



DET102 Data Structures and Algorithms

Lecture 04: Sorting Algorithms

The problem of sorting

- **Input:** n numbers in an array (a list). $A[1..n]$
- **Output:** permutation $B[1..n]$ of A such that $B[1] \leq B[2] \leq B[3] \leq \dots \leq B[n]$.

For example:

$A = [2, 5, 1, 6, 3, 4] \longrightarrow B = [1, 2, 3, 4, 5, 6]$

- Increasing order: from small to large
- Decreasing order: from large to small
- How to sort ?
 - Array/list
 - loop

Original records

ID	A	B
Player1	70	80
Player2	90	95
Player3	95	60
Player4	80	95

Decreasing order in A

ID	A	B
Player3	95	60
Player2	90	95
Player4	80	95
Player1	70	80

A: Sort Key

We will need **struct** or **class** to deal with records sorting.

Stable sort

- A sorting algorithm is *stable* in which the two elements that are equal remain in the same relative position after performing the sorting.

ID	A	B
Player1	70	80
Player2	90	95
Player3	95	60
Player4	80	95

Original records

ID	A	B
Player2	90	95
Player4	80	95
Player1	70	80
Player3	95	60

Stable sorting

ID	A	B
Player4	80	95
Player2	90	95
Player1	70	80
Player3	95	60

Unstable sorting



Sorting algorithms

- Time complexity
- Stable or not ?
- Extra memory is needed ?
- Features of the input data affect the time complexity ?

In-place

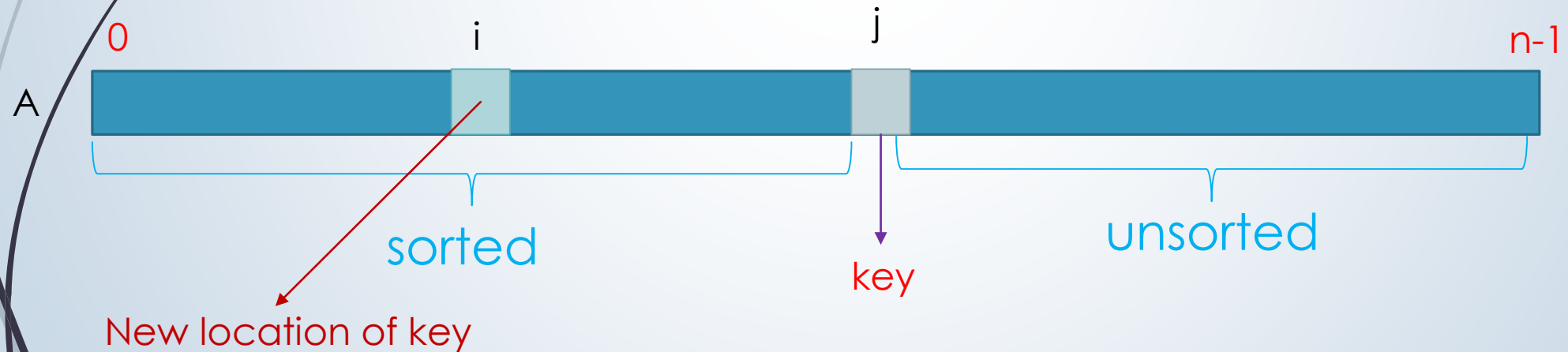
A sorting algorithm is *in-place* only if a constant number of data elements of an input array are ever stored outside the array. No additional storage is required and it is possible to sort a large list without the need of additional working storage.

Insertion sort

InsertionSort (A, n)

for $j=1$ to $n-1$

insert key $A[j]$ into the (*already sorted*) sub-array $A[0..j-1]$
by pairwise key-swaps down to its right position





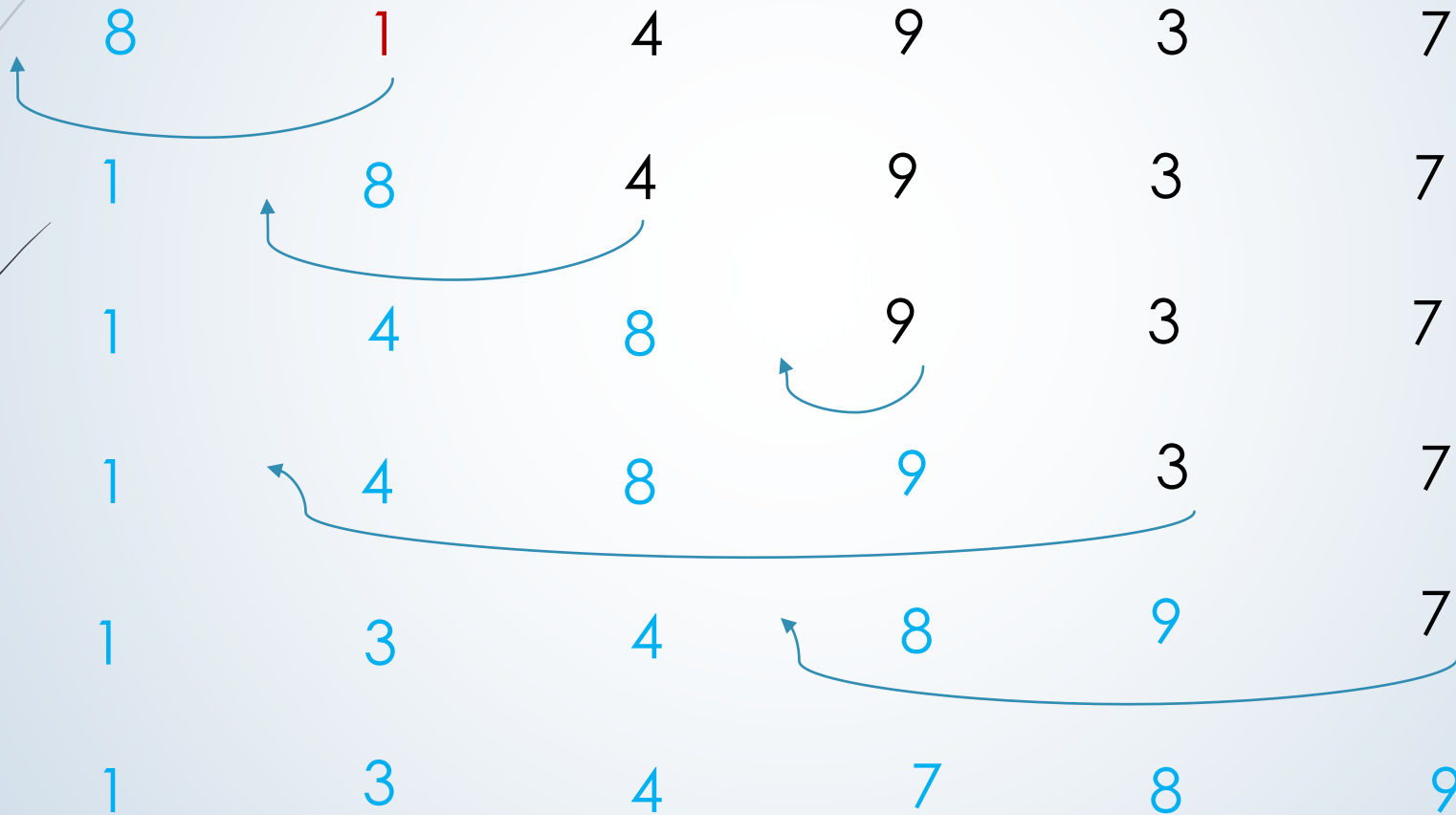
Insertion sort

```
for (a scan from 2nd item to end){  
    copy unsorted entry from the list;  
    shift previous entries;  
    insert the unsorted entry to correct location;  
}
```

<https://www.youtube.com/watch?v=qktBUYMO7o8>

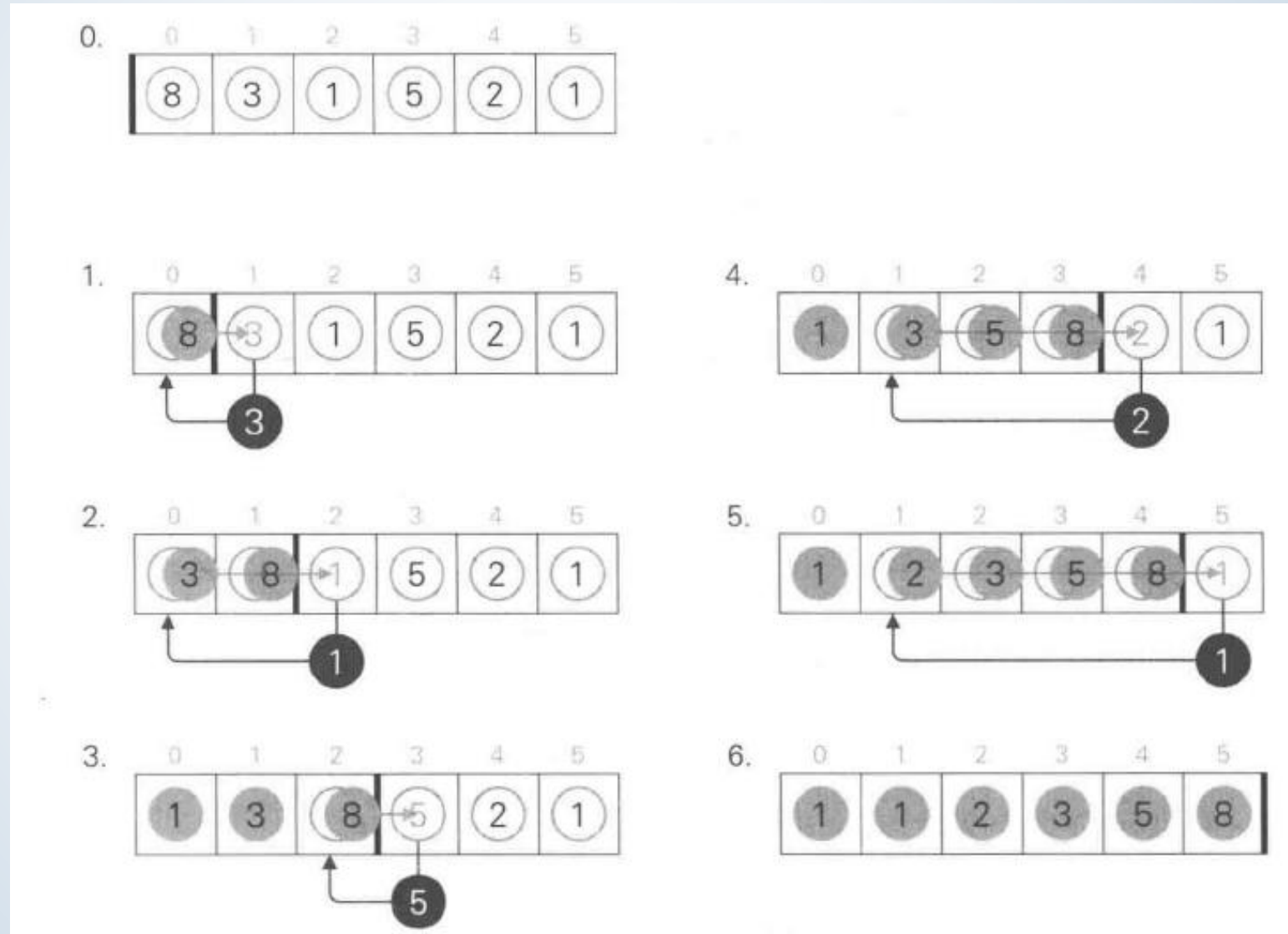
Example of insertion sort

$n=6$
 $n-1=5$ rounds



$A = \{8, 3, 1, 5, 2, 1\}$

Is it a stable sort ?



Insertion Sort

insertionSort (A, N)

// A is an array that has N items, index from 0.

- 1. for i = 1 to A.length-1**
- 2. key = A[i]**
- 3. /* insert A[i] into the sorted sequence A[0,...,j-1] */**
- 4. j = i - 1**
- 5. while j >= 0 and A[j] > key**
- 6. A[j+1] = A[j]**
- 7. j--**
- 8. A[j+1] = key**



Exercise: InsertionSort

If your answer is wrong

- Is your array large enough ?
- Does the array start from 0 or 1 ?
- Do you use loop variables i, j correctly ?
- Any extra blank space or new lines ?



Insertion Sort

- **Time Complexity:** $O(n^2)$
- **Auxiliary Space:** $O(1)$
- **In Place:** Yes
- **Stable:** Yes
- **Uses**
 - Insertion sort is used when number of elements is small.
 - It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

Insertion Sort Complexity (1)

- The **best case** input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., $O(n)$).
 - During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array.

Insertion Sort Complexity (2)

- ▶ The simplest **worst case** input is an array sorted in reverse order.
- ▶ The set of all worst case inputs consists of all arrays where each element is the smallest or second-smallest of the elements before it. In these cases every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e., $O(n^2)$).

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



Insertion Sort Complexity (3)

- The **average case** is also quadratic, which makes insertion sort impractical for sorting large arrays.
- However, insertion sort is **one of the fastest algorithms for sorting very small arrays**, even faster than quicksort; indeed, good quick sort implementations use insertion sort for arrays smaller than a certain threshold, also when arising as subproblems; the exact threshold must be determined experimentally and depends on the machine, but is commonly around **ten**.

Bubble sort

Starting from the beginning of the list,

- compare every adjacent pair, swap their position if they are not in the right order (the latter one is smaller than the former one).
- After each iteration, **one less element (the last one) is needed to be compared** until there are no more elements left to be compared.

From Wikipedia https://en.wikipedia.org/wiki/Bubble_sort

How many passes are there ?

What is the asymptotic complexity of bubble sort ?

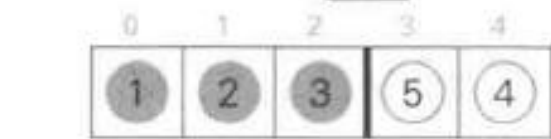
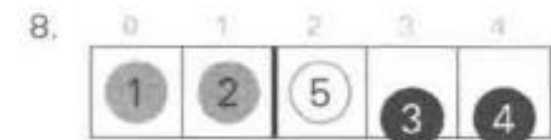
Bubble Sort

bubbleSort (A, N)

// A is an array consisting of N items. Index from 0

1. flag = 1 // there exists at least one reverse-order pair
2. while flag > 0
3. flag = 0
4. for j from N-1 to **1**
5. if $A[j] < A[j-1]$
6. swap $A[j]$ and $A[j-1]$
7. flag = 1

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



Improved Bubble Sort

bubbleSort (A, N)

1. flag = 1
2. i = 0 // starting index of the unsorted part
3. while flag > 0
4. flag = 0
5. for j from N-1 to i+1
6. if A[j] < A [j-1]
7. swap A [j] and A [j-1]
8. flag = 1
9. i++



Bubble Sort

- The previous program sorts an array's values into ascending order.
- The technique is called the ***bubble sort*** or the ***sinking sort*** because the smaller values gradually “bubble” their way upward to the top of the array like air bubbles rising in water, while the larger values sink to the bottom of the array of elements are compared.
- Complexity: $O(n^2)$
- **In Place:** Yes
- **Stable:** Yes

<https://www.youtube.com/watch?v=Jdtq5uKz-w4>



Selection sort

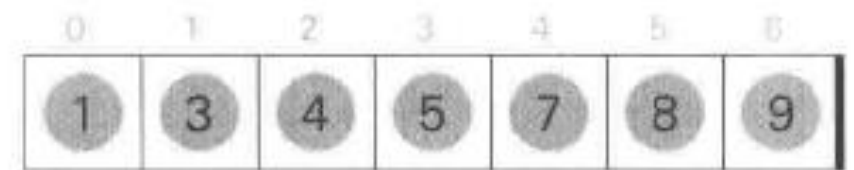
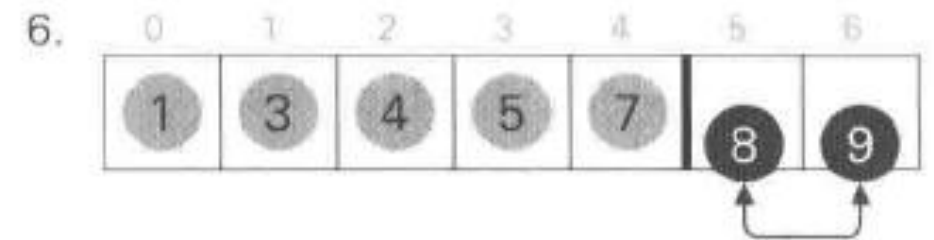
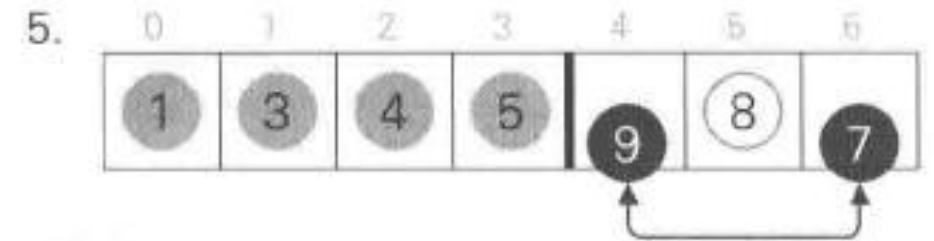
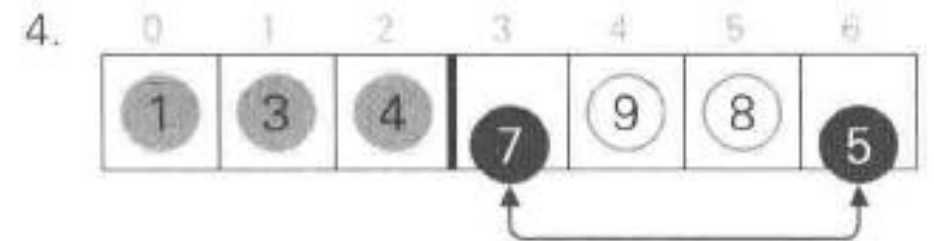
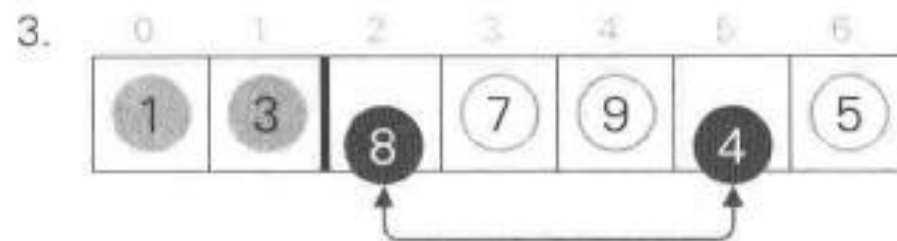
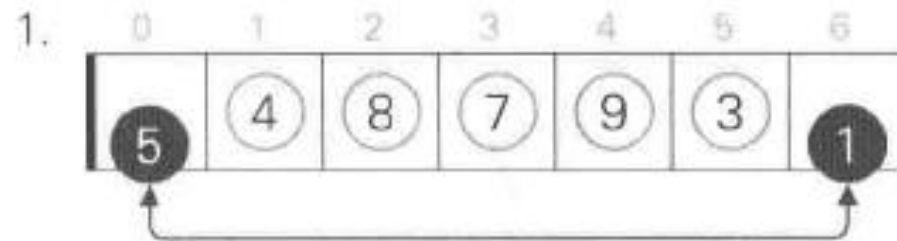
- The selection sort algorithm sorts an array by repeatedly finding the **minimum** element (considering ascending order) from unsorted part and putting it at the beginning.
- The algorithm maintains two subarrays in a given array.
 - The subarray which is already sorted.
 - Remaining subarray which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

```
selectionSort(A, N)
```

1. for i from 0 to N-1
2. minj = i
3. for j from i to N-1
4. if $A[j] < A[\text{minj}]$
5. minj = j
6. swap $A[i]$ and $A[\text{minj}]$

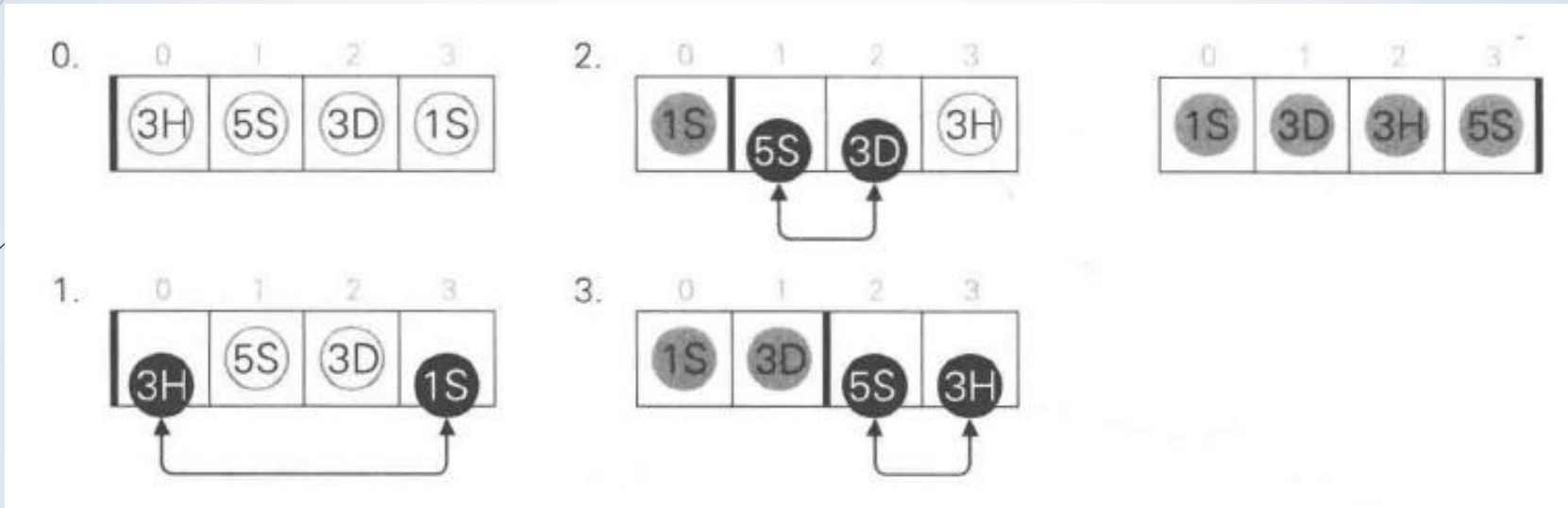
$A = \{5, 4, 8, 7, 9, 3, 1\}$



Another example: $A = \{3H, 5S, 3D, 1S\}$

There are four items. We sort based on the numbers.

3H is in front of 3D.



After sorting, 3D is in front of 3H.

Not stable



Selection Sort

- **Time Complexity**

- $O(n^2)$ as there are two nested loops.

- **Auxiliary Space:**

- $O(1)$

- The good thing about selection sort is that it never makes more than $O(n)$ swaps and can be useful **when memory write is a costly operation.**

- **Stability :**

- The default implementation is not stable. However it can be made stable.

- **In Place :**

- Yes, it does not require extra space.

Summary

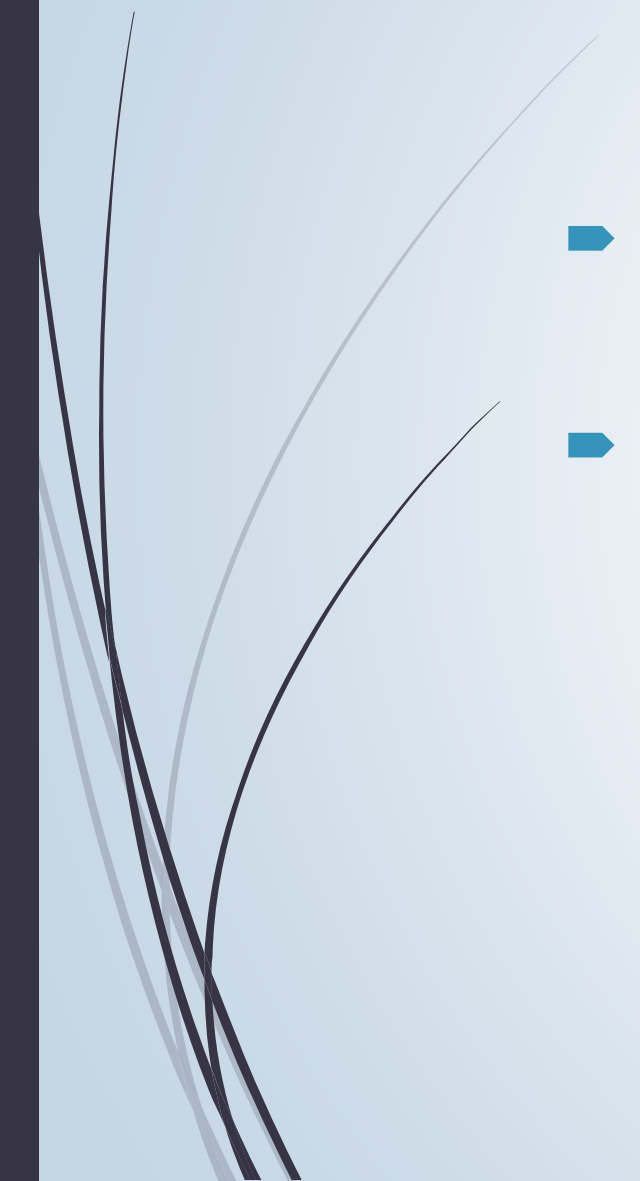
	Worst Case complexity	When to use	Stable	In-place
Insertion Sort	$O(n^2)$	when number of elements is small; when input array is almost sorted	Yes	Yes
Bubble Sort	$O(n^2)$	Easy to implement	Yes	Yes
Selection Sort	$O(n^2)$	when memory write is a costly operation.	No	Yes

Challenge Question

- The default selection sorting is not stable. Let's write a program to verify it.
- Suppose we are given a deck of cards. There are totally 36 cards of four suits {S, H, C, D} and 9 values {1, 2, 3, 4, 5, 6, 7, 8, 9}. For example, 'nine of heart' is represented by H9, and 'one of diamond' is represented by D1.
- Write a program to sort a given set of cards in ascending order by their values using Bubble sort and selection sort respectively. Then verify if the selection sort is stable for each given input.



Hint

- ▶ Use struct or class to represent each card
 - ▶ Remember that Bubble sort is stable.
- 



Merge sort



Merge Sort

- ➡ The process overall is thus:
 - ➡ Split the original list into two halves
 - ➡ Sort each half (using merge sort)
 - ➡ Merge the two sorted halves together into a single sorted list

https://www.youtube.com/watch?v=3aTfQvs-_hA

Example

34 56 78 12 45 3 99 23

34 56 78 12 | 45 3 99 23

34 56 | 78 12 | 45 3 | 99 23

34 | 56 | 78 | 12 | 45 | 3 | 99 | 23

34 56 | 12 78 | 3 45 | 23 99

12 34 56 78 | 3 23 45 99

3 12 23 34 45 56 78 99

Algorithm for Merge Sort

```
procedure mergesort (first, last, array)
    mid= (first + last) / 2
    mergesort (first, mid, array)
    mergesort (mid+1, last, array)
    merge_two_halves (mid, array)
end mergesort
```


Merge two sorted lists

L: 12 34 56 78

R: 3 23 45 99

3

L: 12 34 56 78

R: 3 23 45 99

12

L: 12 34 56 78

R: 23 45 99

23

L: 34 56 78

R: 23 45 99

34

L: 34 56 78

R: 45 99

45

L: 56 78

R: 45 99

56

L: 56 78

R: 99

78

L: 78

R: 99

99

R: 99

Merge two sorted lists

L: 1, 4, 5, 10

R: 2, 12, 18, 27

1 2 4 5 10 12, 18, 27

Time $= \Theta(n)$ to merge a total of n elements (linear time)

Analyzing merge sort

MERGE-SORT $A[1 \dots n]$

1. If $n = 1$, done.
2. Recursively sort $A[1 \dots \lfloor n/2 \rfloor]$
and $A[\lfloor n/2 \rfloor + 1 \dots n]$.
3. **"Merge"** the two sorted lists

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n), & \text{if } n > 1 \end{cases}$$

$T(n)$

$\Theta(1)$

$T(n/2)$

$T(n/2)$

$\Theta(n)$

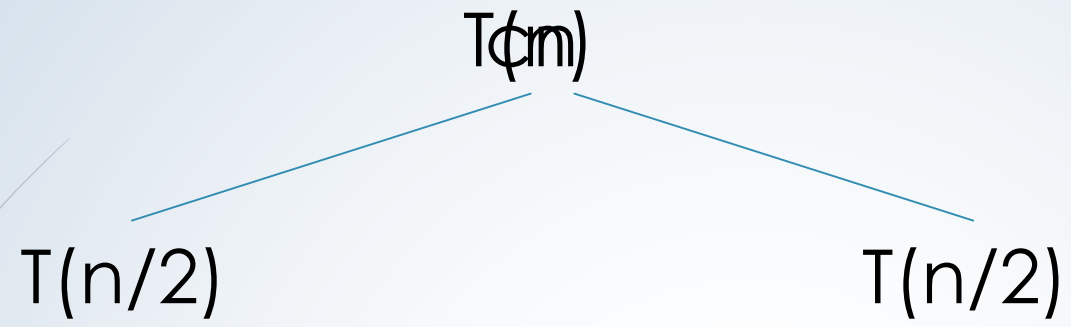
$T(n) = ?$



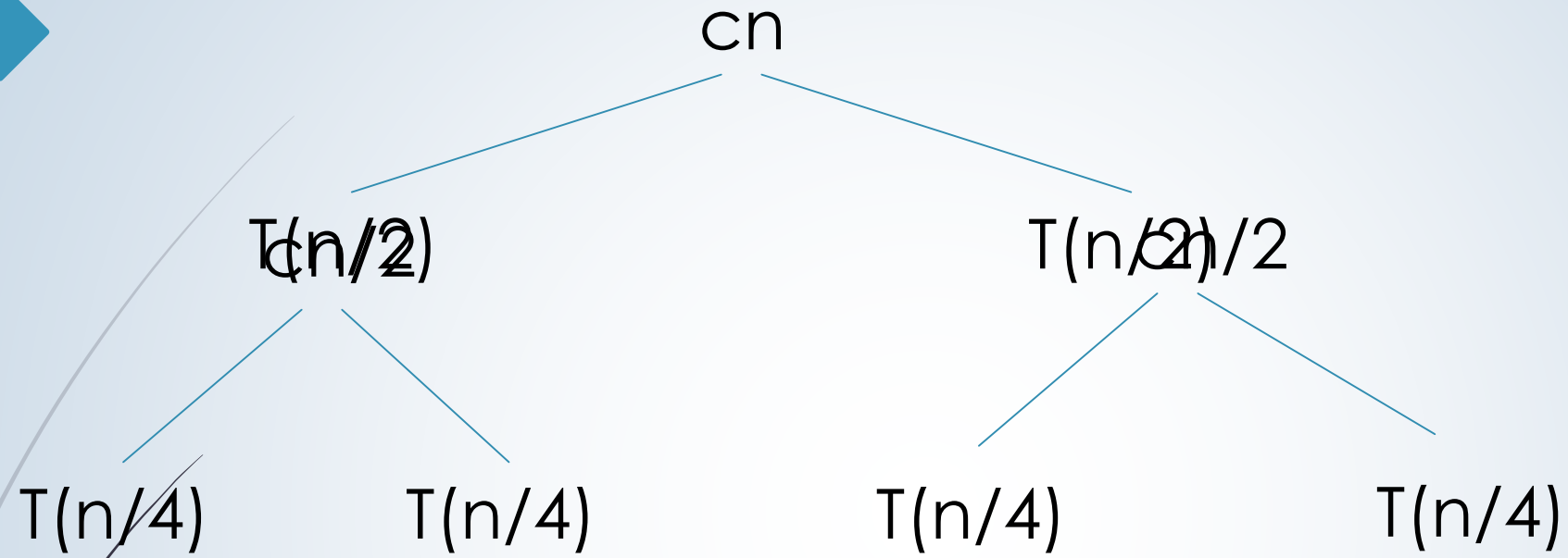
Recurrence solving

- Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.
- We can use *recursion tree*.

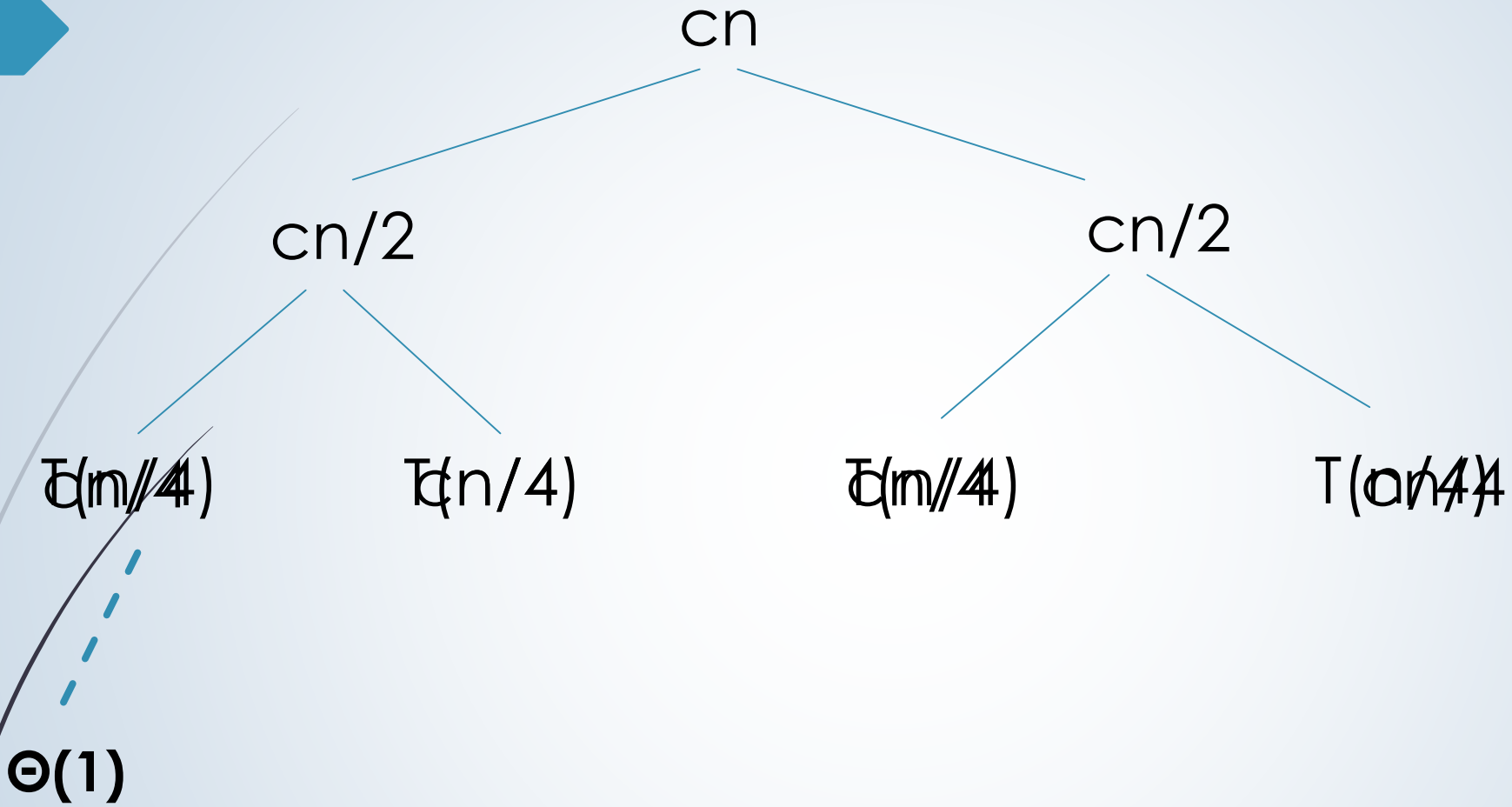
$$T(n) = 2T(n/2) + cn$$



$$T(n) = 2T(n/2) + cn$$



$$T(n) = 2T(n/2) + cn$$



$$T(n) = 2T(n/2) + cn$$

$$h = 1 + \lg n$$

$$\Theta(1)$$

$$cn$$

$$cn$$

$$cn$$

$$\Theta(n)$$



Merge Sort

- Merge sort is a more efficient sorting algorithm than either *selection sort* or *bubble sort*.
- Complexity: **$\Theta(n \log n)$** in all 3 cases
$$T(N) = 2T(N/2) + \Theta(N)$$
- The basic idea is that if you know you have two sorted lists, you can combine them into a single large sorted list by just looking at the first item in each list. Whichever is smaller is moved to the single list being assembled. There is then a new first item in the list from which the item was moved, and the process repeats.



Merge Sort

- **Auxiliary Space**

- $O(n)$

- **Sorting In Place**

- No in a typical implementation


- **Stable**

- Yes

- **Applications of Merge Sort**

- Merge Sort is useful for sorting linked lists in $O(n \log n)$ time.

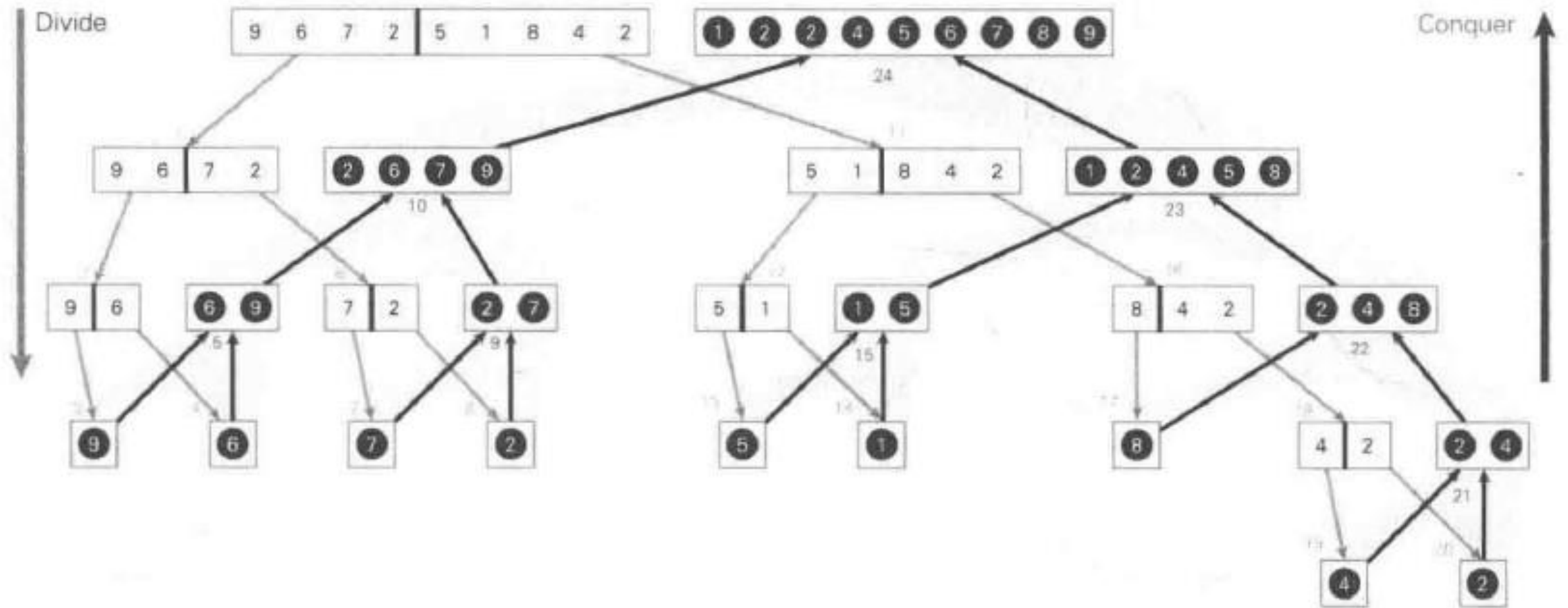
- Used in External Sorting



```
Merge(A, left, mid, right)
  n1 = mid - left;
  n2 = right - mid;
  create array L[0...n1], R[0...n2]
  for i = 0 to n1-1
    do L[i] = A[left + i]
  for i = 0 to n2-1
    do R[i] = A[mid + i]
  L[n1] = SENTINEL
  R[n2] = SENTINEL
  i = 0;
  j = 0;
  for k = left to right-1
    if L[i] <= R[j]
      then A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
      j = j + 1
```

```
Merge-Sort(A, left, right){
  if left+1 < right
    then mid = (left + right)/2;
      call Merge-Sort(A, left, mid)
      call Merge-Sort(A, mid, right)
    call Merge(A, left, mid, right)
```

Sort {9,6,7,2,5,1,8,4,2}



A decorative graphic on the left side of the slide. It features a solid blue arrow pointing to the right, positioned horizontally. Behind the arrow and extending downwards and to the right are several thin, curved black lines that create a sense of motion or flow.

Quick sort

Quick Sort

- Quick Sort is a very efficient sorting algorithm invented by **Sir Charles Antony Richard Hoare** when he was 26.
- Complexity
 - Average $O(n \log n)$
 - Worst case $O(n^2)$
- It has two phases:
 - the partition phase and
 - the sort phase.



Quick Sort

- In each step of quick sort a range of an array is sorted.
 - The last element in the range is used as the **pivot value**.
 - The rest of the values in the range are grouped into two partitions: one containing the values larger than the pivot value and the other containing the values smaller than the pivot value.
 - Each of the partitions is sorted by a recursive call of quick sort.

Always pick first element as pivot.

Always pick last element as pivot (implemented below)

Pick a random element as pivot.

Pick median as pivot.

Partition(A, p, r)

1 $x = A[r]$

2 $i = p-1$

3 for $j = p$ to $r-1$

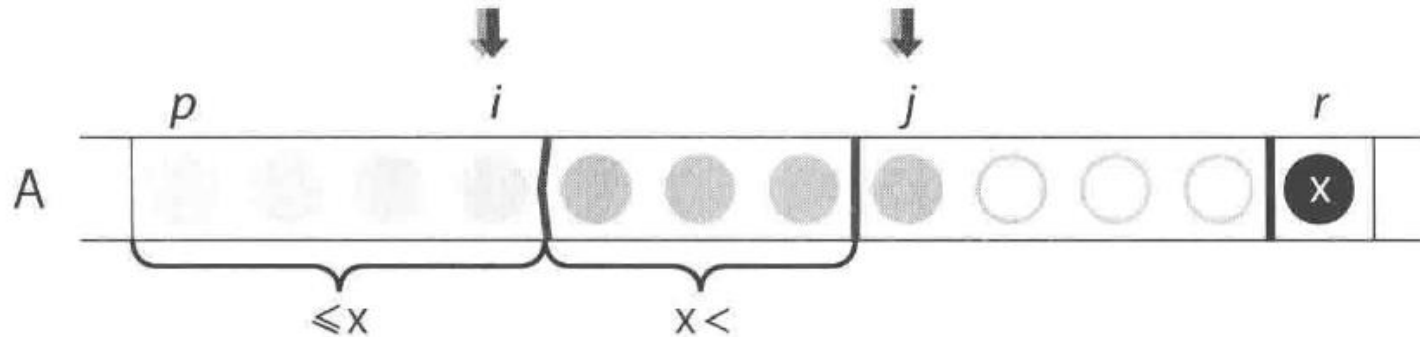
4 do if $A[j] \leq x$

5 then $i = i+1$

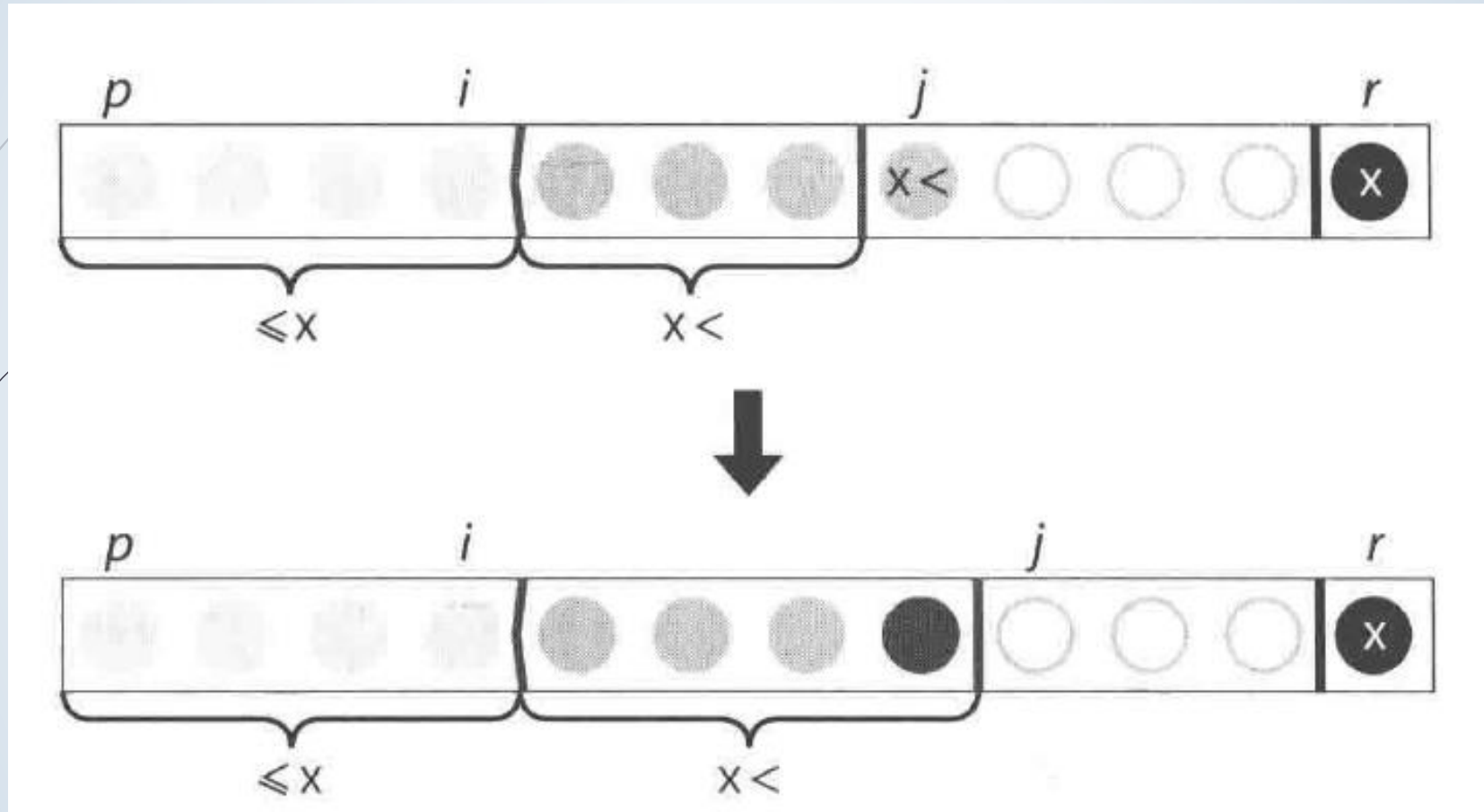
6 exchange $A[i]$ and $A[j]$

7 exchange $A[i+1]$ and $A[r]$

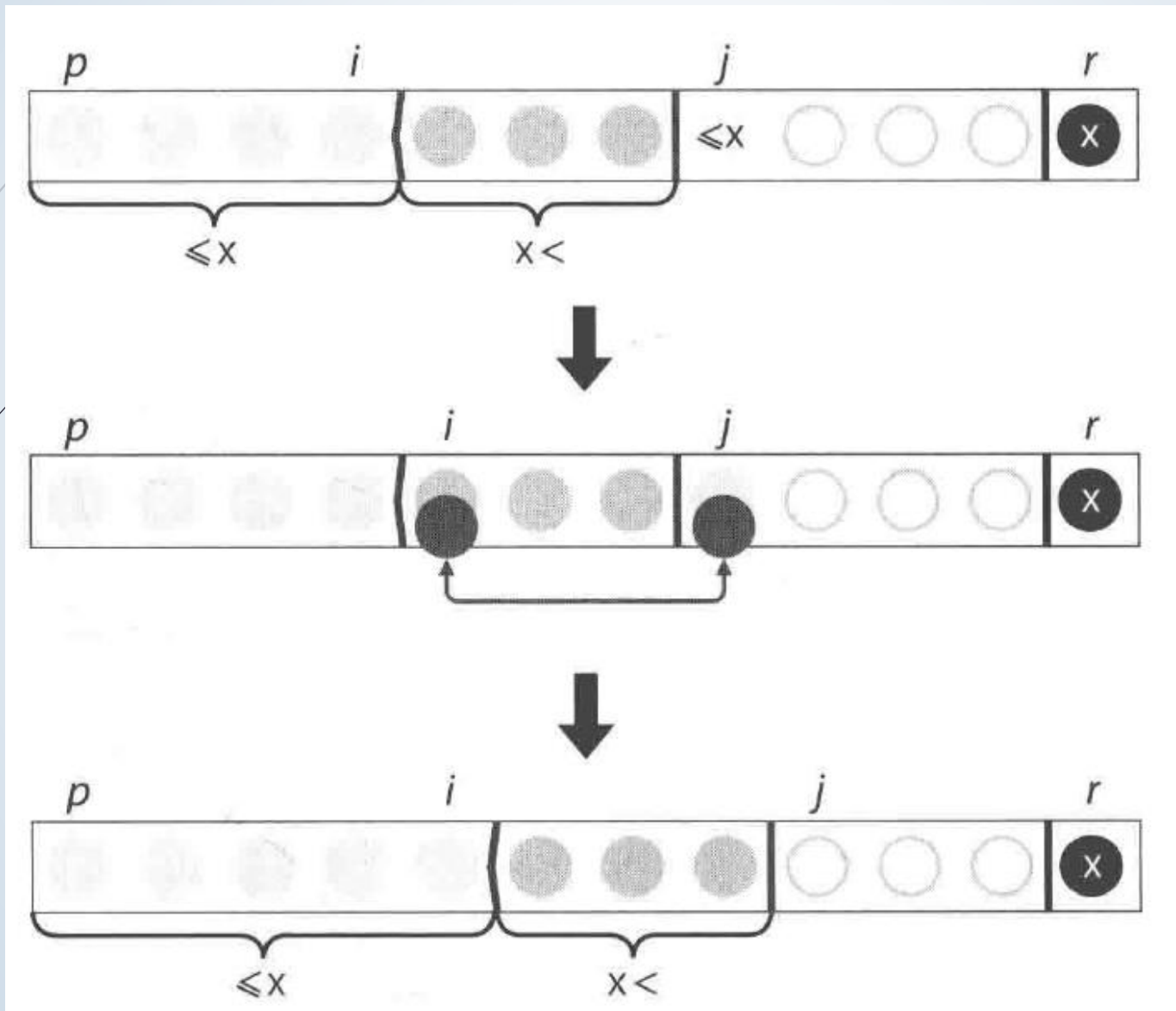
8 return $i+1$

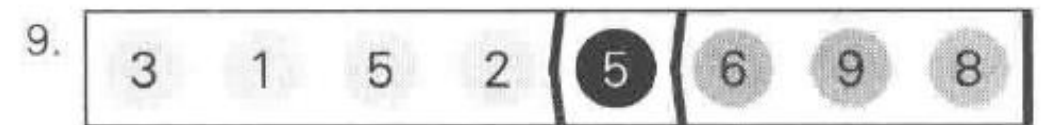
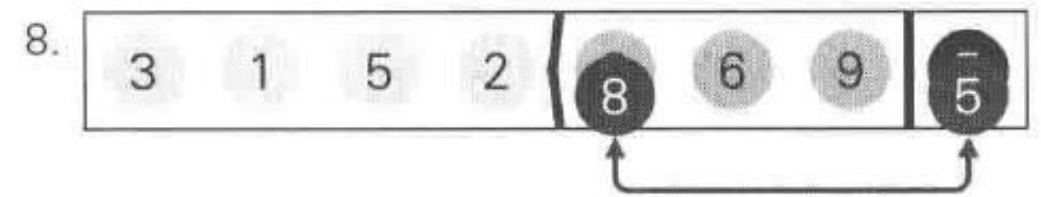
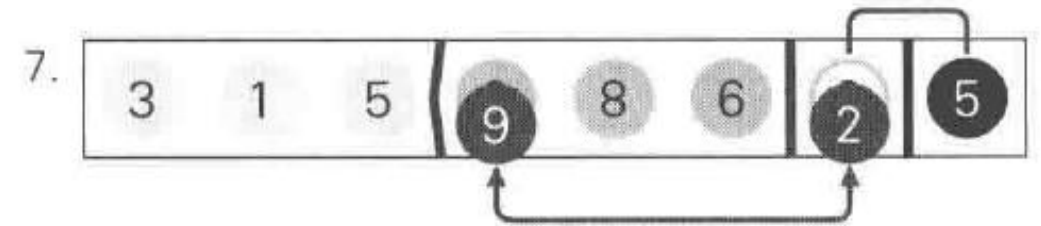
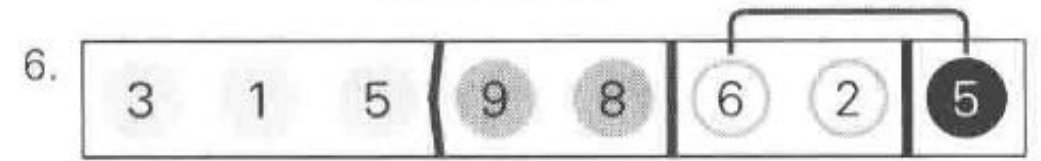
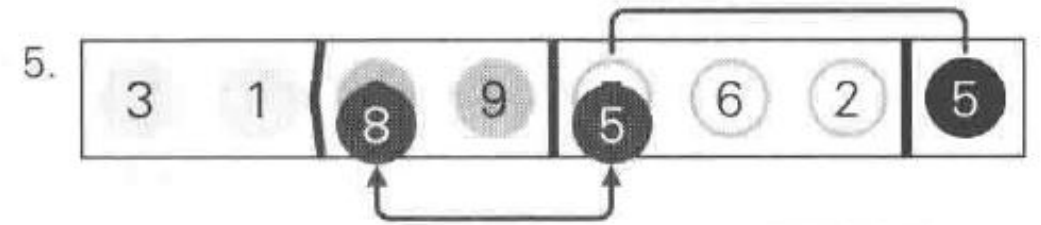
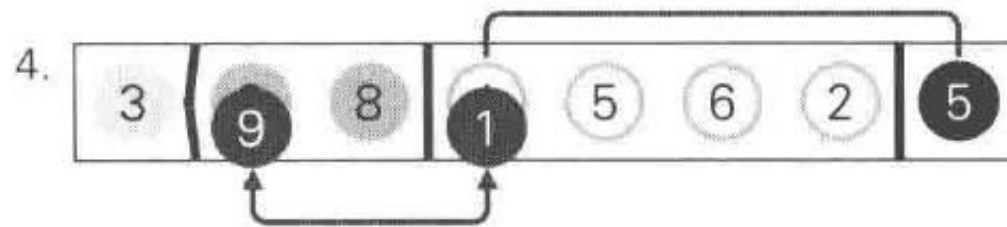
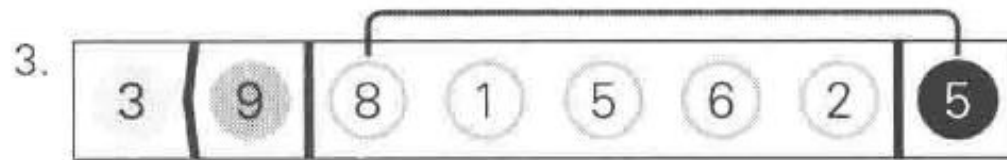
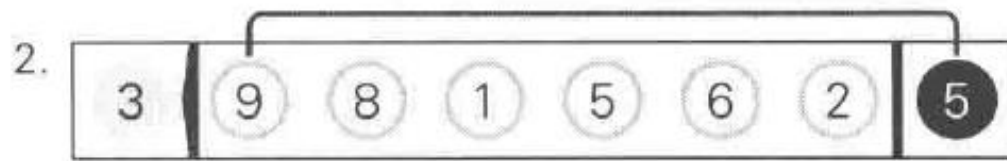
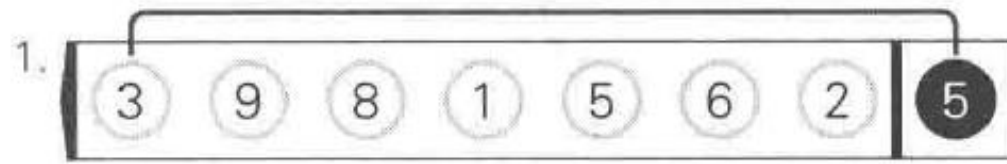


Case 1: $A[j] > x$



Case 2: $A[j] \leq x$





Quick Sort

- Quick sort partitions the array into two sections, the first of “small” elements and the second of “large” elements. It then sorts the small and large elements separately.
- Ideally, partitioning would use the *median* of the given values, but the median can only be found by scanning the whole array and this would slow the algorithm down. In that case the two partitions would be of equal size; In the simplest versions of quick sort an arbitrary element, typically the first/last element is used as an estimate (guess) of the median.

Quick Sort

```
Partition(A, p, r)
```

```
1 x = A[r]
```

```
2 i = p-1
```

```
3 for j = p to r-1
```

```
4     do if A[j] <= x
```

```
5         then i = i+1
```

```
6             exchange A[i] and A[j]
```

```
7 exchange A[i+1] and A[r]
```

```
8 return i+1
```

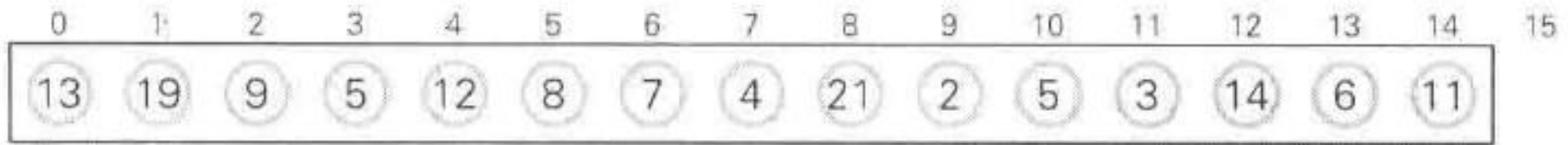
```
Quicksort(A, p, r)
```

```
1 if p < r
```

```
2     then q = Partition(A, p, r)
```

```
3         run Quicksort(A, p, q-1)
```

```
4         run Quicksort(A, q+1, r)
```

1

partition(A, 0, 14)



2

partition(A, 0, 8)

7

partition(A, 10, 14)



3

partition(A, 0, 4)

6

partition(A, 6, 8)



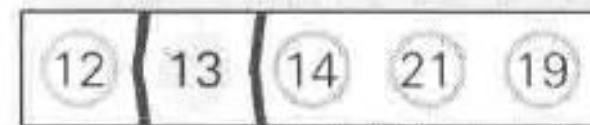
4

partition(A, 2, 4)



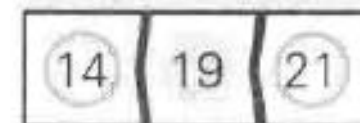
5

partition(A, 3, 4)



8

partition(A, 12, 14)



Quick Sort complexity (1)

- **Worst Case:** The worst case occurs when the partition process always picks greatest or smallest element as pivot.

$$T(n) = T(0) + T(n-1) + \theta(n)$$

which is equivalent to

$$T(n) = T(n-1) + \theta(n)$$

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

Quick Sort complexity (2)

- ➡ **Best Case:** The best case occurs when the partition process always picks the middle element as pivot.

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

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

Quick Sort complexity (3)

- 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

$$T(n) = T(n/9) + T(9n/10) + \theta(n)$$

Solution of above recurrence is also $O(n \log n)$

Is Quick Sort really quick?

- 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.

Quick Sort

➔ Is QuickSort stable?

The default implementation is not stable. However any sorting algorithm can be made stable by considering indexes as comparison parameter.

➔ Is QuickSort In-place?

As per the broad definition of in-place algorithm it qualifies as an in-place sorting algorithm as it uses extra space only for storing recursive function calls but not for manipulating the input.

Comparison based sorting

- ➔ A sorting algorithm is compared-based if an operation of comparing two keys can be preformed on a given list of data elements having keys.
- ➔ To determine the relative order of given two elements a_i and a_j , we perform one of the following tests:

$$a_i < a_j, a_i > a_j, a_i \leq a_j, a_i \geq a_j, a_i = a_j$$

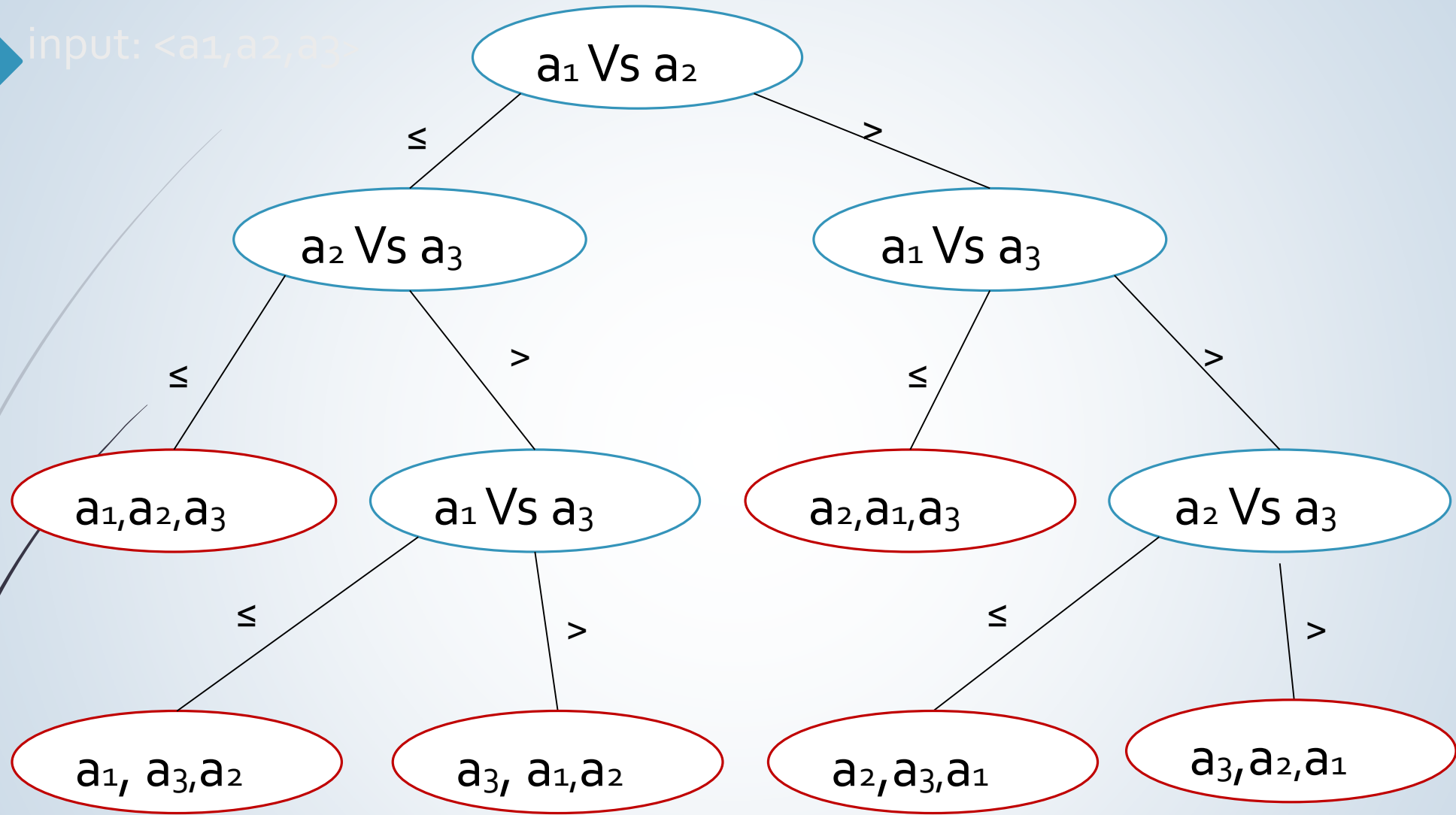
Algorithm	Worst-case	Average-case	Best-case	In-place
Insertion sort	$O(n^2)$	$O(n^2)$	$O(n)$	√
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	X
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	√
Quick sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	√
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n)$	√



Lower Bounds for sorting

- No matter what the algorithm might be , there is some input which will cause it to run in $\Omega(n \log n)$ time.
- Decision tree model
 - The decision tree is a fully binary tree.
 - Each node corresponds to one of the comparisons in the algorithm.
 - The sorting algorithm starts at the root node.
 - The whole process is repeated until a leaf is encountered.
 - Each leaf represents one (correct) ordering of the input.

input: $\langle a_1, a_2, a_3 \rangle$



The decision tree model for an insertion sort.



$\Omega(n \log n)$ Lower Bound

Two properties

- there must be $n!$ permutation leaves, one corresponding for each possible ordering of n elements.
- length (number of edges) of longest path in decision tree (its height) is either equal to the worst case number of comparisons or less than or equal to the worst-case number of operations of algorithm (lower bound on time).

Theorem

- Any decision tree for sorting n elements has height $\Omega(n \log n)$.

A decorative graphic on the left side of the slide. It features a solid blue arrow pointing to the right, positioned horizontally. Behind the arrow and extending downwards and to the right are several thin, curved black lines that create a sense of motion or flow.

Non-comparison based sorting



Counting Sort (1/2)

- Comparison-based sorting algorithms require $\Omega(n \log n)$ time
- But we can sort in $O(n)$ time using more powerful operations
 - When elements are integers in $\{0, \dots, M-1\}$, bucket sort needs $O(M+n)$ time and $O(M)$ space
 - When $M=O(n)$, counting sort needs $O(2n)=O(n)$ time

Counting Sort (2/2)

- Idea: Require a counter (auxiliary) array $C[0..M-1]$ to count the number of occurrences of each integer in $\{0, \dots, M-1\}$
- Algorithm:
 - **Step 1:** initialize all entries in $C[0..M-1]$ to 0
 - **Step 2:** For $i=0$ to $n-1$
 - Use $A[i]$ as an array index and increase $C[A[i]]$ by one
 - **Step 3:** For $j=0$ to $M-1$
 - Write $C[j]$ copies of value j into appropriate places in $A[0..n-1]$

<https://www.geeksforgeeks.org/python-program-for-counting-sort/?ref=lbp>

Counting Sort (Example)

- Input: 3, 4, 6, 9, 4, 3 where $M=10$

Counter array A:

0	1	2	3	4	5	6	7	8	9

- Step 1: Initialization

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

- Step 2: Read 3

($A[3] = A[3] + 1$)

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

• Read 4

($A[4] = A[4] + 1$)

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

• Read 6

($A[6] = A[6] + 1$)

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

• Read 9

($A[9] = A[9] + 1$)

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

• Read 4

($A[4] = A[4] + 1$)

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

• Read 3

($A[3] = A[3] + 1$)

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

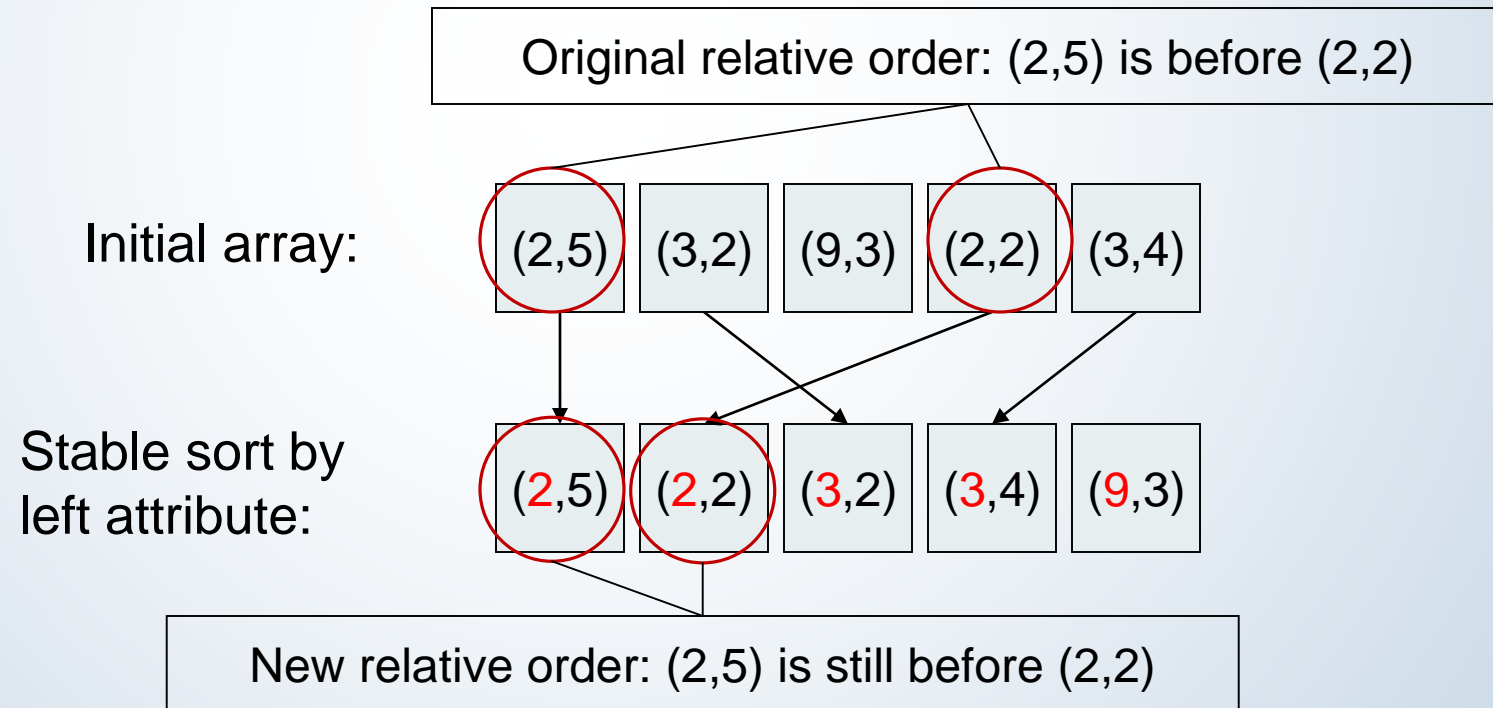
Bucket Sort

- Set up a list of empty buckets. A bucket is initialized for each element in the array.
- Iterate through the bucket list and insert elements from the array. Where each element is inserted depends on the input list and the largest element of it. We can end up with 0..n elements in each bucket.
- Sort each non-empty bucket. You can do this with any sorting algorithm. If we're working with a small dataset, each bucket won't have many elements then Insertion Sort works wonders for us here.
- Visit the buckets in order. Once the contents of each bucket are sorted, when concatenated, they will yield a list in which the elements are arranged based on your criteria.

<https://stackabuse.com/bucket-sort-in-python/>

Review: Stable Sort

- Definition: A *stable* sorting algorithm is one that preserves the original relative order of elements with equal key
- E.g., suppose the left attribute is the key attribute



Using Stable Sort (1/4)

- Suppose we sort some 2-digit integers
- **Phase 1: Stable sort by the right digit (the least significant digit)**

Initial array:

25	32	93	22	34
----	----	----	----	----

Sort by
right digit:

3 <u>2</u>	2 <u>2</u>	9 <u>3</u>	3 <u>4</u>	2 <u>5</u>
------------	------------	------------	------------	------------

Using Stable Sort (2/4)

- Suppose we sort some 2-digit integers
- **Phase 2: Stable sort by the left digit (the second least significant digit)**

Initial array:

25	32	93	22	34
----	----	----	----	----

Sort by
right digit:

32	22	93	34	25
----	----	----	----	----

Stable sort by
left digit:

<u>22</u>	<u>25</u>			
-----------	-----------	--	--	--

Using Stable Sort (3/4)

- Suppose we sort some 2-digit integers
- **Phase 2: Stable sort by the left digit (the second least significant digit)**

Initial array:

25	32	93	22	34
----	----	----	----	----

Sort by
right digit:

32	22	93	34	25
----	----	----	----	----

Stable sort by
left digit:

<u>2</u> 2	<u>2</u> 5	<u>3</u> 2	<u>3</u> 4	
------------	------------	------------	------------	--

Using Stable Sort (4/4)

- Suppose we sort some 2-digit integers
- **Phase 2: Stable sort by the left digit (the second least significant digit)**

Initial array:

25	32	93	22	34
----	----	----	----	----

Sort by
right digit:

32	22	93	34	25
----	----	----	----	----

Stable sort by
left digit:

<u>2</u> 2	<u>2</u> 5	<u>3</u> 2	<u>3</u> 4	<u>9</u> 3
------------	------------	------------	------------	------------



Radix Sort

- ▶ Bucket sort is not efficient if M is large
- ▶ The idea of radix sort:
 - ▶ Apply stable bucket sort on each digit (from **Least** Significant Digit to **Most** Significant Digit)
- ▶ A complication:
 - ▶ Just keeping the count is not enough
 - ▶ Need to keep the actual elements
 - ▶ Use a queue for each digit

<https://www.geeksforgeeks.org/python-program-for-radix-sort/>

Radix Sort (Example) (1/3)

- Input: 170, 045, 075, 090, 002, 024, 802, 066
- The **first** pass
 - Consider **the least significant digits** as keys and move the keys into their buckets

0	17 <u>0</u> , 09 <u>0</u>
1	
2	00 <u>2</u> , 80 <u>2</u>
3	
4	02 <u>4</u>
5	04 <u>5</u> , 07 <u>5</u>
6	06 <u>6</u>
7	
8	
9	

- Output: 170, 090, 002, 802, 024, 045, 075, 066

Radix Sort (Example) (2/3)

- The **second** pass
- Input: 170, 090, 002, 802, 024, 045, 075, 066
 - ➡ Consider **the second least significant digits** as keys and move the keys into their buckets

0	0 <u>0</u> 2, 8 <u>0</u> 2
1	
2	0 <u>2</u> 4
3	
4	0 <u>4</u> 5
5	
6	0 <u>6</u> 6
7	1 <u>7</u> 0, 0 <u>7</u> 5
8	
9	0 <u>9</u> 0

- ➡ Output: 002, 802, 024, 045, 066, 170, 075, 090

Radix Sort (Example) (3/3)

- The **third** pass
- Input: 002, 802, 024, 045, 066, 170, 075, 090
 - ➡ Consider **the third least significant digits** as keys and move the keys into their buckets

0	<u>0</u> 02, <u>0</u> 24, <u>0</u> 45, <u>0</u> 66, <u>0</u> 75, <u>0</u> 90
1	<u>1</u> 70
2	
3	
4	
5	
6	
7	
8	<u>8</u> 02
9	

- ➡ Output: 002, 024, 045, 066, 075, 090, 170, 802 (Sorted)

Worst-case Time Complexity

- Assume d digits, each digit comes from $\{0, \dots, M-1\}$
- For each digit,
 - $O(M)$ time to initialize M queues,
 - $O(n)$ time to distribute n numbers into M queues
- Total time = $O(d(M+n))$
- When d is constant and $M = O(n)$, we can make radix sort run in linear time, i.e., $O(n)$.

Non-comparison based sorts

- Sort by means other than comparing two elements.
 - Counting sort: input elements are in a range of $1, 2, 3, \dots, k$. Use array indexing to count the number of elements of each value.
 - Radix sort: each integer consists of d digits, and each digit is in the range of $1, 2, \dots, m$.
 - Bucket sort: require advanced knowledge of input distribution.

Algorithm	Worst case	Average case	Best case	In-place
Counting	$O(n+k)$	$O(n+k)$	$O(n+k)$	X
Radix sort	$O(d(n+m))$	$O(d(n+m))$	$O(d(n+m))$	X
Bucket sort	$O(n^2)$	$O(n)$	$O(n)$	X

The std::sort() Function in C++

- ▶ The std::sort() function in C++ is a built-in function that is used to sort any form of data structure in a particular order. It is defined in the algorithm header file. The sort() function prototype is given below.

```
void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

- ▶ The function does not return anything. It just updates the elements/items from the first up to the last iterables or positions. The third parameter(optional) comp has to be a function that determines the order in which the elements are going to be sorted. When not specified, the sorting takes place in ascending order considering it to be the std::less<int>() function by default.

Sort() in STL

```
1  #include<iostream>
2  #include<algorithm>
3  using namespace std;
4
5  int main() {
6      int n, v[5];
7
8      for ( int i = 0; i < 5; i++ ) cin >> v[i];
9
10     sort(v, v + 5);
11
12     for ( int i = 0; i < 5; i++ ) {
13         cout << v[i] << " ";
14     }
15     cout << endl;
16
17     return 0;
18 }
```



Sort Algorithm



- In C++ STL, we have a sort function which can sort in increasing and decreasing order. Not only integral but you can sort user-defined data too using this function.
- Internally it uses IntroSort, which is a combination of QuickSort, HeapSort and InsertionSort. (This is important as it occurs internally and no one knows about it.)
- By default, it uses QuickSort, but if QuickSort is doing unfair partitioning and taking more than $N \cdot \log N$ time, it switches to HeapSort. When the array size becomes very small, it switches to InsertionSort. (This conversion and checking occurs internally and is not very popular to people)

More sorting functions in Algorithm

Sorting:

sort	Sort elements in range (function template)
stable_sort	Sort elements preserving order of equivalents (function template)
partial_sort	Partially sort elements in range (function template)
partial_sort_copy	Copy and partially sort range (function template)
is_sorted <small>C++11</small>	Check whether range is sorted (function template)
is_sorted_until <small>C++11</small>	Find first unsorted element in range (function template)
nth_element	Sort element in range (function template)

Partitions:

is_partitioned <small>C++11</small>	Test whether range is partitioned (function template)
partition	Partition range in two (function template)
stable_partition	Partition range in two - stable ordering (function template)
partition_copy <small>C++11</small>	Partition range into two (function template)
partition_point <small>C++11</small>	Get partition point (function template)