CYDEO

Data Structures & Algorithms

Day 10

Sorting Algorithms

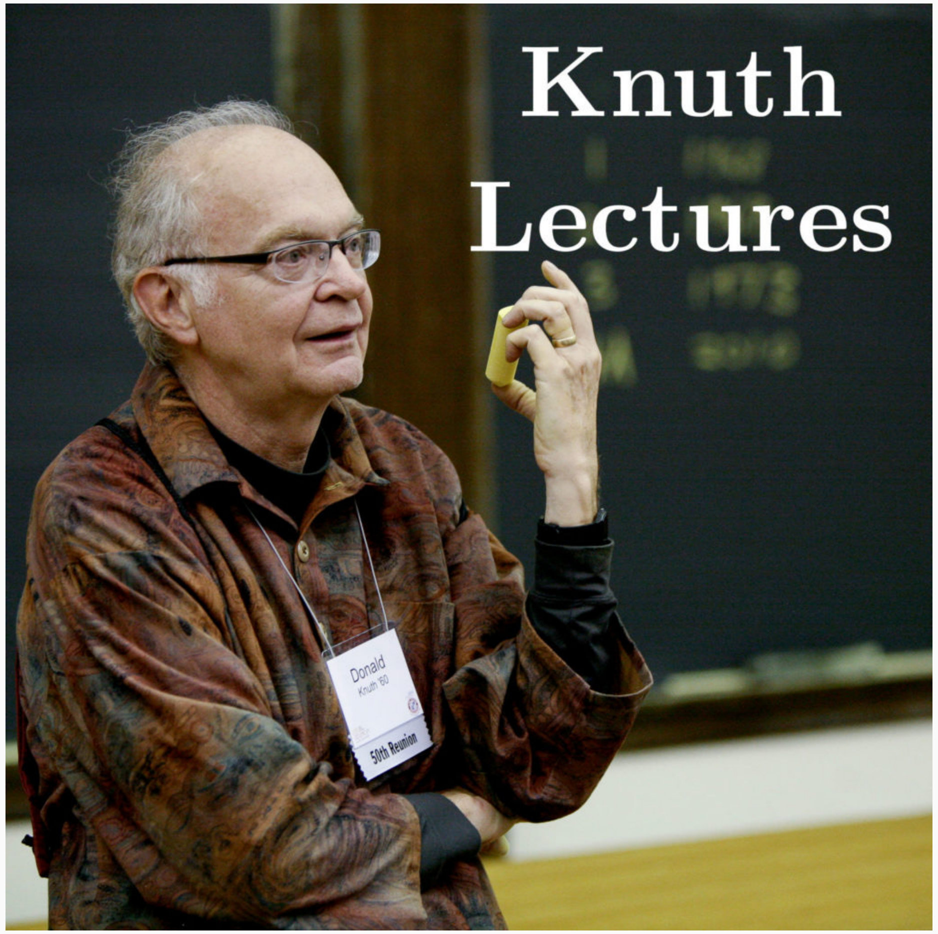
Today's Content

- Intro into Sorting Algorithms
- LRU Cache Problem

Sorting Problem

- *“Computer manufacturers of the 1960s estimated that more than 25% of the running time on their computers was spent on sorting... in fact, there were many installations in which the task of sorting was responsible for more than half of the computing time.”*

The Art of Computer Programming, Vol 3.
by **Donald Knuth**,



Algorithm Design Strategies & Sorting Algorithms

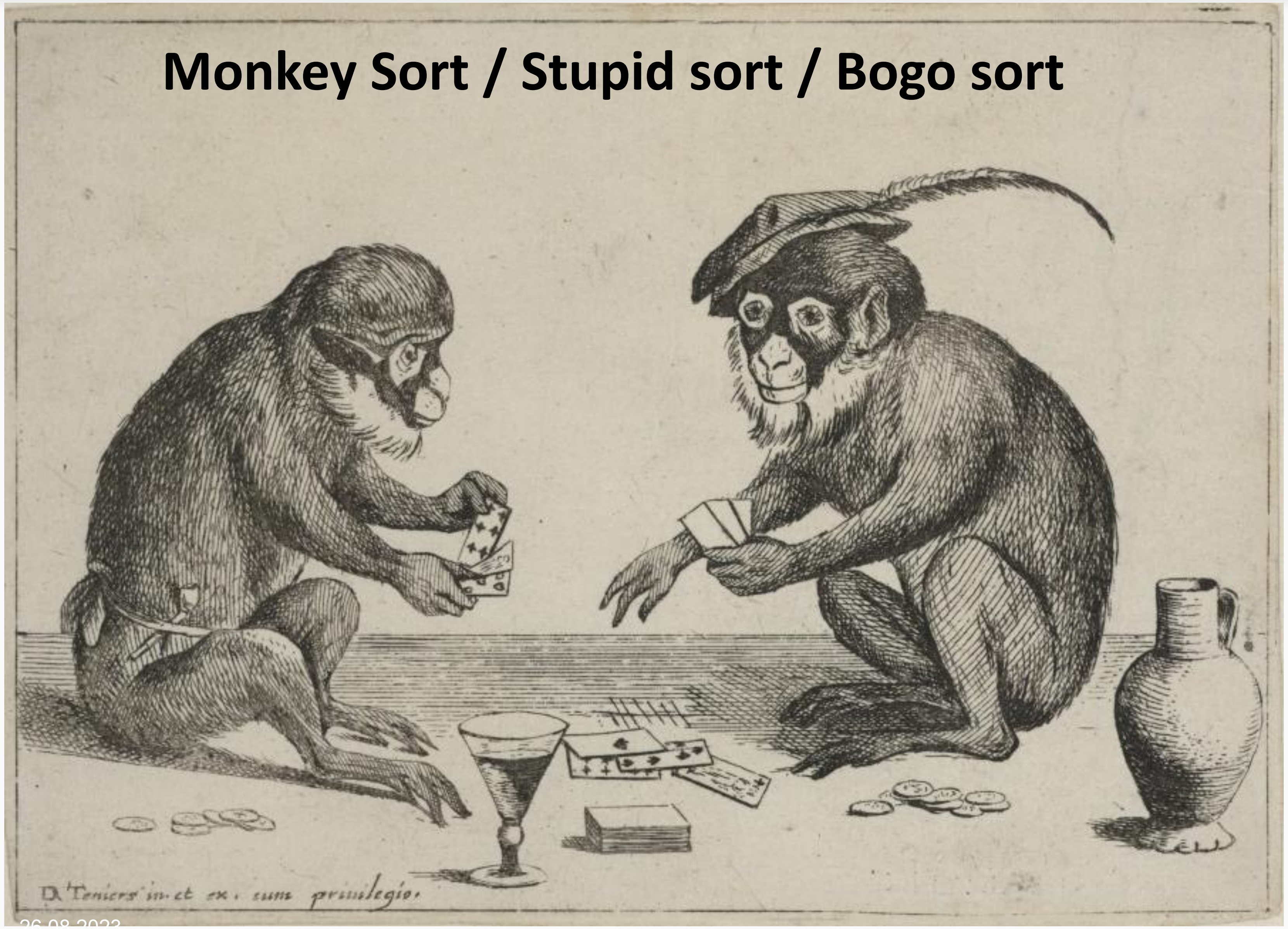
1. Brute force (*Monkey Sort, Selection Sort, Bubble Sort*)
2. Decrease and Conquer (*Insertion Sort*)
3. Divide and Conquer (*Merge Sort, Quick Sort*)
4. Transform and Conquer (*Heap Sort, Tree Sort*)
5. Linear time sorting (*Bucket Sort, Radix Sort*)

Design Strategy 1 – Brute Force

- Simplest design strategy
- The most straightforward approach, usually based on the problem statement.

Design Strategy 1 – Brute Force : Monkey Sort

Monkey Sort / Stupid sort / Bogo sort



```
while (not sorted) {
        shuffle(list)
}
```

Design Strategy 1 – Brute Force : Monkey Sort

Monkey Sort

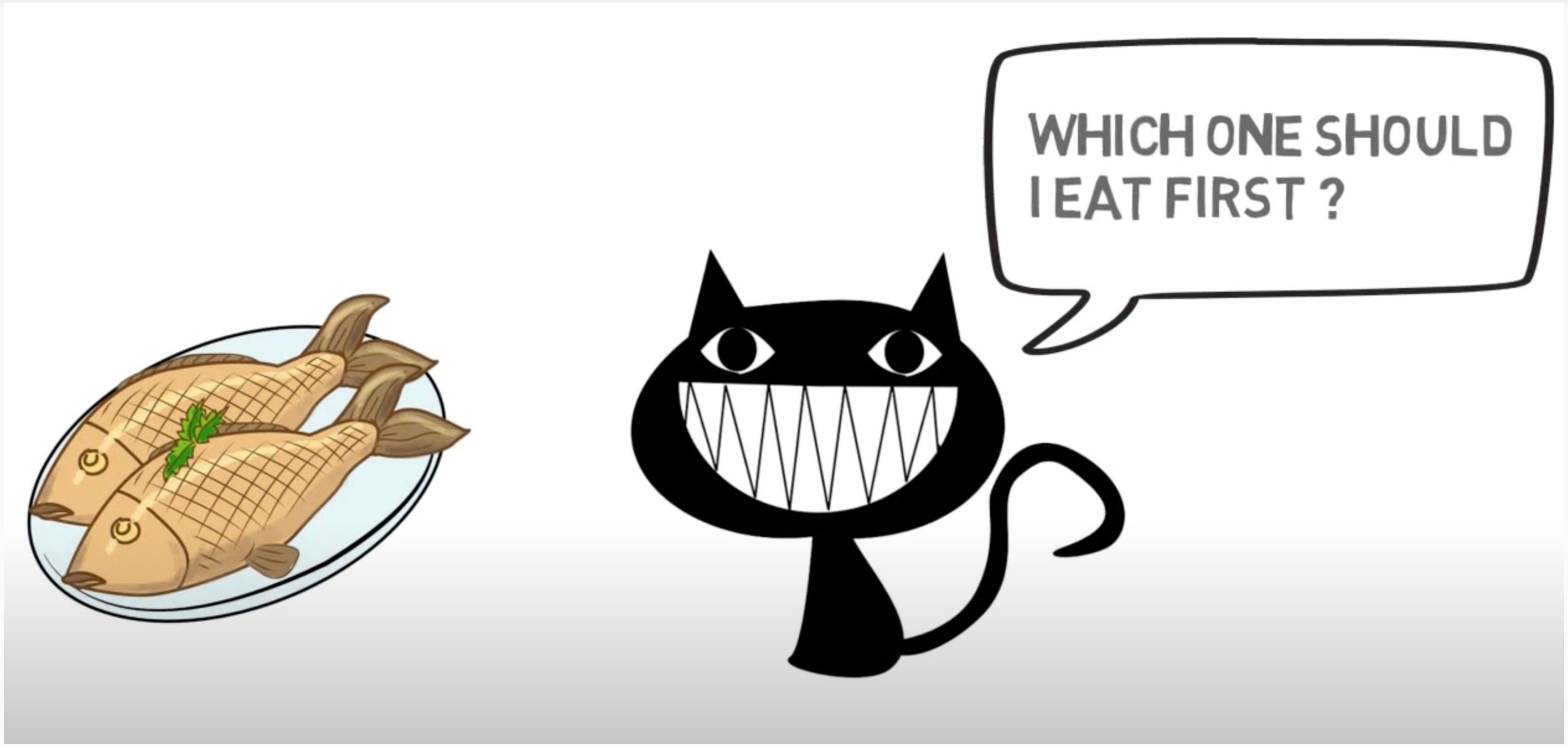
- The algorithm appears to be correct.
- But even if we had a machine that could run *10^8 operations per second*, and even if we could generate and test one permutation in a single operation, we would need almost *800 years for an array of size 20.*
- For an *array of size 100*, it would take *10^{142} years*.

Design Strategy 1 – Brute Force : Any better algorithm?

Brute Force Sorting Algorithms better than Monkey Sort:

- Selection Sort
- Bubble Sort

Brute Force - Selection Sort



Brute Force - Selection Sort

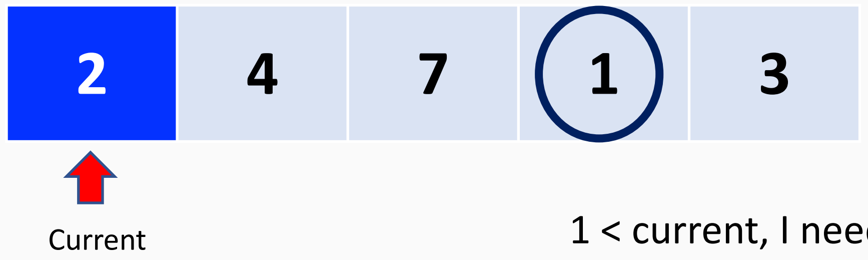
- The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning.
- The algorithm maintains two parts in a given array.

    1) The part which is sorted.
    2) Remaining part which is unsorted.

- In every iteration of selection sort, the minimum from the unsorted part is picked up and moved to the sorted part.

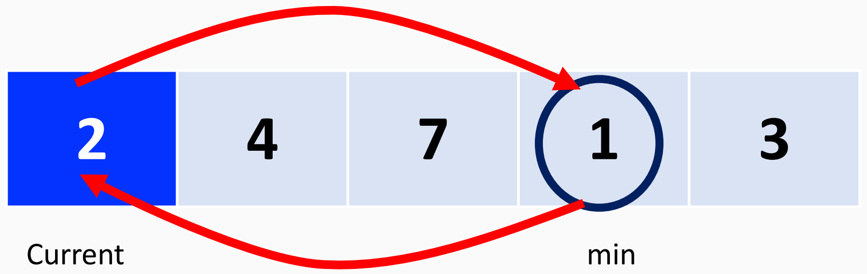
Selection Sort-How it works

Step 1: Set first index as min,
Then try to find value less than current.

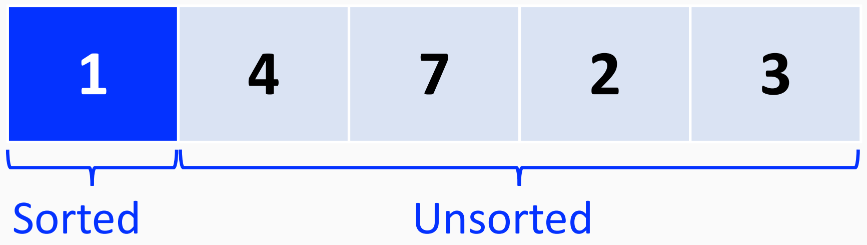


1 < current, I need to 2 and 1 swap them

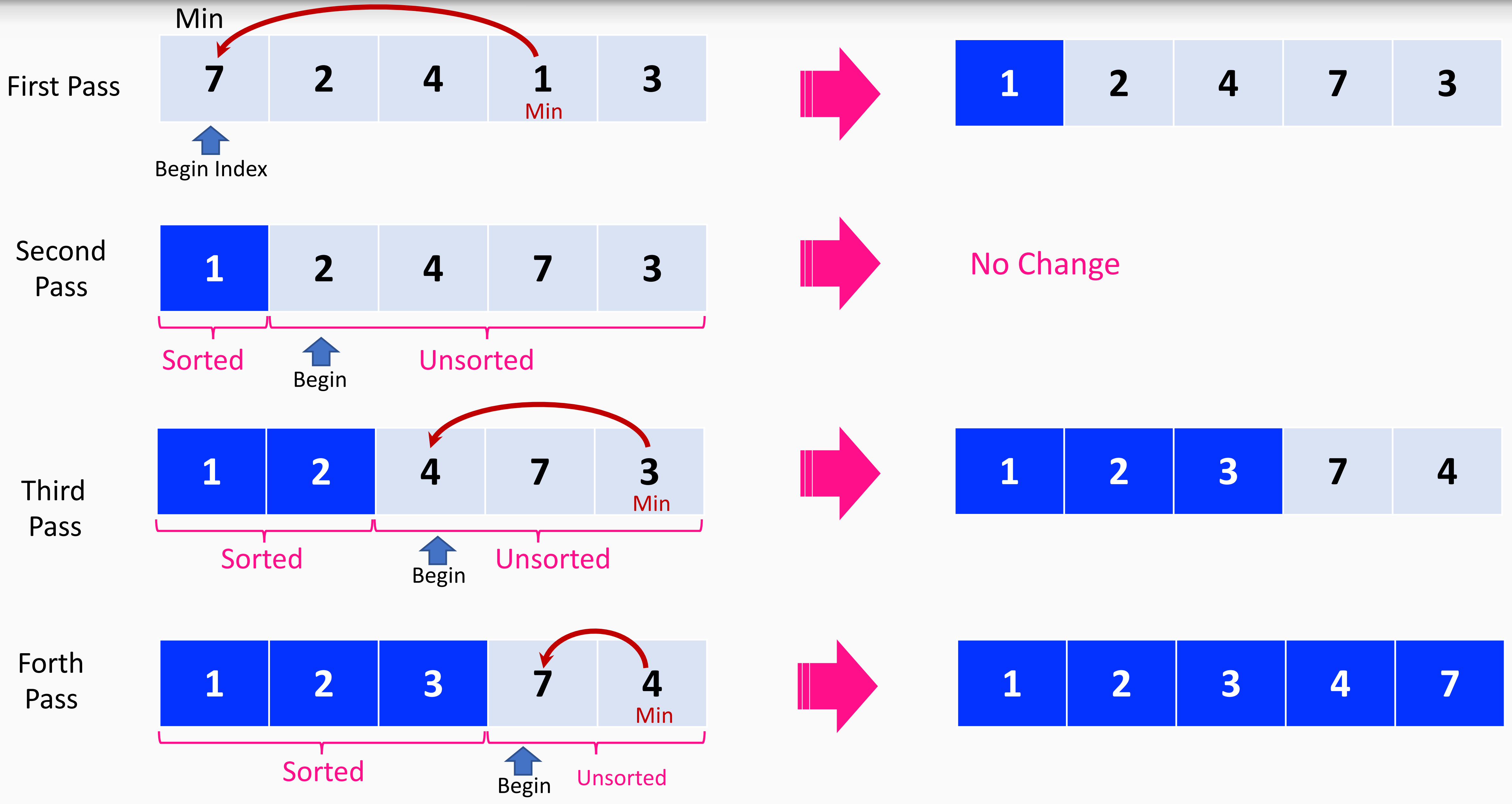
Step 2: Replace current with
the minimum:



Step 3: Now continue the same with the unsorted part:



Selection Sort



Selection Sort - Performance

SELECTION SORT

	Best	**Worst**
Passes	O(n)	O(n)
Comparisons	O(n)	O(n)
Total	**$O(n^2)$**	**$O(n^2)$**
	Quadratic	*Quadratic*

Worse than Bubble sort in best case!

Selection Sort Implementation in Java

Lets switch to IntelliJ for Selection Sort Implementation

Brute Force - Bubble Sort

- Simplest of all sorting algorithms

How it works:

- Start from the beginning and **compare ith and ith+1 element.**
- if **ith** is greater **swap ith and ith+1**
- Continue first two steps ***array.length*** times

```
➢for (i=0;i<array.length;i++)
        for (n=0;n<array.length-1;n++) {
            // Compare two sequential items if they are not in order swap them.
            - If a[n]>a[n+1] swap (n, n+1)
    }
```

- **In every iteration ith +1 largest bubbles up to its order.**

Brute Force - Bubble Sort

- **Algorithm:**

```
for (i=0;i<array.length; i++) {
    for (n=0;n<array.length-1;n++) { If a[n]>a[n+1] swap (n, n+1) }
}
```

Example: (7,2,4,1,3)

First Pass:

$(\boxed{\mathbf{7},2},4,1,3) \rightarrow (2,\mathbf{7},4,1,3)$

$(2,\boxed{\mathbf{7},4},1,3) \rightarrow (2,4,\mathbf{7},1,3)$

$(2,4,\boxed{\mathbf{7},1},3) \rightarrow (2,4,1,\mathbf{7},3)$

$(2,4,1,\boxed{\mathbf{7},3}) \rightarrow (2,4,1,3,\mathbf{7})$

Second Pass:

$(\boxed{2,\mathbf{4}},1,3,7) \rightarrow$ No Swap

$(2,\boxed{\mathbf{4},1},3,7) \rightarrow (2,1,\mathbf{4},3,7)$

$(2,1,\boxed{\mathbf{4},3},7) \rightarrow (2,1,3,\mathbf{4},7)$

$(2,1,3,\boxed{\mathbf{4},7}) \rightarrow$ No Swap

Third Pass:

$(\boxed{\mathbf{2},1},3,4,7) \rightarrow (1,\mathbf{2},3,4,7)$

$(1,2,3,4,7) \rightarrow$ No Swap

$(1,2,3,4,7) \rightarrow$ No Swap

$(1,2,3,4,7) \rightarrow$ No Swap

What does no swap through out a pass mean?

Brute Force - Bubble Sort

- Refine algorithm with “no swap” check:

```
for (i=0; i<array.length; i++) {
  boolean swap=false;
     for (n=0;n<array.length-1;n++) {
          If a[n]>a[n+1] { swap (n, n+1); swap=true;}
     }
   If (!swap) return;
}
```

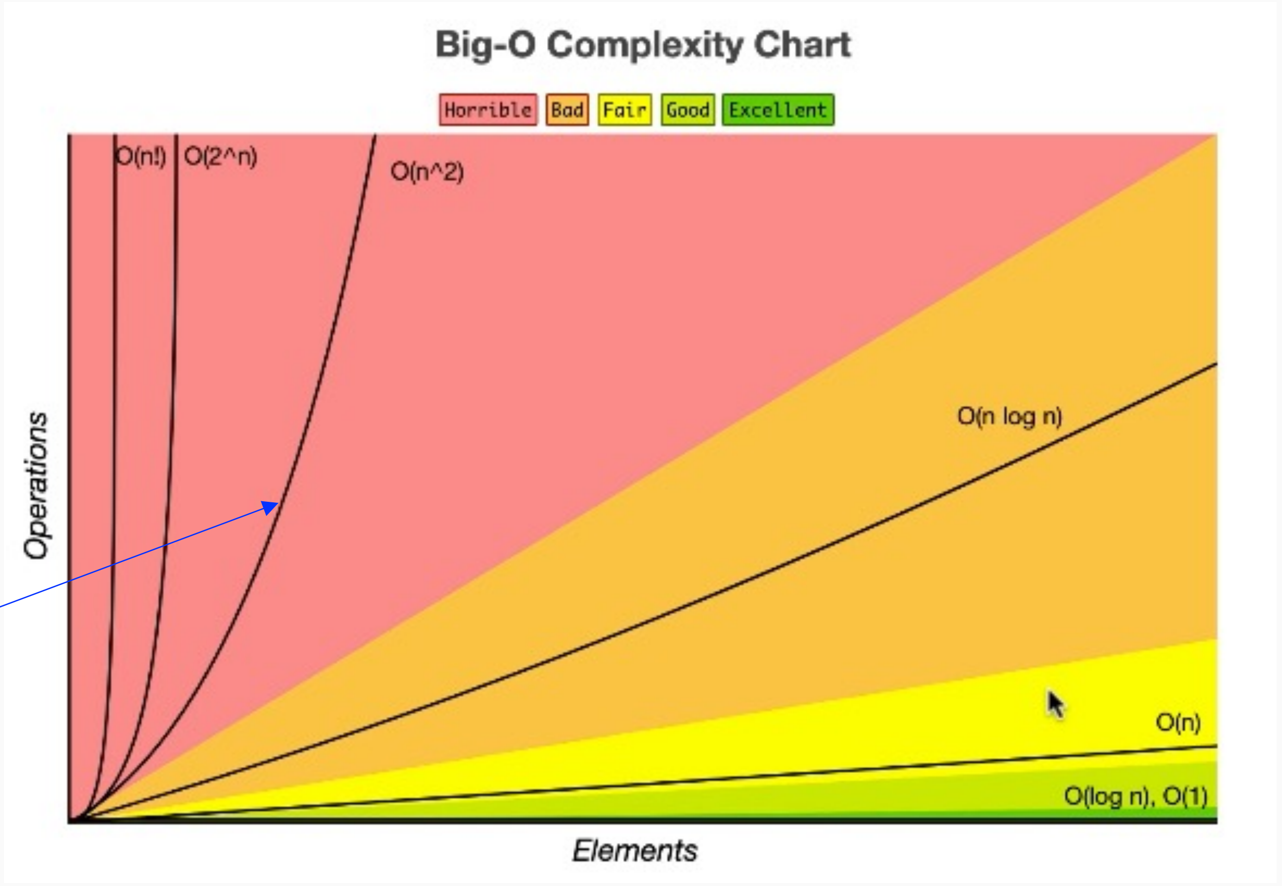
Bubble Sort Implementation in Java

Lets switch to IntelliJ for Bubble Sort Implementation

Sorting Algorithms- Bubble Sort

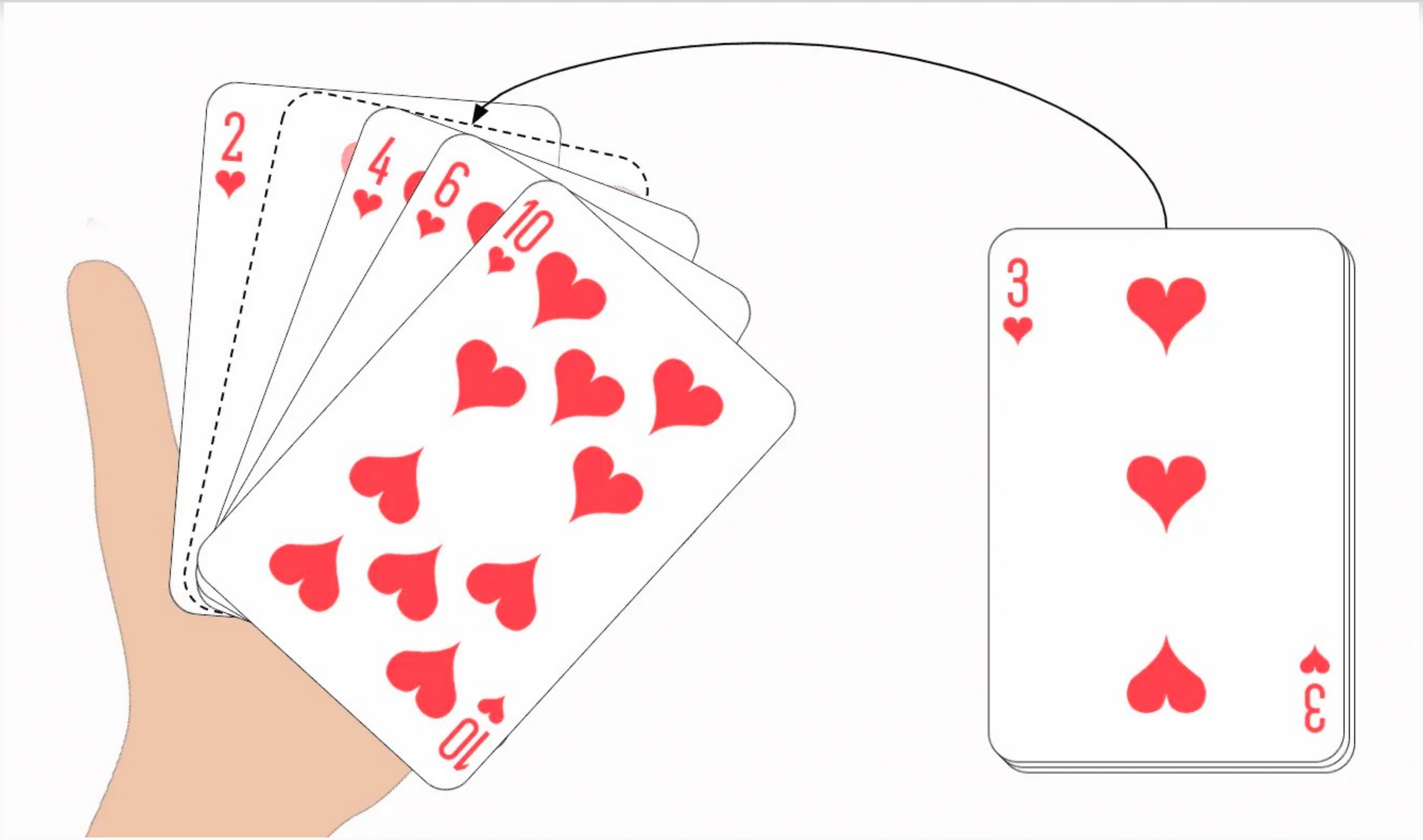
BUBBLE SORT

	Best	**Worst**
Passes	O(1)	O(n)
Comparisons	O(n)	O(n)
Total	**O(n)**	**O(n^2)**
	Linear	*Quadratic*



No swap in the first pass (already sorted).

Insertion Sort



Design strategy 2: Decrease and Conquer

- Decrease or reduce problem instance to smaller instance of the same problem and extend solution.
- Conquer the problem by solving smaller instance of the problem. Extend solution of smaller instance to obtain solution to original problem .
- Basic idea of the decrease-and-conquer technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instance.
- This approach is also known as incremental or inductive approach.
- It can be either top-down (recursive) or bottom-up (non-recursive)

Design strategy 2: Decrease and Conquer

- Decrease and conquer is a technique used to solve problems by reducing the size of the input data at each step of the solution process.
- This technique is similar to divide-and-conquer, in that it breaks down a problem into smaller subproblems, but the difference is that in decrease-and-conquer, the size of the input data is reduced at each step.
- The technique is used when it’s easier to solve a smaller version of the problem, and the solution to the smaller problem can be used to find the solution to the original problem.

Design strategy 2: Decrease and Conquer

- Decrease and conquer is a technique used to solve problems by reducing the size of the input data at each step of the solution process.
- This technique is similar to divide-and-conquer, in that it breaks down a problem into smaller subproblems, but the difference is that in decrease-and-conquer, the size of the input data is reduced at each step.
- The technique is used when it's easier to solve a smaller version of the problem, and the solution to the smaller problem can be used to find the solution to the original problem.

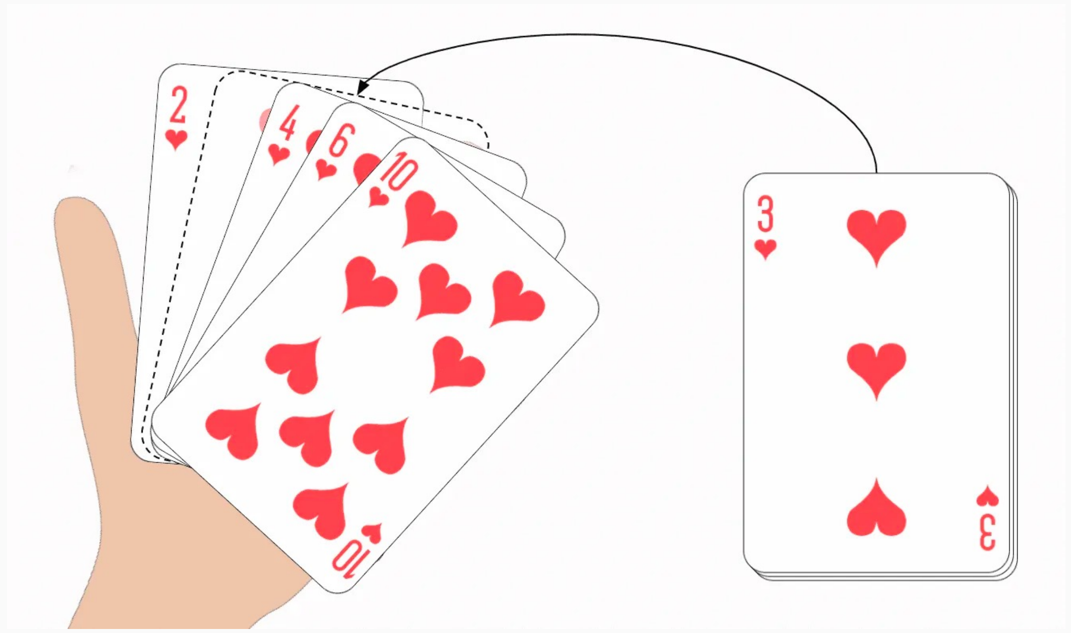
Design strategy 2: Decrease and Conquer

- There are three major variations of decrease-and-conquer:

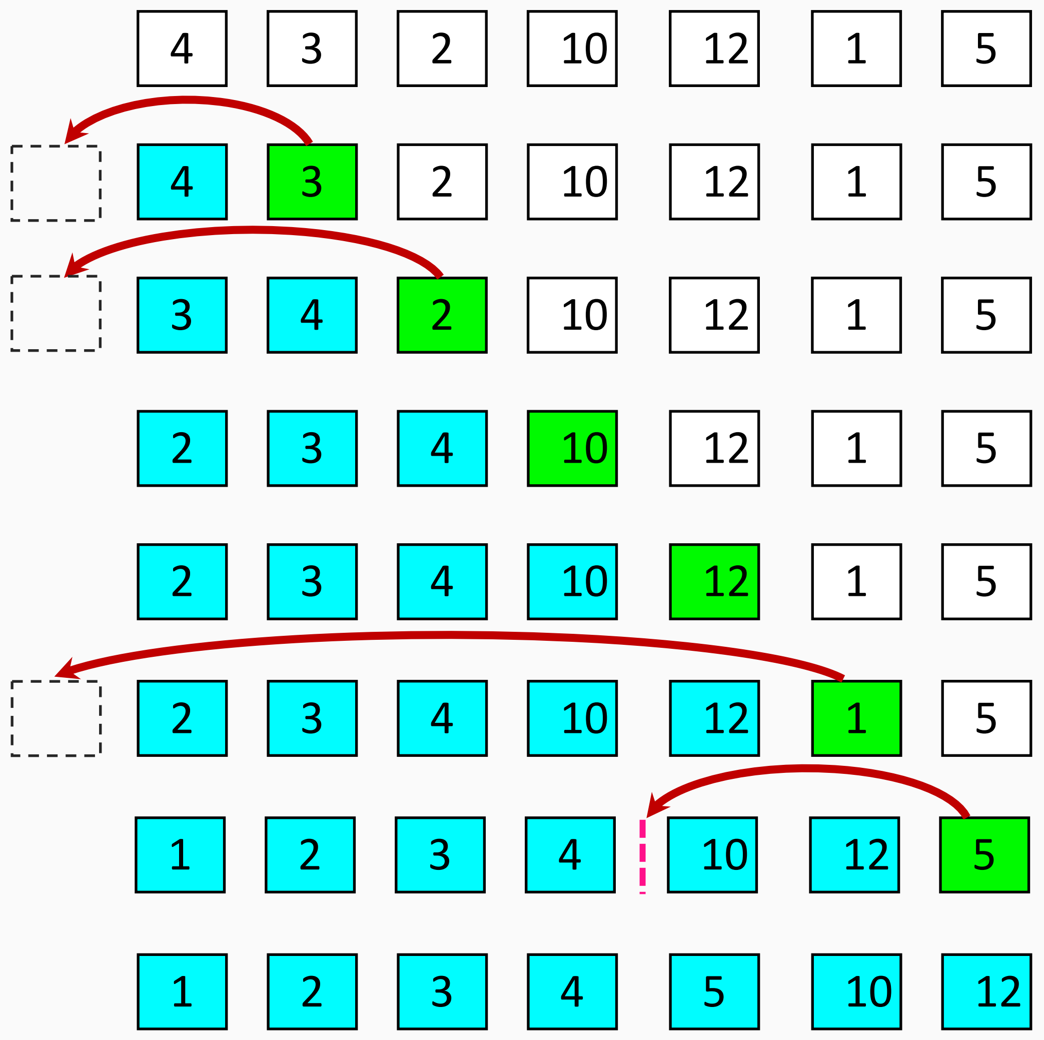
    1. Decrease by a constant
        - *Insertion sort*
    2. Decrease by a constant factor
        - *Binary Search*
    3. Variable size decrease
        - *Euclidean algorithms (Greatest Common Divisor)*

Decrease and Conquer : Insertion Sort

- Very similar to Selection sort (in terms of sorted-unsorted part of array)
- The array is virtually split into a sorted and an unsorted part.
- Values from the unsorted part are picked and inserted into the correct position in the sorted part.



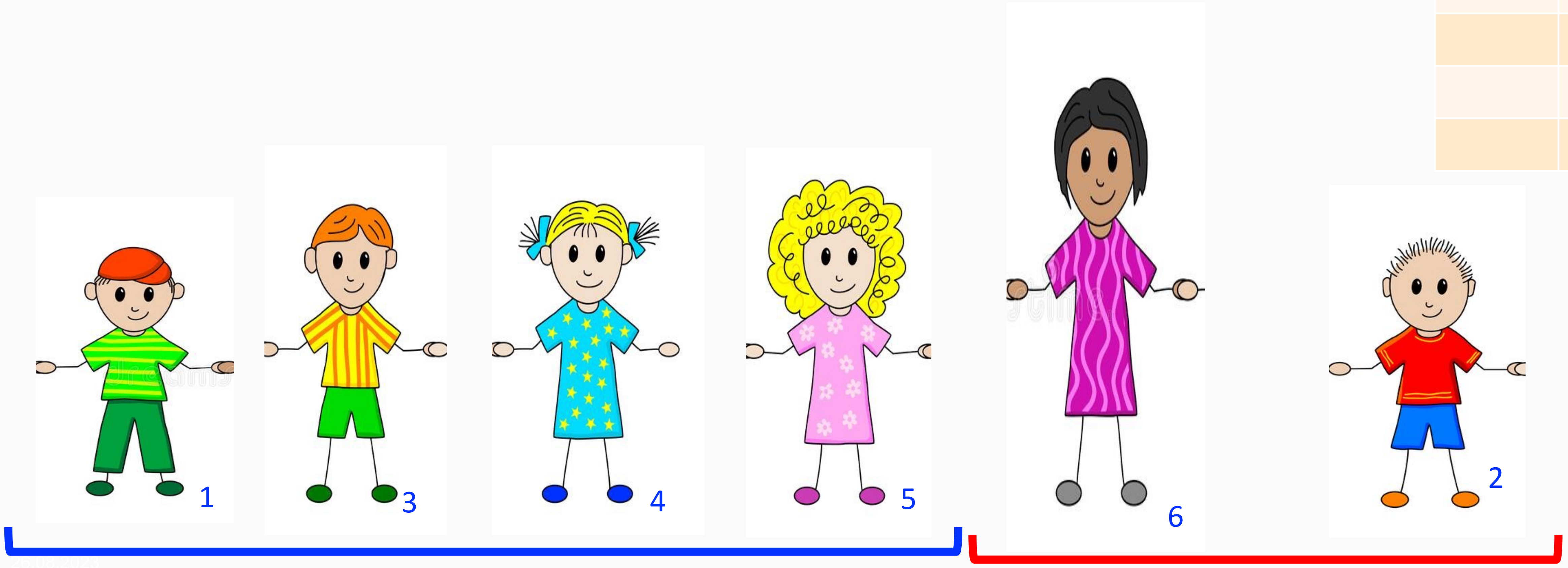
Insertion Sort- How It Works?



Insertion Sort

toBeInserted=4

i	J
1	0
2	1
3	2



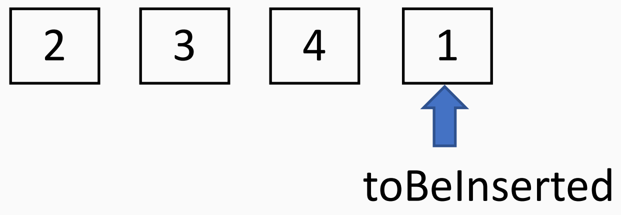
Unsorted Part

Sorting Algorithms – Insertion Sort

To sort an array of size N in ascending order:

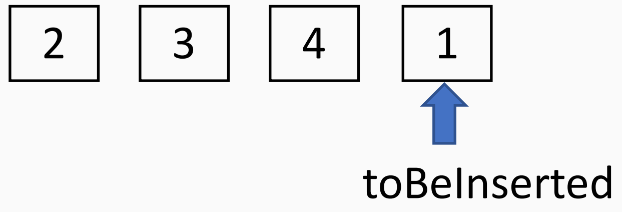
- Iterate from arr[1] to arr[N] over the array.
- Compare the current element (toBeInserted) to its predecessor.
- If the toBeInserted is smaller than its predecessor, compare it to the elements before.

  Move the greater elements one position up to make space for the swapped element.



Sorting Algorithms – Insertion Sort

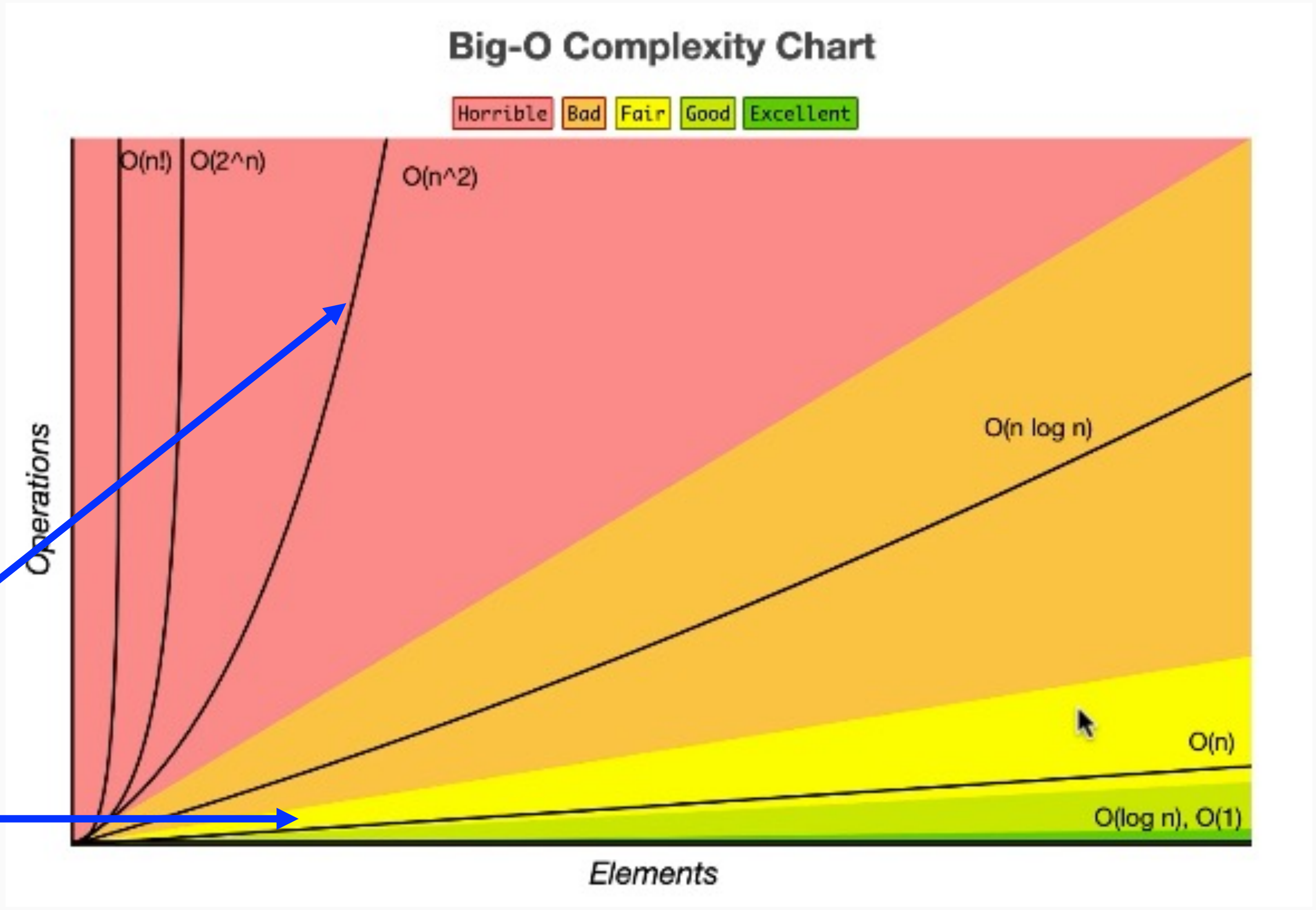
```
insertionSort(int[] array){
  for(int i=1;i<array.length;i++) {  //Iterate from arr[1] to arr[N] over the array.
    int toBeInserted=array[i];       // Save ith value
    int j=i-1;                       // Start from i –1  (predecessor)
    while(j>=0 && array[j]>toBeInserted) {  // Go back and compare until ith < array[j]
      array[j+1]=array[j]; // shifting operation here
      j--;
    }
    array[j+1] = toBeInserted;
  }
  return array;
}
```



Insertion Sort - Performance

INSERTION SORT

	Best	**Worst**
Iteration	O(n)	O(n)
Shift Items	O(1)	O(n)
Total	**O(n)**	**$O(n^2)$**
	Linear	*Quadratic*



Performance similar to Bubble Sort

Insertion Sort - Implementation

Lets switch to IntelliJ for Insertion Sort Implementation

Design Strategy 3 - Divide and Conquer

- This technique can be divided into the following three parts:

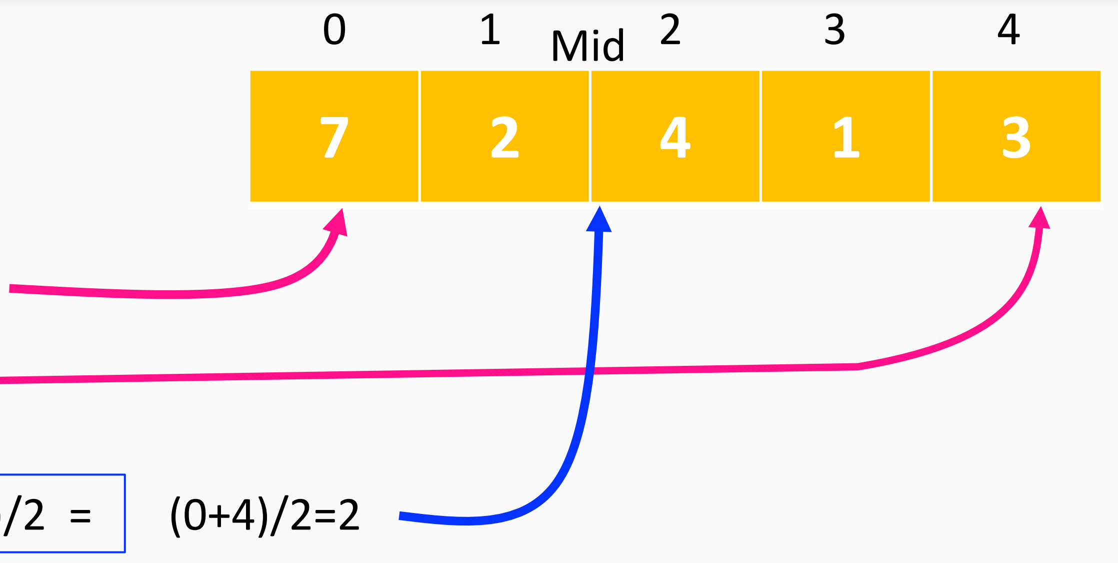
1. **Divide:** This involves dividing the problem into some sub problem.
2. **Conquer:** Solve sub problems.
3. **Combine:** Combine the sub-solutions to get the final solution.

Divide and Conquer- Merge Sort

- Merge Sort is a Divide and Conquer algorithm.
- Idea is to break down the list into smaller and smaller sublists.
- We continue to break down the list until we get a sublist with one element.
- Sort sublists and merge them to produce sorted sublist/list.

Merge Sort

1. Divide

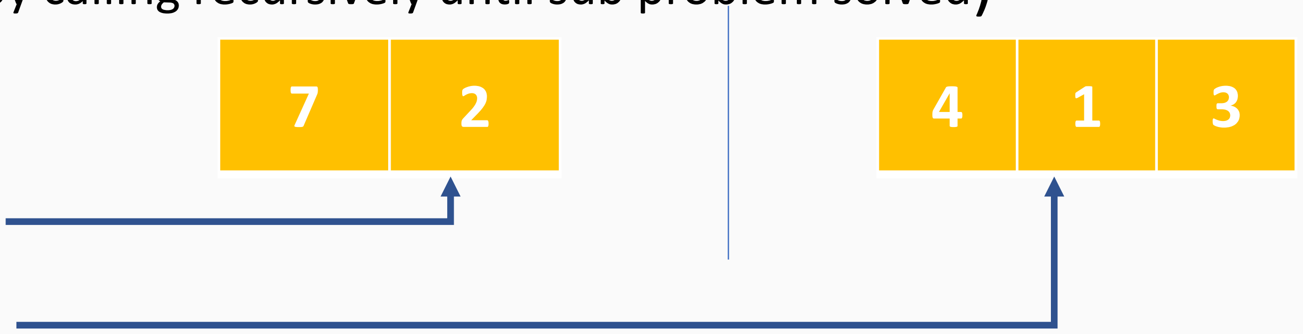


Lower Bound (Lb)

Upper Bound (Ub)

$\boxed{\text{Middle= (Lb+Ub)/2 =}}$ (0+4)/2=2

2. Conquer- (Sub problem by calling recursively until sub problem solved)

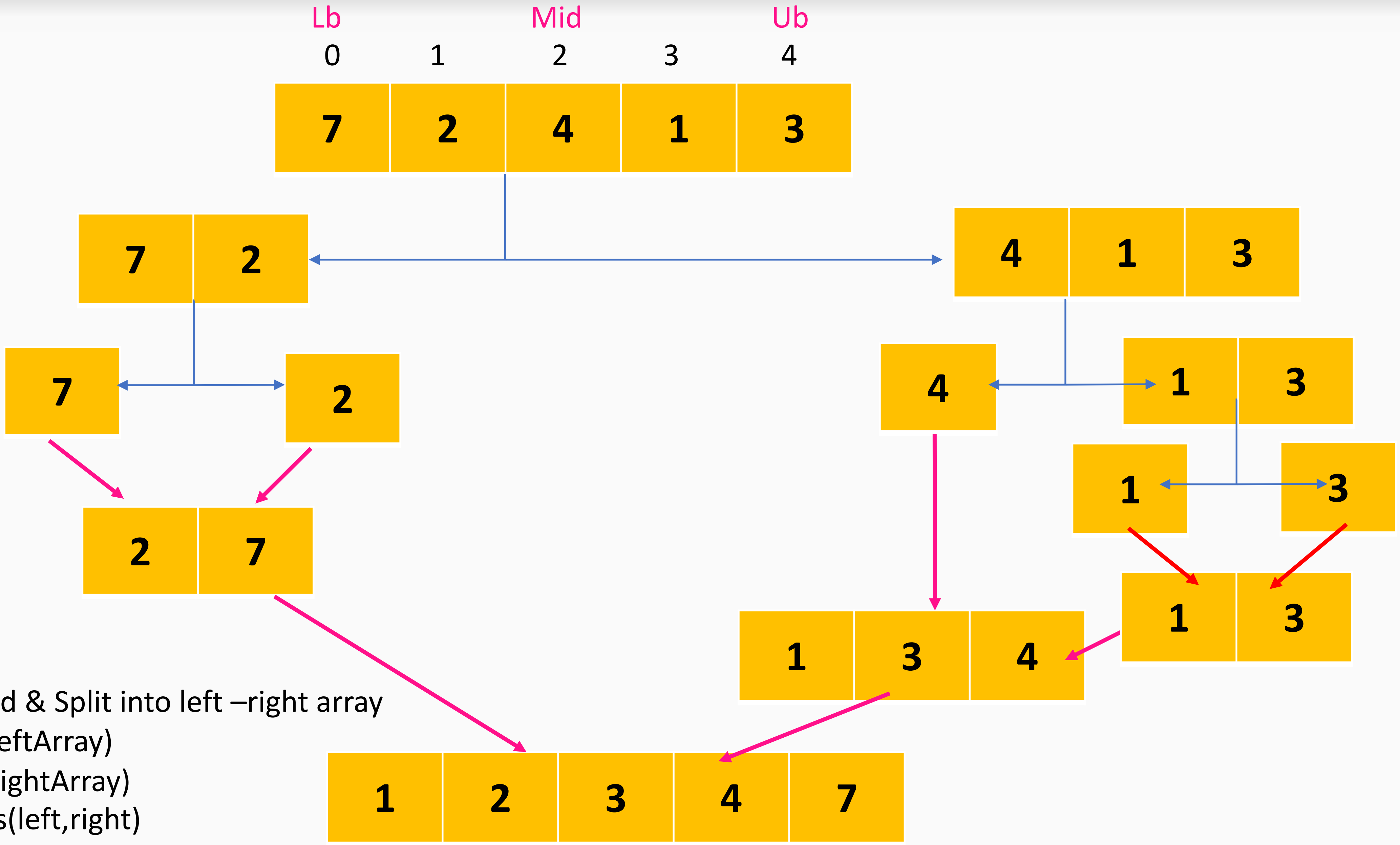


- MergeSort (Array, Lb, Mid-1)
- MergeSort (Array, Mid, Ub)

3. Merge Sub-Lists by Sorting

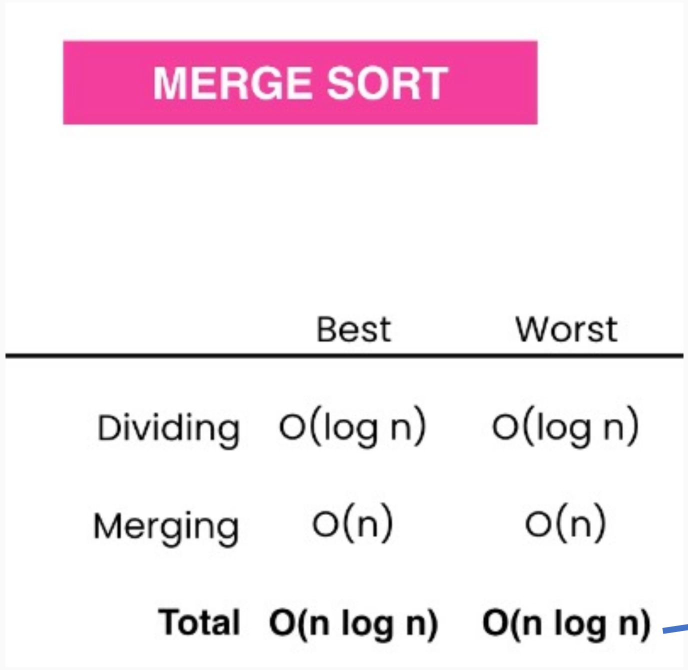
- Merge (Array, Lb, Mid, Ub) ➡ While merging do sorting here !

Merge Sort



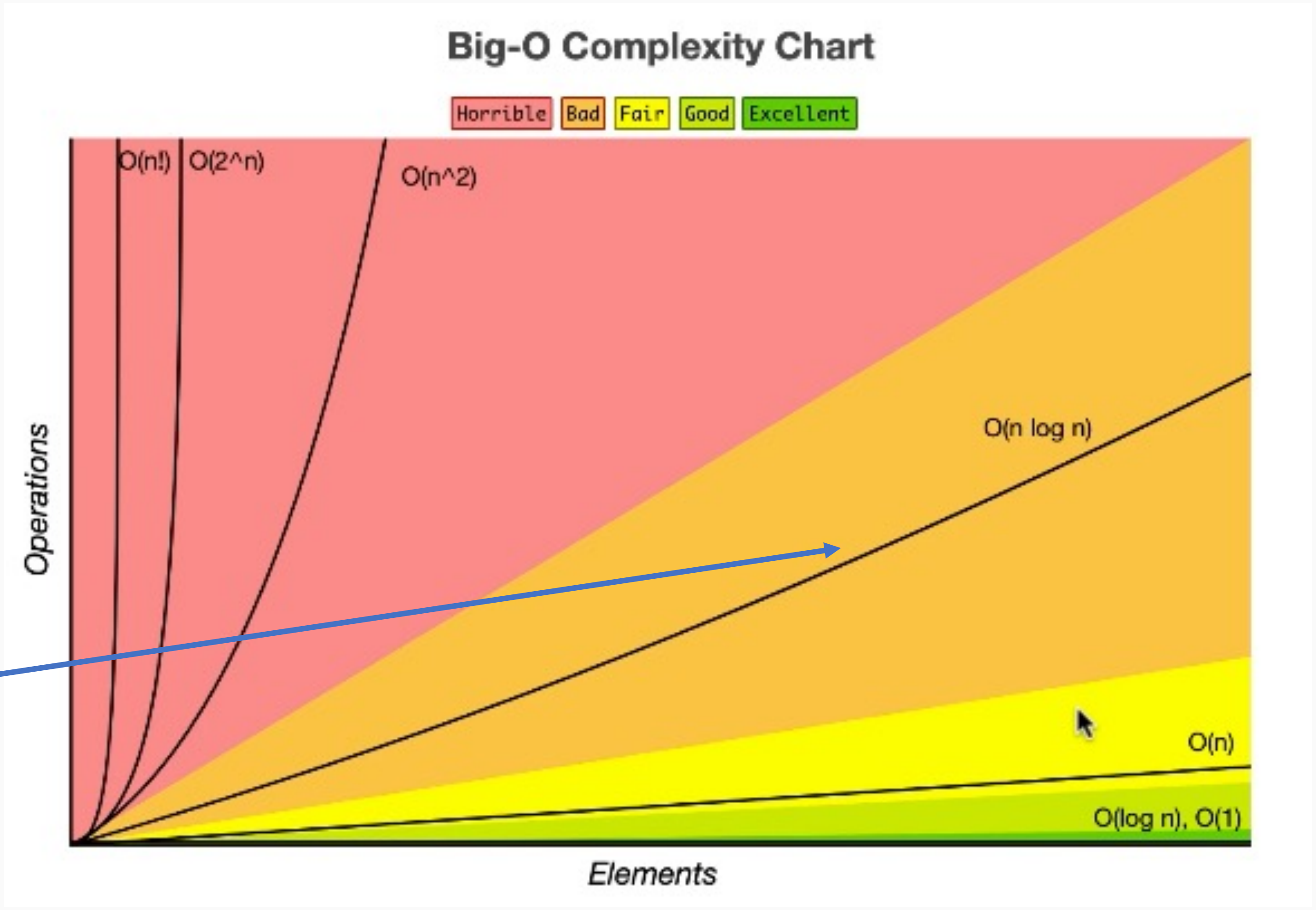
```
MergeSort(){
        Calculate mid & Split into left –right array
        MergeSort(leftArray)
        MergeSort(rightArray)
        MergeArrays(left,right)
```

Merge Sort-Performance



MERGE SORT

	Best	Worst
Dividing	O(log n)	O(log n)
Merging	O(n)	O(n)
Total	**O(n log n)**	**O(n log n)**



- But requires additional space for divided Arrays.

Merge Sort Algorithm- Implementation

```
• MergeSort(Array){
    if (array.length<2) return;  // single item
          mid=(array.length/2);

          leftArray=Array[0.. mid-1];
          rightArray=Array[mid.. array.length];

          MergeSort(leftArray);
          MergeSort(RightArray);

            Merge(leftArray, rightArray, Array)
    }
}
```

Merge Sort - Implementation

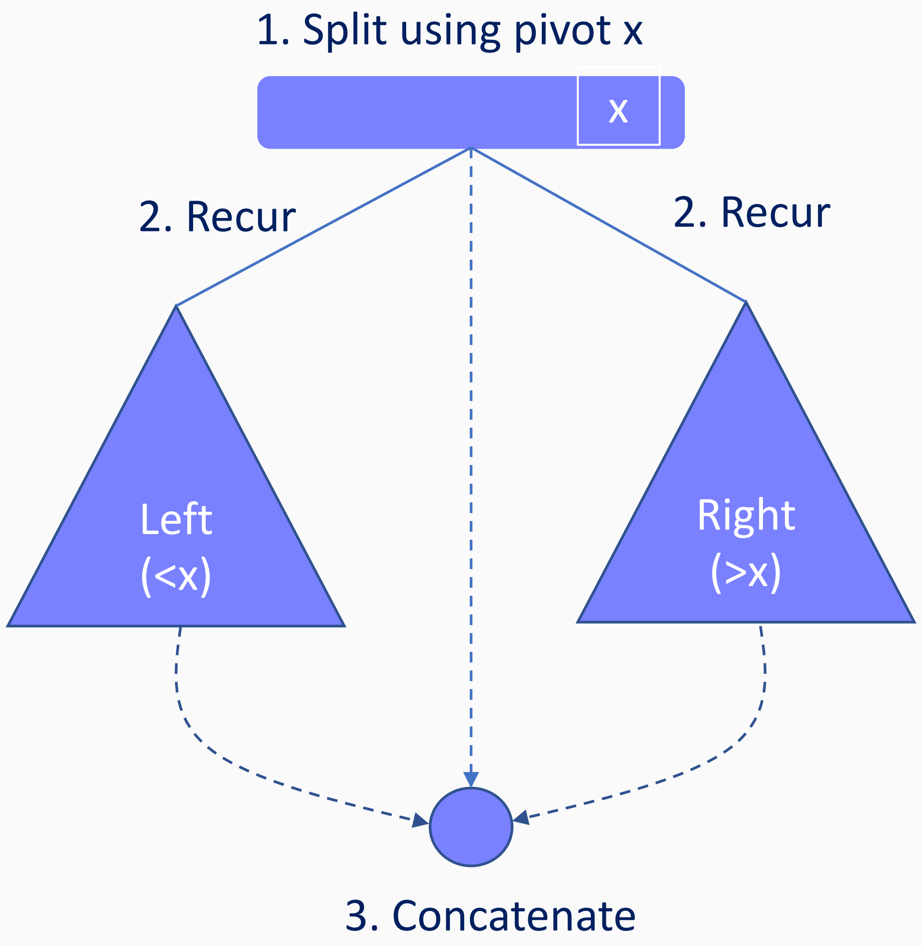
Lets switch to IntelliJ for Insertion Sort Implementation

Divide and Conquer– Quick Sort

- Quicksort algorithm is also based on the divide-and-conquer paradigm.
- In particular, the quick-sort algorithm consists of the following three steps :

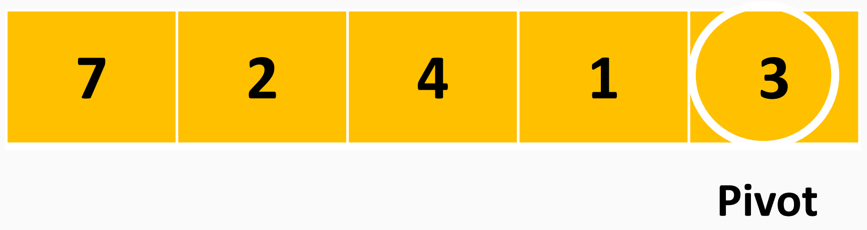
1. **Divide:** If S sequence has at least two elements, select a specific element *x* from S which is called as pivot. As is a common practice, choose last element as pivot. Split S into 3 subsequences.
   - **left**: elements less than *x*
   - **right:** elements greater than *x*
   - **middle:** elements equal to *x*
2. **Conquer:** Recursively sort sequences **left** and **right**.
3. **Combine:** Put back elements into S in order by first inserting **left** , then **middle** and then **right** subsequence.

Sorting Algorithms – Quick Sort



Sorting Algorithms – Quick Sort

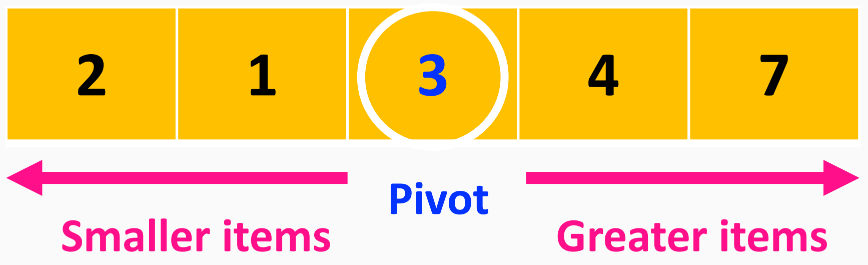
1. Select a pivot.



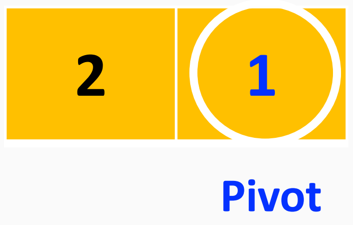
How to Pick Up a Pivot

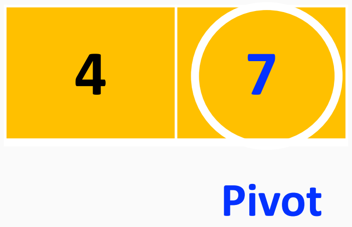
- Always pick first element as pivot.
- Always pick last element as pivot
- Pick a random element as pivot.
- Pick median as pivot.

2. Arrange all greater on the right, lesser on the left. (Partitioning)



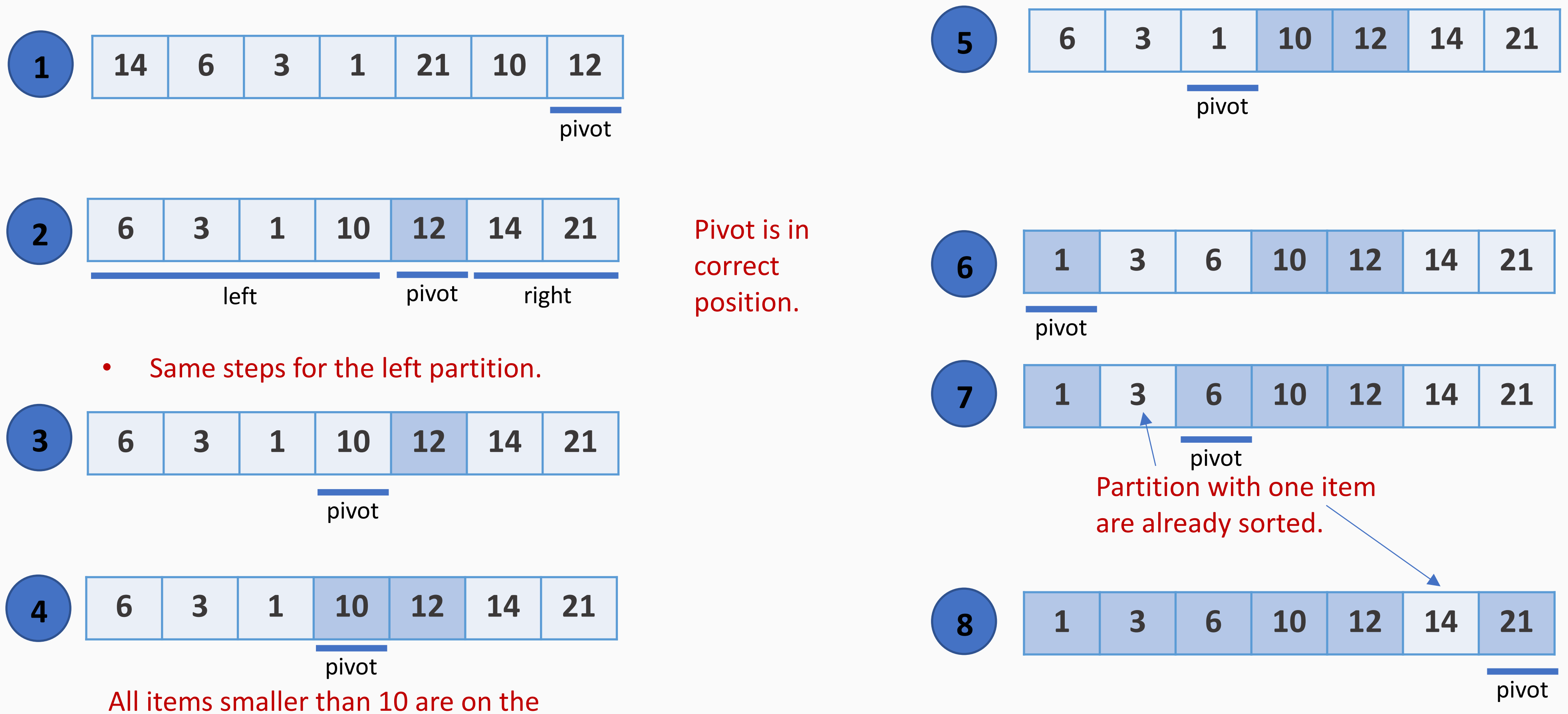
3. Perform the same steps for the partitions.





Sorting Algorithms – Quick Sort

- Select a pivot.
- Arrange all larger on the right, smaller on the left.
- Perform the same steps for the partitions.



Pivot is in correct position.

- Same steps for the left partition.

All items smaller than 10 are on the left. Pivot is in correct position.

Move the larger items on right

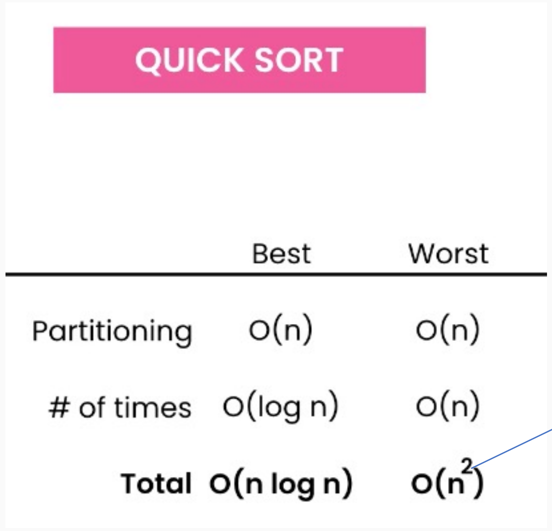
Pivot is in correct position.

Partition with one item are already sorted.

Pivot is in correct position.

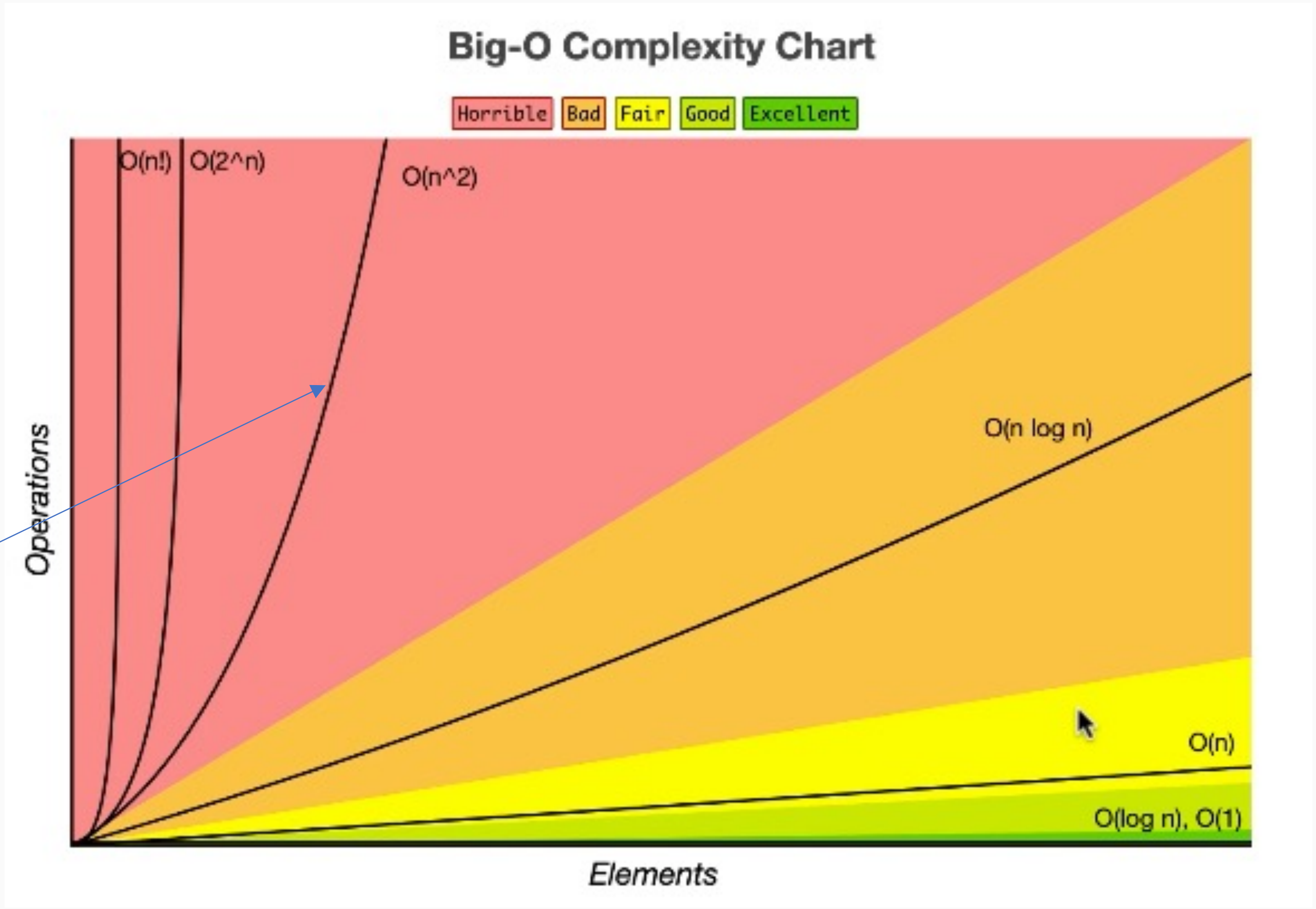
Pivot is in correct position.

Quick Sort-Performance



QUICK SORT

	Best	Worst
Partitioning	O(n)	O(n)
# of times	O(log n)	O(n)
Total	**O(n log n)**	**$O(n^2)$**



Arrays.sort() is using Dual-Pivot QuickSort which offers nlogn

Quick Sort - Implementation

Lets switch to IntelliJ for Insertion Sort Implementation

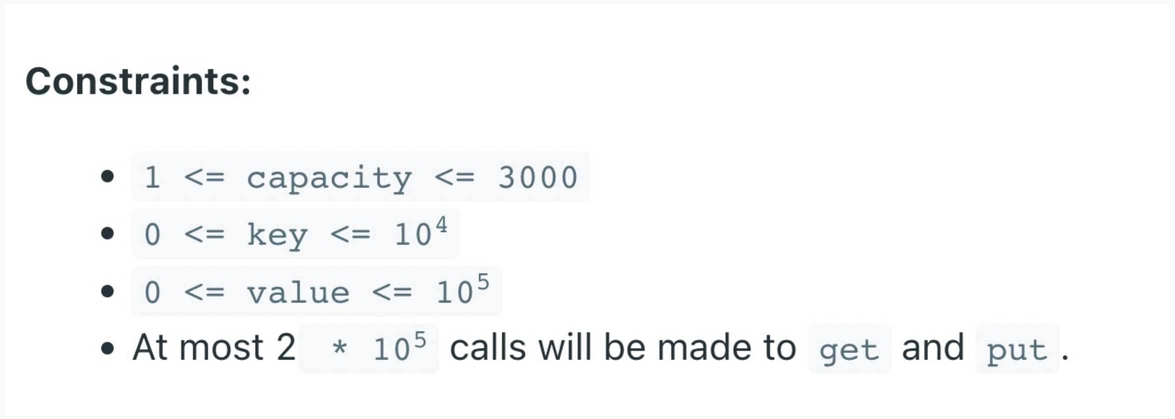
Algo Question – LRU Cache

Problem: LRU Cache

Design a data structure that follows the constraints of a **Least Recently Used (LRU) cache**.

Implement the LRUCache class:

- LRUCache(int capacity) Initialize the LRU cache with **positive** size capacity.
- int get(int key) Return the value of the key if the key exists, otherwise return -1.
- void put(int key, int value) Update the value of the key if the key exists. Otherwise, add the key-value pair to the cache. If the number of keys exceeds the capacity from this operation, **evict** the least recently used key.
- The functions get and put must each run in O(1) average time complexity.

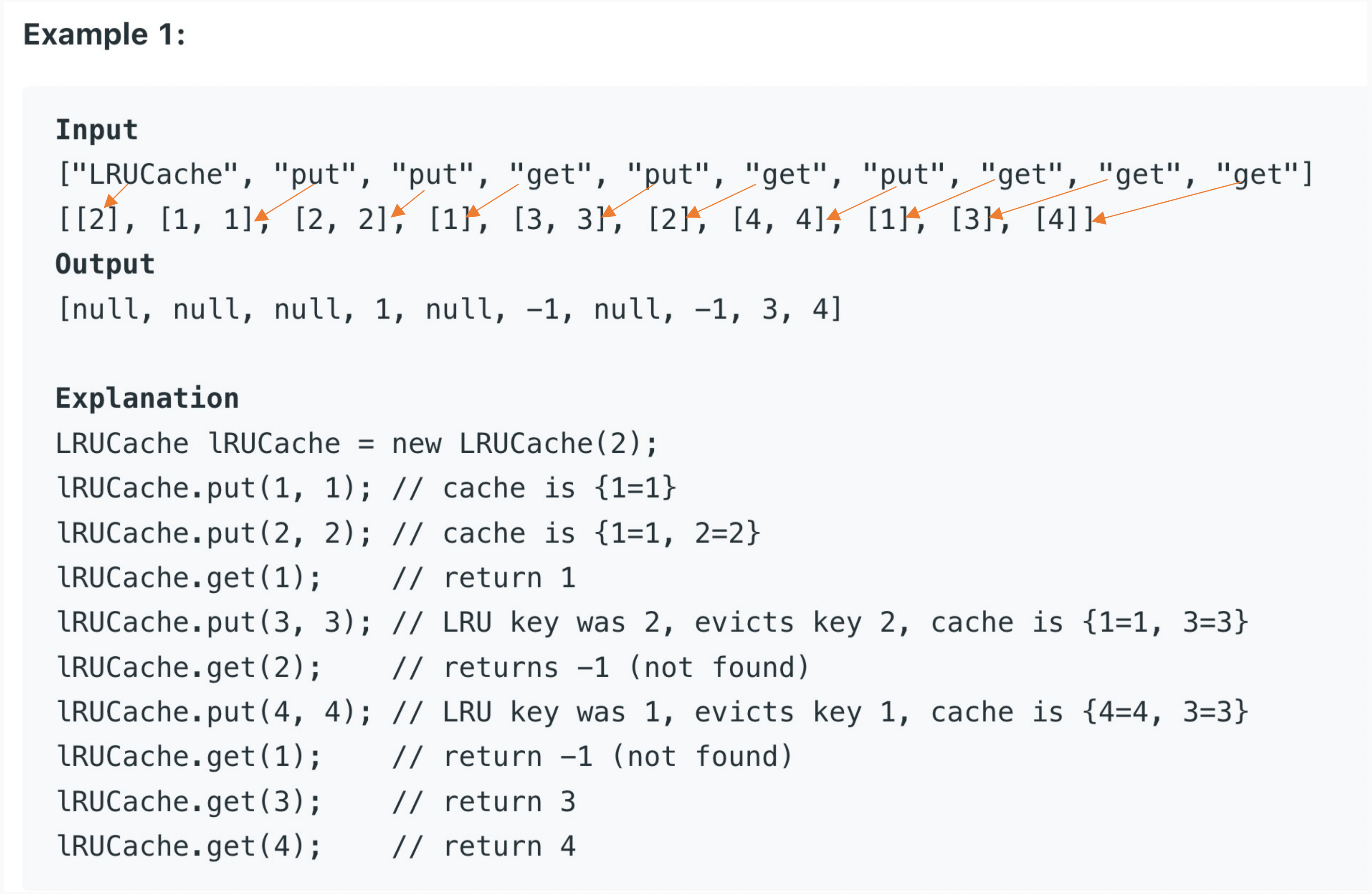


Constraints:

- `1 <= capacity <= 3000`
- `0 <= key <=` 10^4
- `0 <= value <=` 10^5
- At most $2 * 10^5$ calls will be made to `get` and `put`.

Algo Question – LRU Cache

Problem: LRU Cache – Example

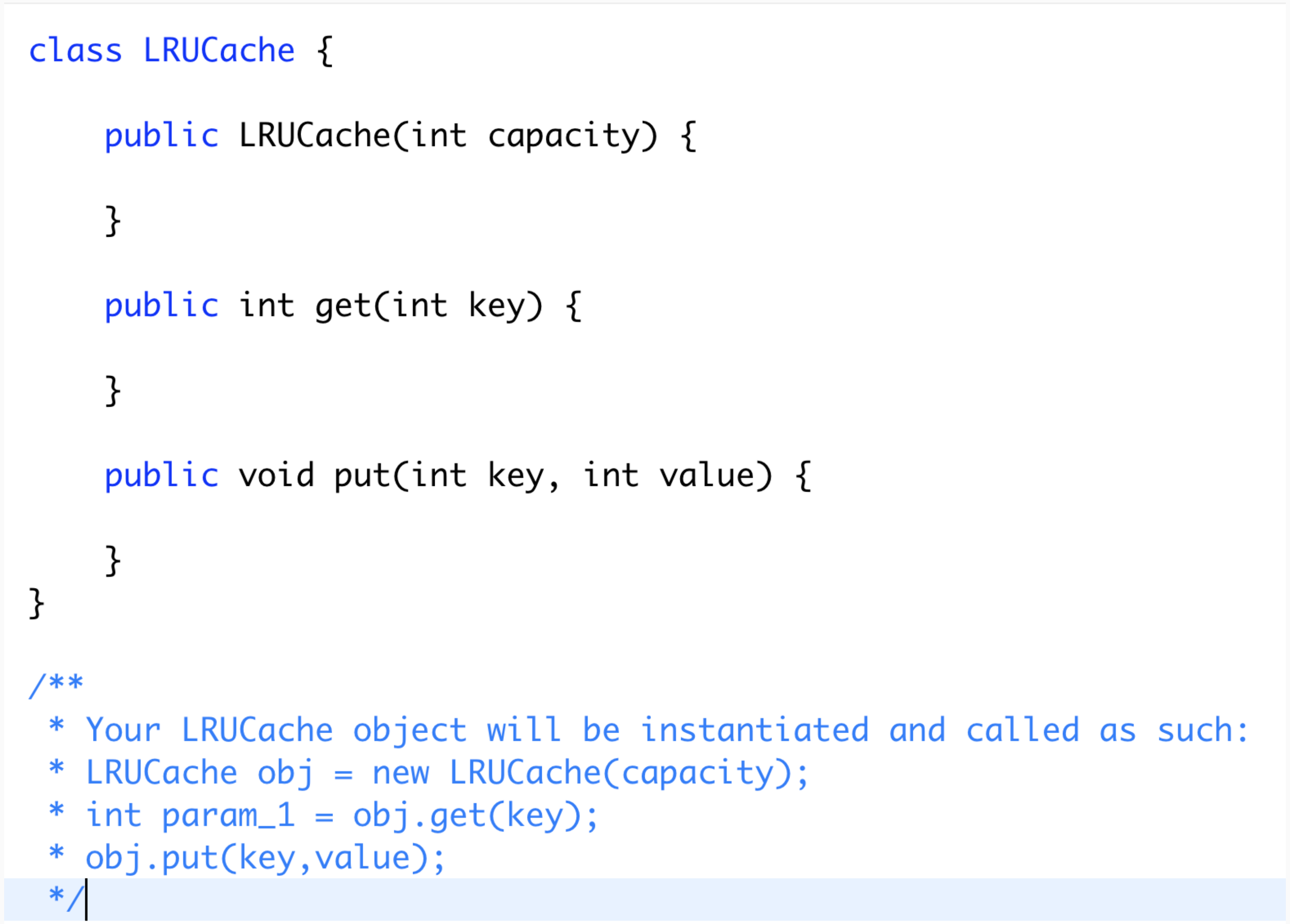


Example 1:

```
Input
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
Output
[null, null, null, 1, null, -1, null, -1, 3, 4]

Explanation
LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // cache is {1=1}
lRUCache.put(2, 2); // cache is {1=1, 2=2}
lRUCache.get(1);    // return 1
lRUCache.put(3, 3); // LRU key was 2, evicts key 2, cache is {1=1, 3=3}
lRUCache.get(2);    // returns -1 (not found)
lRUCache.put(4, 4); // LRU key was 1, evicts key 1, cache is {4=4, 3=3}
lRUCache.get(1);    // return -1 (not found)
lRUCache.get(3);    // return 3
lRUCache.get(4);    // return 4
```

Algo Question – LRU Cache

Problem: LRU Cache – Solution Template



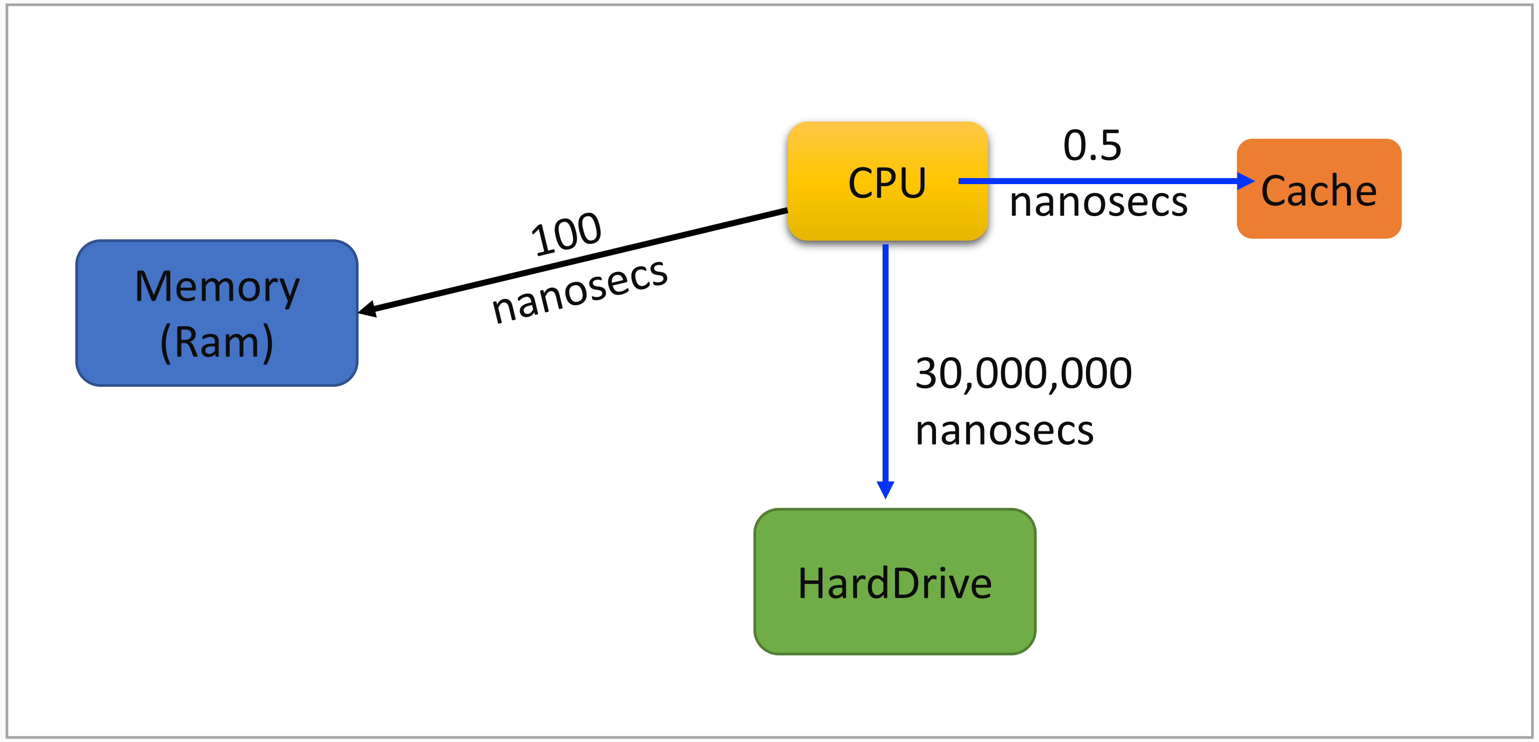
```
class LRUCache {

    public LRUCache(int capacity) {

    }

    public int get(int key) {

    }

    public void put(int key, int value) {

    }
}

/**
 * Your LRUCache object will be instantiated and called as such:
 * LRUCache obj = new LRUCache(capacity);
 * int param_1 = obj.get(key);
 * obj.put(key,value);
 */
```

Algo Question – LRU Cache

1. Understand the problem

What is a Cache?

Approximate cost to access various caches and main memory?



Algo Question – LRU Cache

1. Understand the problem

What is a LRUCache?

Least **R**ecently **U**sed item will be removed or overwritten if capacity is reached.

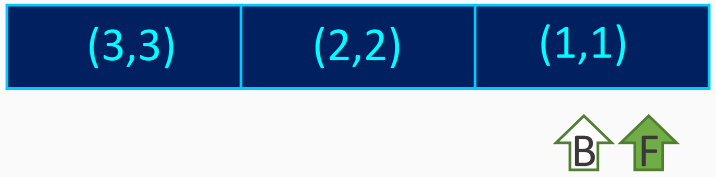
LRUCache lRUCache = new LRUCache(3)

lRUCache.put(1, 1);

lRUCache.put(2, 2);

lRUCache.put(3, 3);

lRUCache.put(4, 4); (4,4)



Algo Question – LRU Cache

1. Understand the problem

What is a LRUCache?

Least recently used item will be removed or overwritten if capacity is reached.

LRUCache lRUCache = new LRUCache(3)

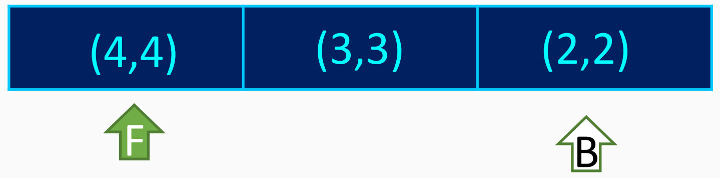
lRUCache.put(1, 1);

lRUCache.put(2, 2);

lRUCache.put(3, 3);

lRUCache.put(4, 4);

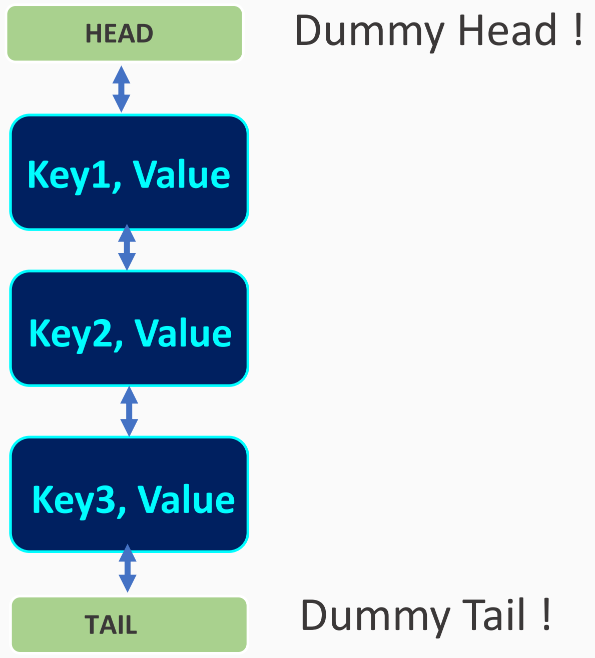
lRUCache.get(2);   Return 2;



Algo Question – LRU Cache

2. Model the Problem

```
public void put(int key, int value) {
    -Check if node with key exists
    - if (not exist) create a new node(key, value)
        -Update size;
        -Add just after head
        - If (size>capacity) pop tail and size--;
      //else case
    -  If (exists) append, remove and add just after head
}


public int get(int key) {
    if(node not_exists) return -1;
        else remove and add just after head
        return node.value;
  }
```

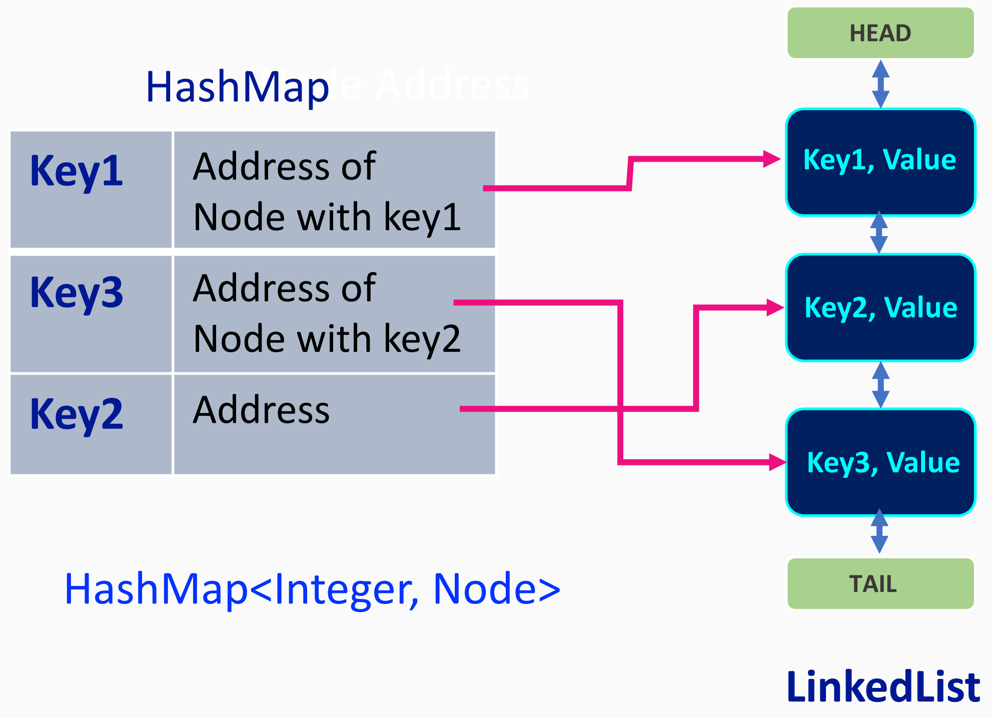


Doubly LinkedList

Algo Question – LRU Cache

3. Optimize the Solution

Problem states : “The functions get and put must each run in O(1) average time complexity.”



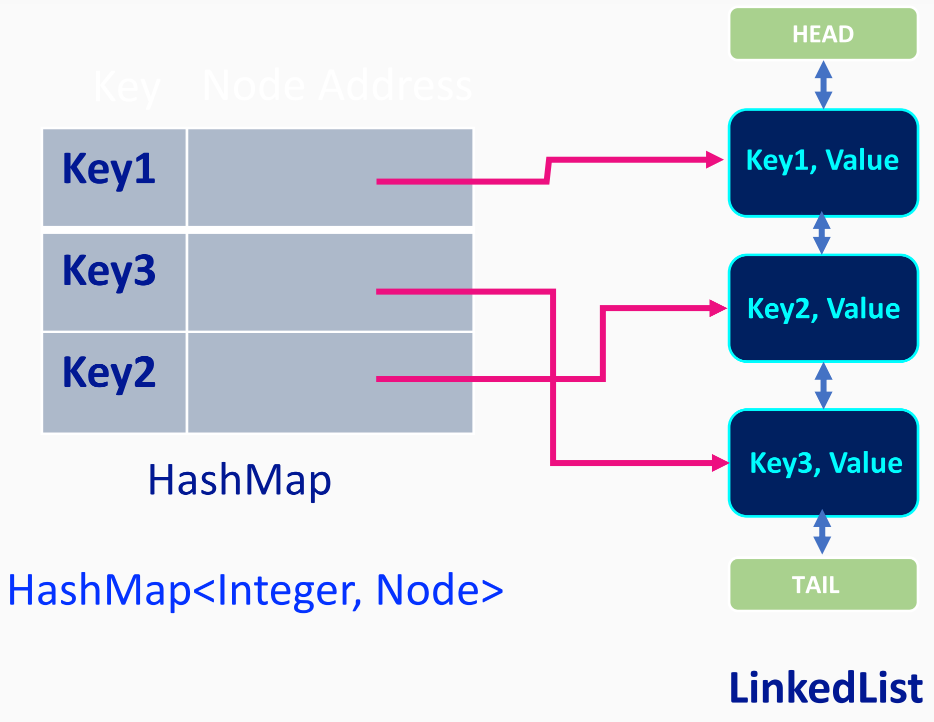
Key1	Address of Node with key1
Key3	Address of Node with key2
Key2	Address

HashMap<Integer, Node>

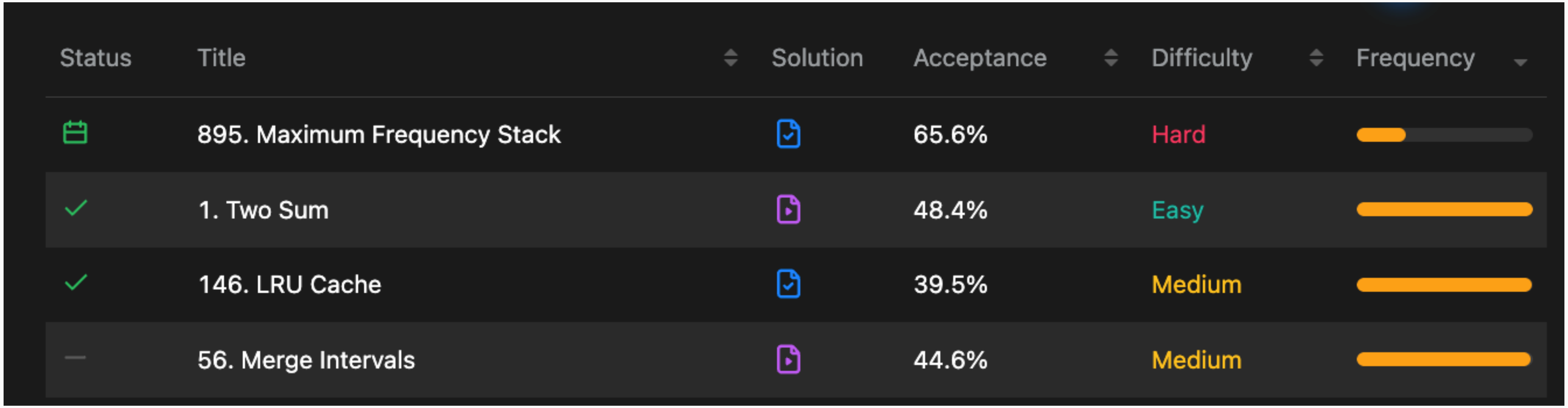
Algo Question – LRU Cache

3. Optimize the Solution

```
public void put(int key, int value) {
        -Check if node with key exists from HashMap
        - if (not exist) create a new node(key, value)
                -Update size;
                -Add just after head and put it in HashMap
                - If (size>capacity) pop tail and size--;
                   and remove from HashMap
        //else case
        -  If (exists) append, remove and add just after head
}

public int get(int key) {
        if(not in HashMap) return -1;
                else remove and add just after head
                return node.value;
   }
```



Why did we pick this question?



Status	Title	Solution	Acceptance	Difficulty	Frequency
	895. Maximum Frequency Stack		65.6%	Hard	
✓	1. Two Sum		48.4%	Easy	
✓	146. LRU Cache		39.5%	Medium	
—	56. Merge Intervals		44.6%	Medium	

You can exercise Linked Lists + Queues + HashMap

Algo Question – LRU Cache

Lets switch to IntelliJ for Solution of LRU Cache

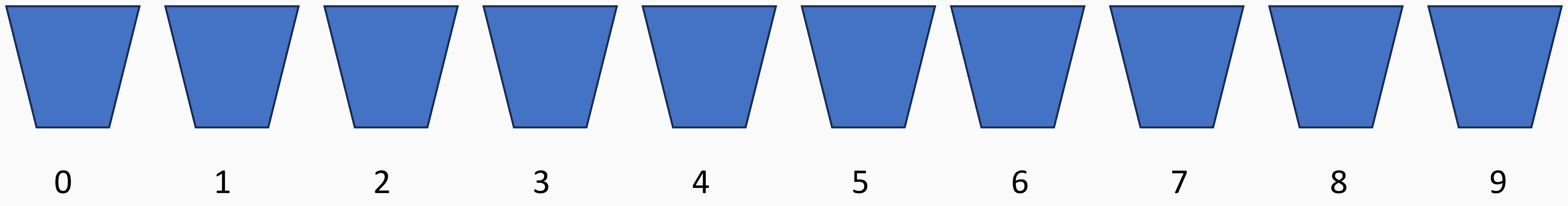
Linear time sorting– Bucket Sort Algorithm

- Bucket sort, also known as bin sort, is a sorting algorithm that divides an array's elements into several buckets.
- The buckets are then sorted one at a time, either using a different sorting algorithm or by recursively applying the bucket sorting algorithm.

Suppose the input array is:

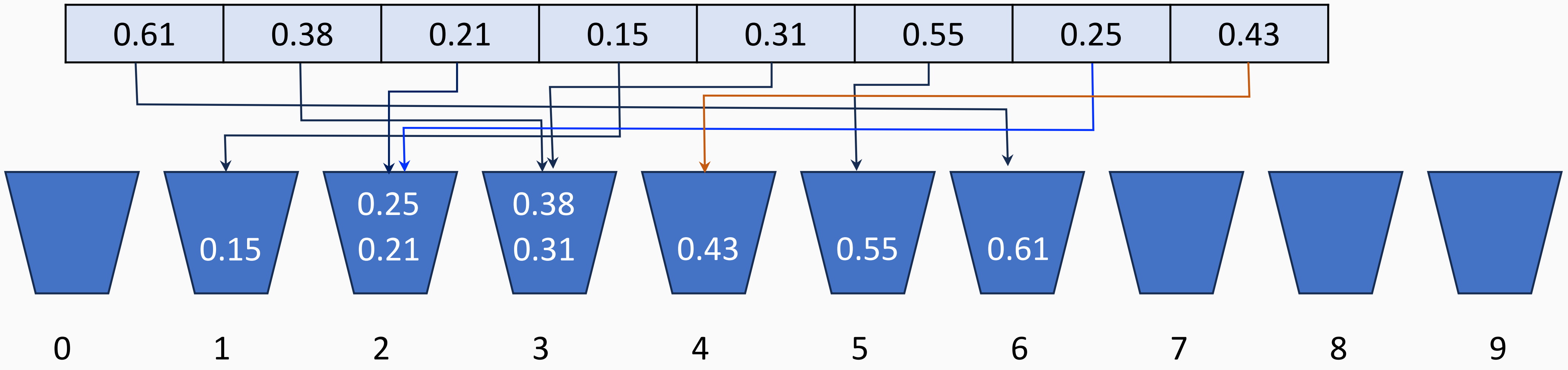
0.61	0.38	0.21	0.15	0.31	0.55	0.25	0.43

Create N empty buckets:



Linear time sorting– Bucket Sort Algorithm

- Toss every element into its appropriate bucket:



- Sort each bucket individually by applying a sorting algorithm:

- Concatenate all the sorted buckets:

Bucket Sort Algorithm- Performance

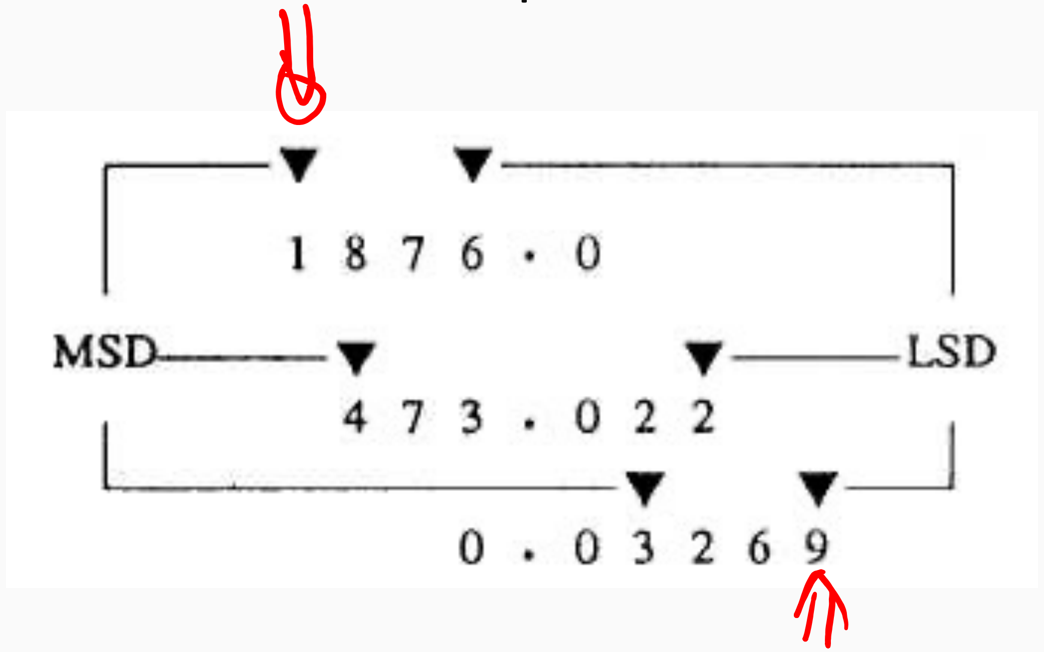
- Bucket sort is generally used with floating-point values
- When input is distributed uniformly over a range
- Not useful if we have a large array because the cost is increased
- It is not an in-place sorting algorithm.

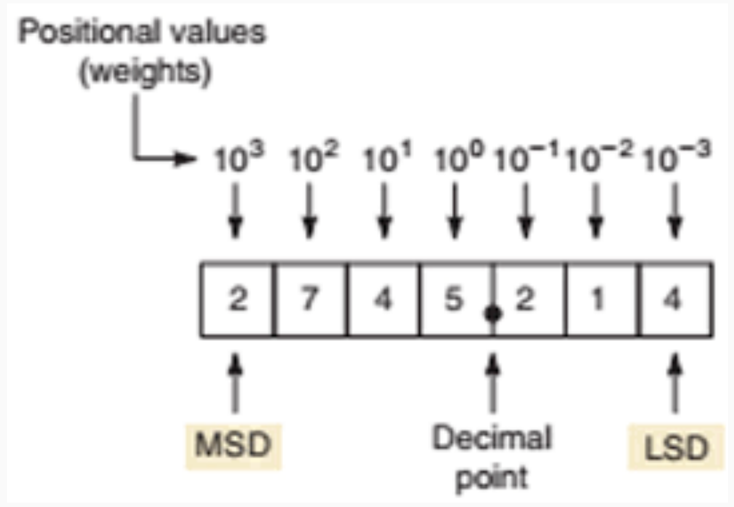
Performance Comparison		
Best Case	Average Case	Worst Case
O (n+k)	O (n+k)	O(n^2)

k: number of buckets

Linear time sorting– Radix Sort Algorithm

- Radix sort algorithm is a non-comparative sorting algorithm in computer science.
- It avoids comparison by creating and categorizing elements based on their radix or base. For elements with more than one significant digit, it repeats the bucketing process for each digit while preserving the previous step's ordering until all digits have been considered.
- Radix sort and bucket sort are almost equivalent; bucket sort goes from MSD to LSD, while radix sort is capable of both "direction" (LSD or MSD).





Linear time sorting– Radix Sort Algorithm

Suppose we have same the input array :

0.61	0.38	0.21	0.15	0.31	0.55	0.25	0.43

Lets start with LSD and sort array just based on LSD:

0.61	0.21	0.31	0.43	0.15	0.55	0.25	0.38

Jump to the next digit left:

0.15	0.21	0.25	0.31	0.38	0.43	0.55	0.61

Radix Sort Algorithm

- Radix Sort is a linear sorting algorithm.
- Radix Sort's time complexity of O(nd), where n is the size of the array and d is the number of digits in the largest number.
- It is not an in-place sorting algorithm because it requires extra space.
- Radix sort algorithm may be slower than other sorting algorithms such as merge sort and Quicksort if the operations are inefficient.
- Because it is based on digits or letters, radix sort is less flexible than other sorts. If the type of data changes, the Radix sort must be rewritten.

Questions

?