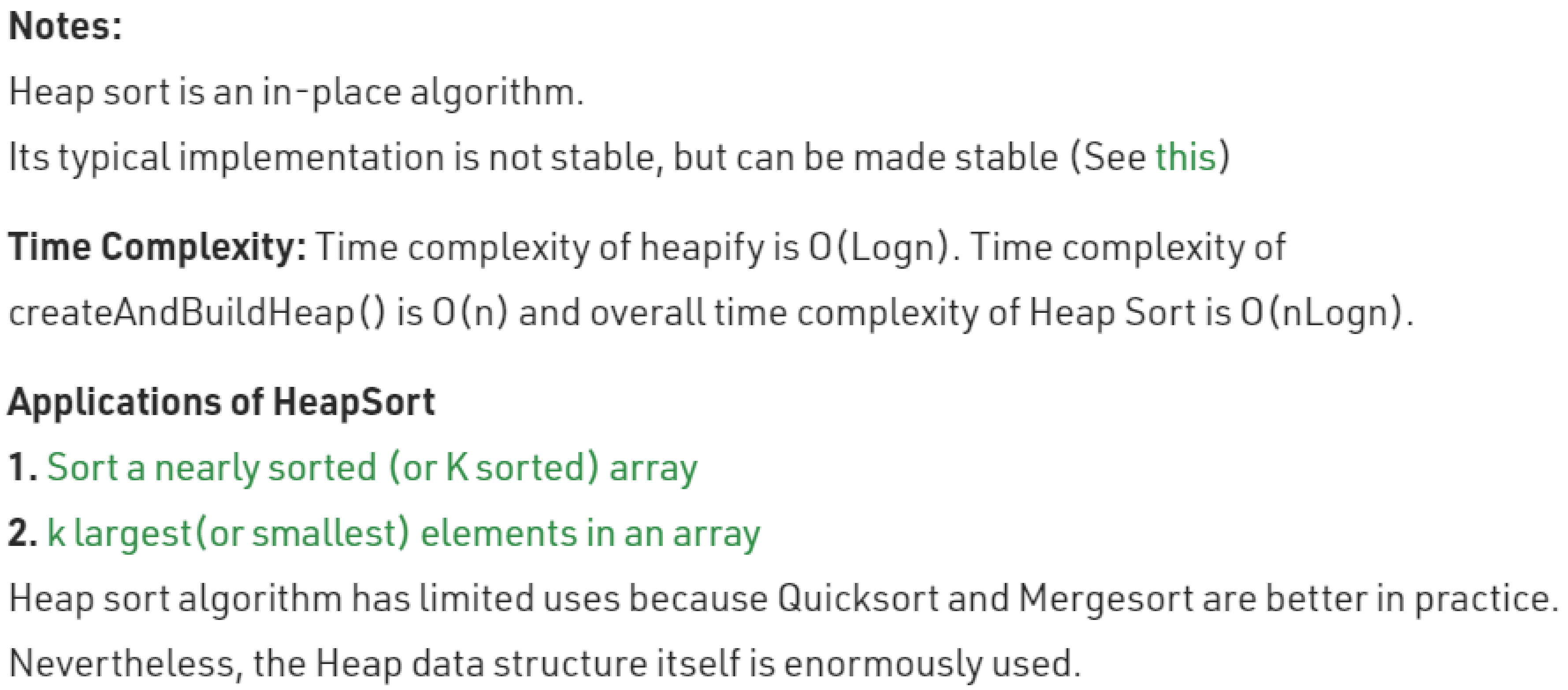
Heap Sort

```
void buildHeap(int *a, int n) {
   int i;
   i = (n - 1) / 2;
   while (i >= 0) {
         heapify(a, i, n);
         i--;
   }
}
```

Heap sort

```
void heapSort(int *a, int n) {
   buildHeap(a, n);
   while (n > 0) {
         n = n - 1;
         swap(a[0], a[n]);
         heapify(a, 0, n);
   }
}
```

Heap Sort



Notes:

Heap sort is an in-place algorithm.

Its typical implementation is not stable, but can be made stable (See this)

Time Complexity: Time complexity of heapify is O(Logn). Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(nLogn).

Applications of HeapSort

1. Sort a nearly sorted (or K sorted) array

2. k largest(or smallest) elements in an array

Heap sort algorithm has limited uses because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used.