Parallel & Distributed Computing (WQD7008)

Week 11

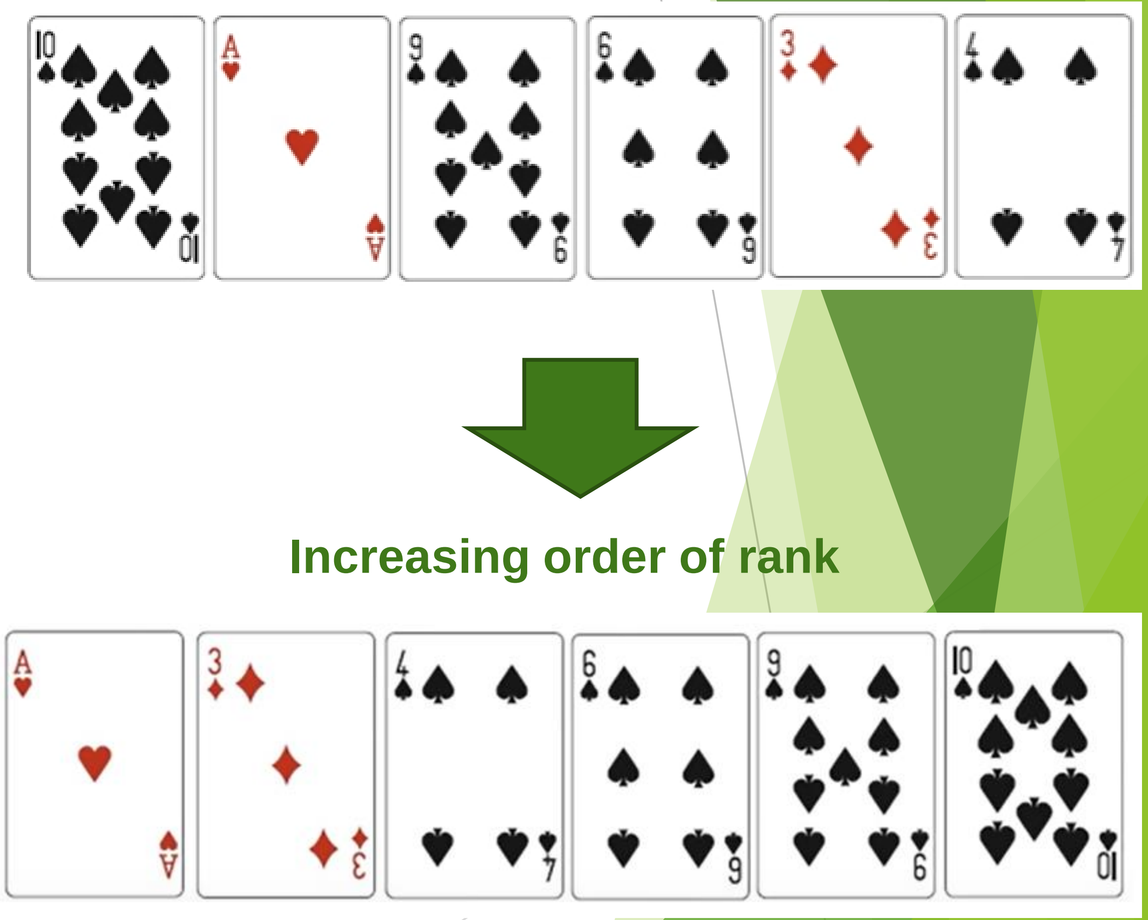
Sorting Algorithms

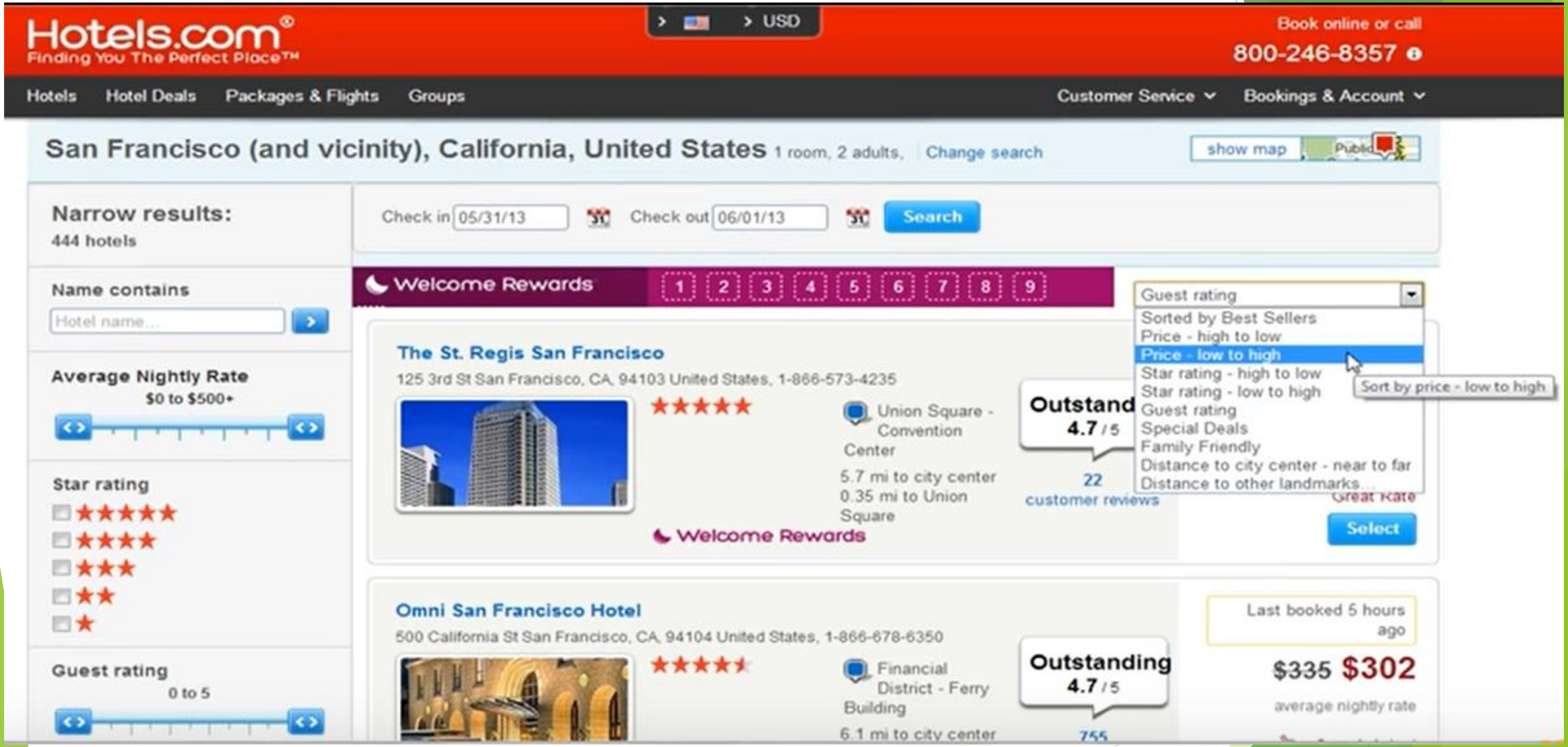
2019/2020 Semester 1

Dr. Hamid Tahaei

Introduction to sorting algorithms

- Practical application
  - People by last name
  - Gaming
  - Countries by population
  - Search engine results by relevance
  - …
- Fundamental to other algorithms




Hotels.com®
Finding You The Perfect Place™
USD
Book online or call
800-246-8357
Hotels
Hotel Deals
Packages & Flights
Groups
Customer Service
Bookings & Account
San Francisco (and vicinity), California, United States 1 room, 2 adults,
Change search
show map
Narrow results:
444 hotels
Check in 05/31/13
Check out 06/01/13
Search
Name contains
Hotel name...
Welcome Rewards
1 2 3 4 5 6 7 8 9
Guest rating
Sorted by Best Sellers
Price - high to low
Price - low to high
Star rating - high to low
Star rating - low to high
Guest rating
Special Deals
Family Friendly
Distance to city center - near to far
Distance to other landmarks...
Sort by price - low to high
Average Nightly Rate
$0 to $500+
The St. Regis San Francisco
125 3rd St San Francisco, CA, 94103 United States, 1-866-573-4235
Union Square - Convention Center
5.7 mi to city center
0.35 mi to Union Square
Outstand
4.7 / 5
22 customer reviews
Welcome Rewards
Select
Star rating
Omni San Francisco Hotel
500 California St San Francisco, CA, 94104 United States, 1-866-678-6350
Financial District - Ferry Building
6.1 mi to city center
Outstanding
4.7 / 5
755
Last booked 5 hours ago
$335 $302
average nightly rate
Guest rating
0 to 5

2012 Summer Olympics medal table[15]

Rank	NOC	Gold	Silver	Bronze	Total
1	United States (USA)	46	29	29	104
2	China (CHN)	38	27	23	88
3	Great Britain (GBR)*	29	17	19	65
4	Russia (RUS)	24	26	32	82
5	South Korea (KOR)	13	8	7	28
6	Germany (GER)	11	19	14	44
7	France (FRA)	11	11	12	34
8	Italy (ITA)	8	9	11	28
9	Hungary (HUN)	8	4	6	18
10	Australia (AUS)	7	16	12	35
11	Japan (JPN)	7	14	17	38
12	Kazakhstan (KAZ)	7	1	5	13
13	Netherlands (NED)	6	6	8	20
14	Ukraine (UKR)	6	5	9	20

Introduction to sorting algorithms

- Sorting is arranging the elements in a list or collection in increasing or decreasing order of some property
- Homogeneous (the same type)

- 2 , 3 , 9 , 4 , 6
- 2 , 3, 4 , 6 , 9 (increasing order of value)
- 9 , 6 , 4 , 3 , 2 (decreasing order of value)

- “fork” , “knife” , “mouse” , “screen” , “key”
- “fork” , “key” , knife” , “mouse” , “screen”

Introduction to sorting algorithms

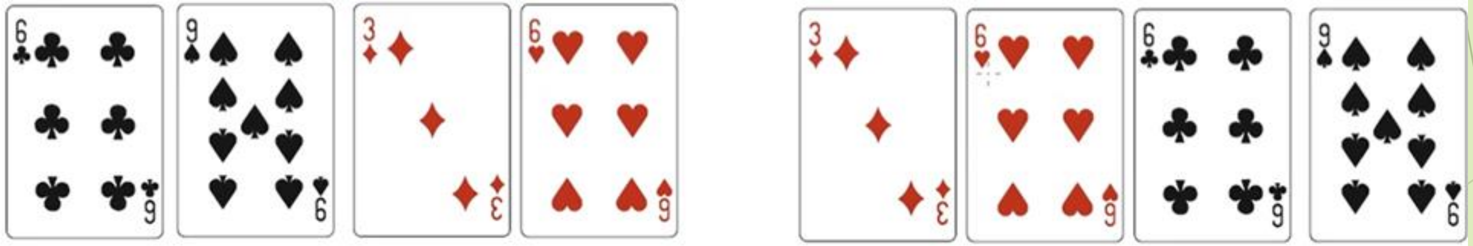
Sorting algorithms

- Bubble sort
- Selection sort
- Insertion sort
- Merge sort
- Quick sort
- Radix sort
- …

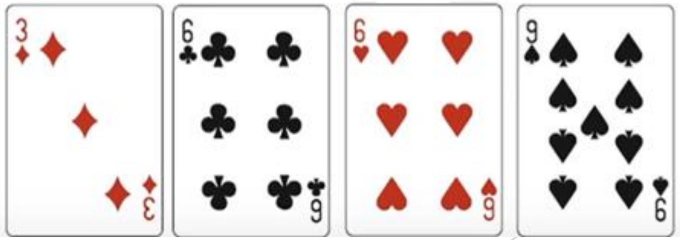
Introduction to sorting algorithms

Classification

- Time complexity
  - Measure of rate of growth of time taken by an algorithm with respect to input size
- Space complexity or memory usage
  - In-place – constant memory
  - Extra memory - Memory usage growth with input-size
- Stability



Insertion sort, Merge Sort, Bubble Sort, etc are stable by nature



like Heap Sort, Quick Sort, etc not stable

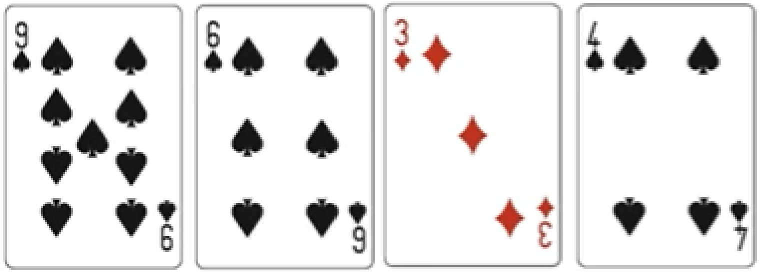
Introduction to sorting algorithms

Classification (continued)

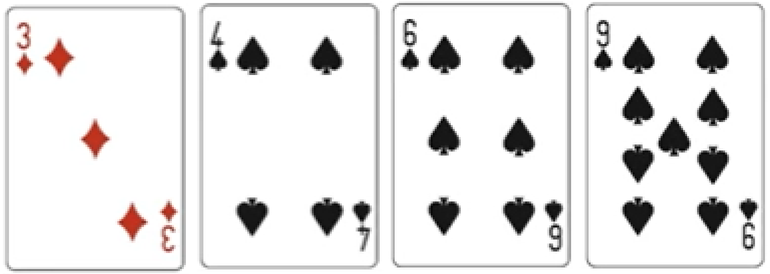
- Internal sort vs external sort
  - Internal → all the records are in main memory (RAM)
  - External → records on auxiliary store (disk/tapes)
- Recursive vs non-recursive
  - Recursive → quick sort , merge sort
  - Non-recursive → insertion sort, selection sort

Selection sort

Lest hand – unsorted



Right hand – sorted



- Find the minimum and put it in array B

A

~~2~~ MAX	~~7~~ MAX	~~4~~ MAX	~~1~~ MAX	~~5~~ MAX	~~3~~ MAX
0	1	2	3	4	5

B

1	2	3	4	5	7
0	1	2	3	4	5

Not In place

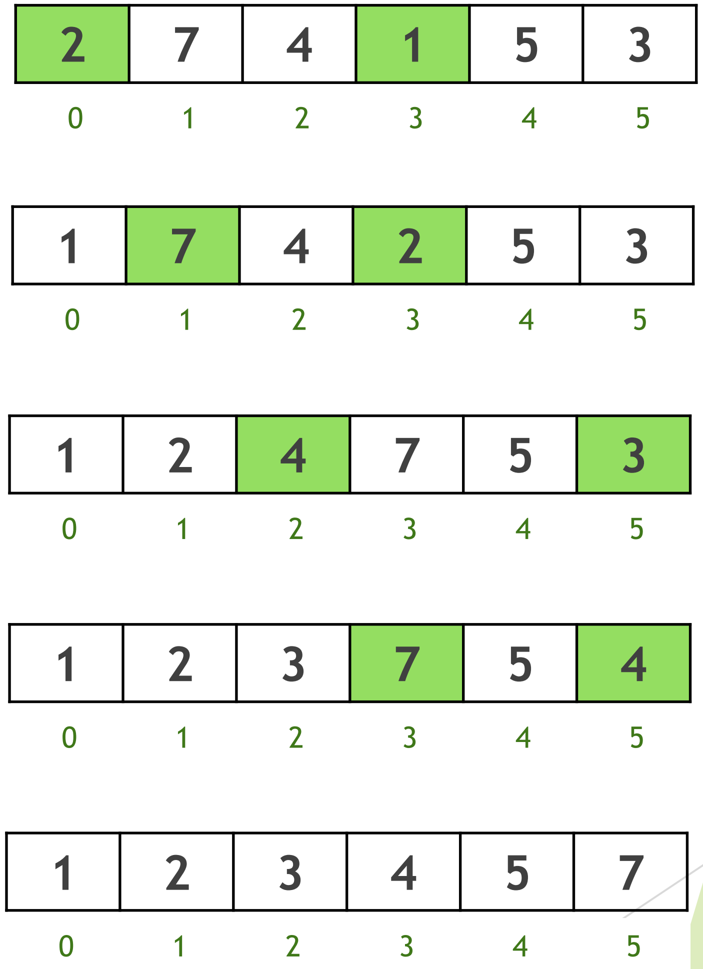
n-place sorting algorithm takes constant amount of extra memory

Selection sort

- In-place selection sorting algorithm

  Find the least(or greatest) value in the array, swap it into the leftmost(or rightmost) component, and then forget the leftmost component, Do this repeatedly.

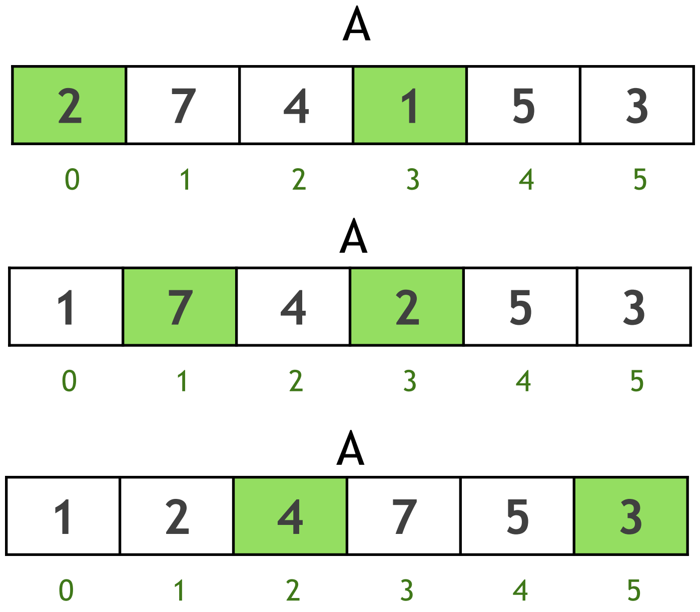
swap (element[min], element[i])



Selection sort

- In-place selection sorting algorithm

```
void SelectionSort(int A[],int n)
{
    for(int i = 0;i< n-1;i++)
    {
        int iMin = i;
        for(int j = i+1;j<n;j++)
        {
            if(A[j] < A[iMin])
                iMin = j;
        }
        int temp = A[i];
        A[i] = A[iMin];
        A[iMin] = temp;
    }
}
```



Selection sort

- Time Complexity
  - Inner loop executes (n-1) times when i=0, (n-2) times when i=1 and so on:
  - c_2= (n-1) + (n-2) + (n-3) + ….……. +2+1 = $\frac{n(n-1)}{2}$
  - Time complexity = $c_1 + c_2 + c_3$ =

$$(n-1)\ c_1 + \frac{n(n-1)}{2}.\ c_2 + (n-1)\ c_3 =$$

$$O(n^2)$$

```
void SelectionSort(int A[],int n)
{
      for(int i = 0;i< n-1;i++)
      {
c1          int iMin = i;
n-1         for(int j = i+1;j<n;j++)
            {
          c2    if(A[j] < A[iMin])
                     iMin = j;
            }
            int temp = A[i];
c3          A[i] = A[iMin];
n-1         A[iMin] = temp;
      }
}
```

- Space Complexity
  - Since no extra space beside n variables is needed for sorting so:

$$O(n)$$

Bubble sort

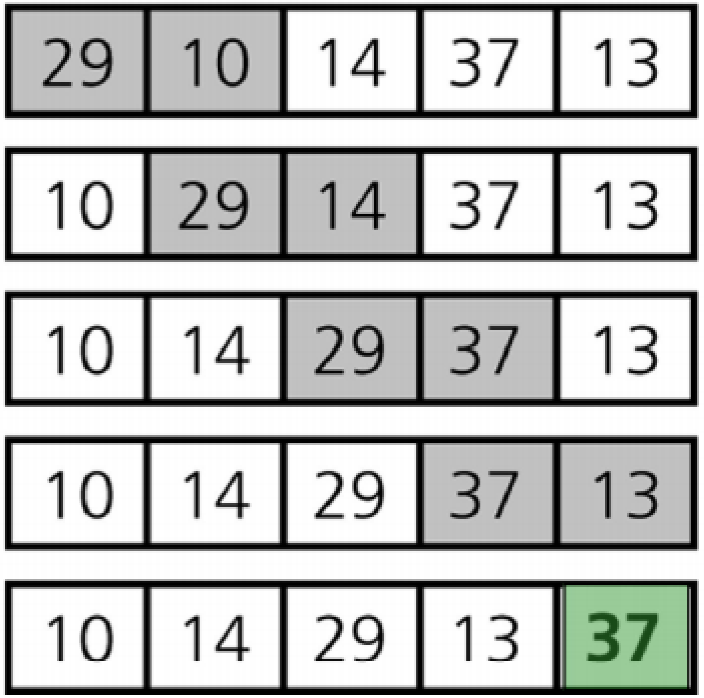
Given an array of n items

- Compare pair of adjacent items
- Swap if the items are out of order
- Repeat until the end of array
  - The largest item will be at the last position
- Reduce n by 1 and go to Step 1

- Analogy
  - Large item is like “bubble” that floats to the end of the array

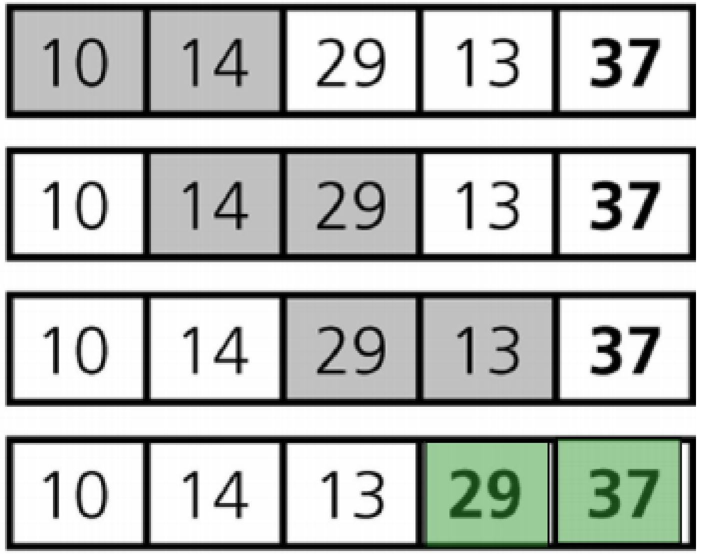
Bubble sort

(a) Pass 1

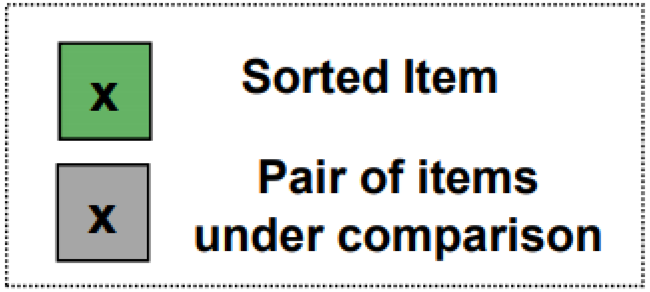


At the end of **Pass 1**, the largest item **37** is at the last position.

(b) Pass 2



At the end of **Pass 2**, the second largest item **29** is at the second last position.



Bubble sort

- Time Complexity

  - c_1= n-1
  - c_2= (n-1) + (n-2) + (n-3) + ………. +2+1 = $\frac{n(n-1)}{2}$
  - Time complexity = $c_1 + c_2$ =

    (n-1) c_1 + $\frac{n(n-1)}{2}$. c_2 =

    O(n^2)

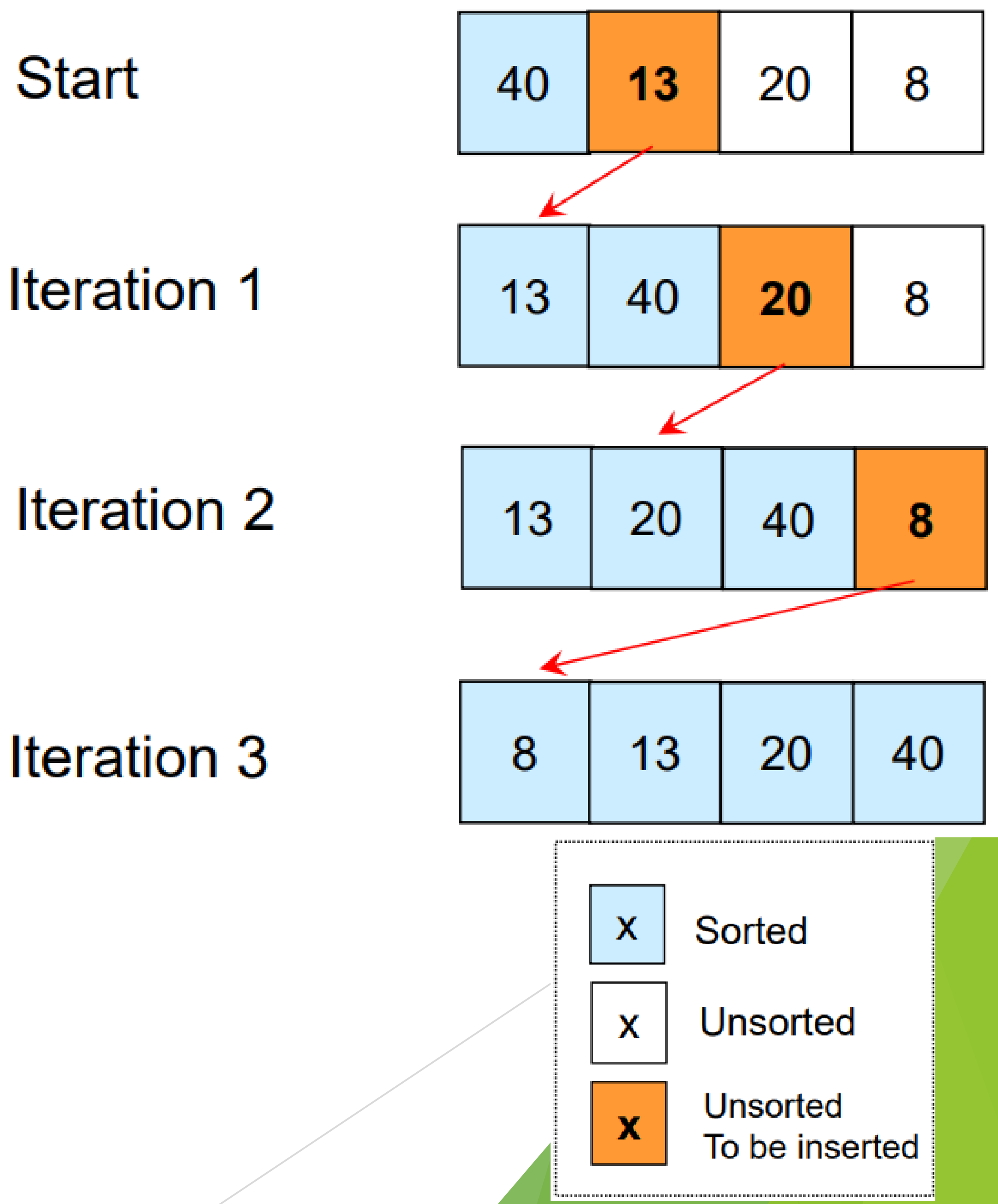
- Worst Case Time Complexity [Big-O]: **O(n²)**
- Best Case Time Complexity [Big-omega]: **O(n)**
- Average Time Complexity [Big-theta]: **O(n²)**
- Space Complexity: **O(1)**

```
void bubbleSort(int array[], int size)
{
  for (int step = 0; step < size - 1; ++step)
  {
    for (int i = 0; i < size - step - 1; ++i)
    {
      if (array[i] > array[i + 1])
      {
        int temp = array[i];
        array[i] = array[i + 1];
        array[i + 1] = temp;
      }
    }
  }
}
```

c_1 c_2

Insertion sort

- One of the most straight forward sorting algorithm
- Similar to how most people arrange a hand of poker cards
- Start with one card in your hand
- Pick the next card and insert it into its proper sorted order
- Repeat previous step for all cards



Insertion sort

- Outer-loop executes (n−1) times
- Number of times inner-loop is executed depends on the input
- Best-case: the array is already sorted and (a[j] > next) is always false
- No shifting of data is necessary
- Worst-case: the array is reversely sorted and (a[j] > next) is always true
- Insertion always occur at the front
- Therefore, the best-case time is O(n)
- And the worst-case time is O(n^2)

```
for i : 1 to length(A) -1
    j = i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j = j - 1
```

Merge sort

- Suppose we only know how to merge two sorted sets of elements into one
- Merge {1, 5, 9} with {2, 11} → {1, 2, 5, 9, 11}

Question

- Where do we get the two sorted sets in the first place?
- Idea (use merge to sort n items)
- Merge each pair of elements into sets of 2
- Merge each pair of sets of 2 into sets of 4
- Repeat previous step for sets of 4 …
- Final step: merge 2 sets of n/2 elements to obtain a fully sorted set

Merge sort

Divide-and-Conquer Method

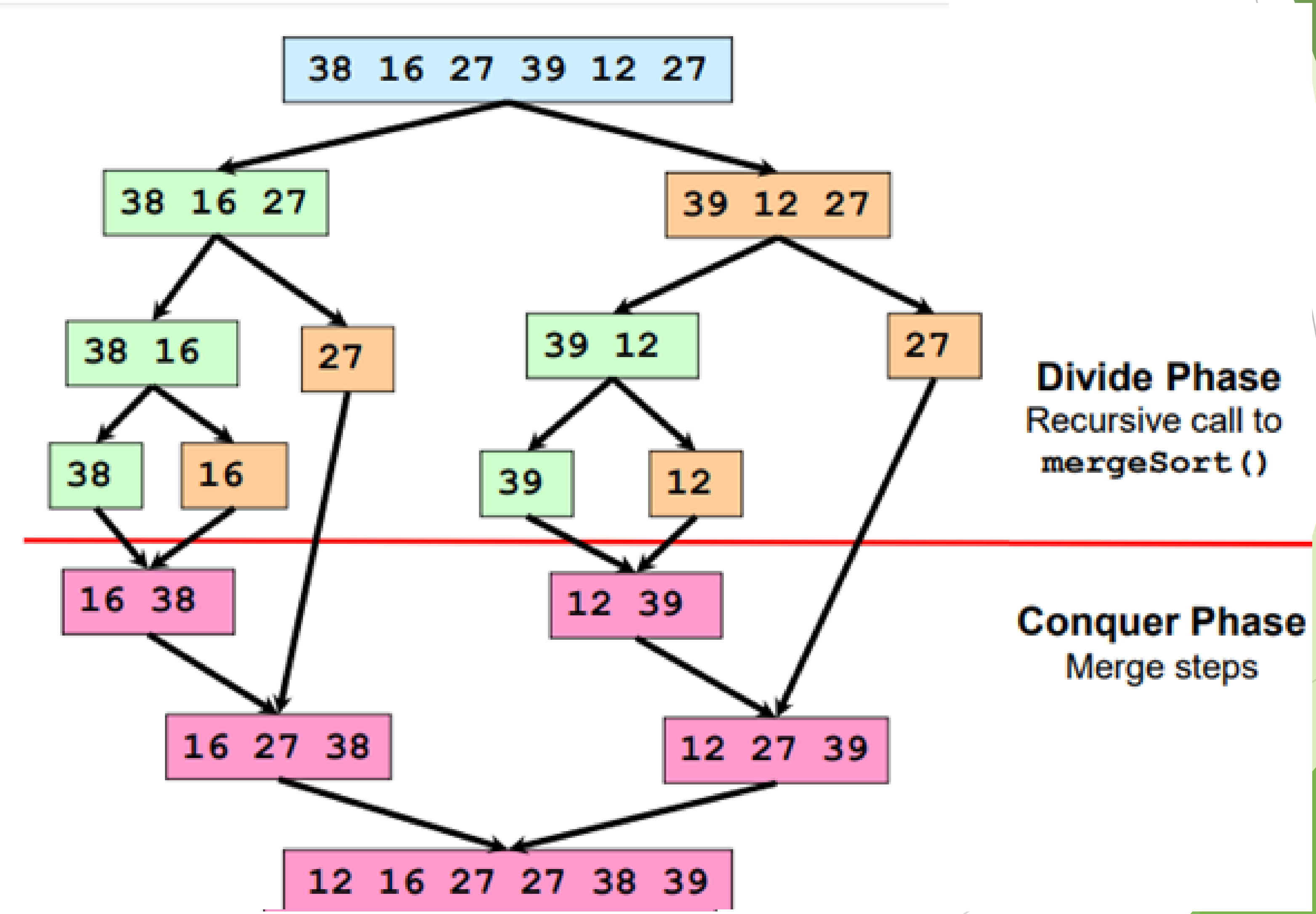
- A powerful problem solving technique
- Divide-and-conquer method solves problem in the following steps
- Divide step
  - Divide the large problem into smaller problems
  - Recursively solve the smaller problems
- Conquer step
  - Combine the results of the smaller problems to produce the result of the larger problem

Merge sort

Divide-and-Conquer: Merge Sort

- Merge Sort is a divide-and-conquer sorting algorithm
- Divide step
  - Divide the array into two (equal) halves
  - Recursively sort the two halves
- Conquer step
  - Merge the two halves to form a sorted array

Merge sort


38 16 27 39 12 27
38 16 27
39 12 27
38 16
27
39 12
27
38
16
39
12
16 38
12 39
16 27 38
12 27 39
12 16 27 27 38 39
Divide Phase
Recursive call to
mergeSort()
Conquer Phase
Merge steps

Merge sort

```
mergesort (array a)
    if ( n == 1 )
      return a

    arrayOne = a[0] ... a[n/2]
    arrayTwo = a[n/2+1] ... a[n]

    arrayOne = mergesort ( arrayOne )
    arrayTwo = mergesort ( arrayTwo )

    return merge ( arrayOne, arrayTwo )
```

time complexity O(nlogn)

```
merge ( array a, array b )
    array c

    while ( a and b have elements )
        if ( a[0] > b[0] )
            add b[0] to the end of c
            remove b[0] from b
        else
            add a[0] to the end of c
            remove a[0] from a

    // At this point either a or b is empty

    while ( a has elements )
        add a[0] to the end of c
        remove a[0] from a

    while ( b has elements )
        add b[0] to the end of c
        remove b[0] from b

    return c
```

Merge sort

Pros

- The performance is guaranteed, i.e. unaffected by original ordering of the input
- Suitable for extremely large number of inputs
- Can operate on the input portion by portion

Cons

- Not easy to implement
- Requires additional storage during merging operation
- O(n) extra memory storage needed
- Not In-place algorithm

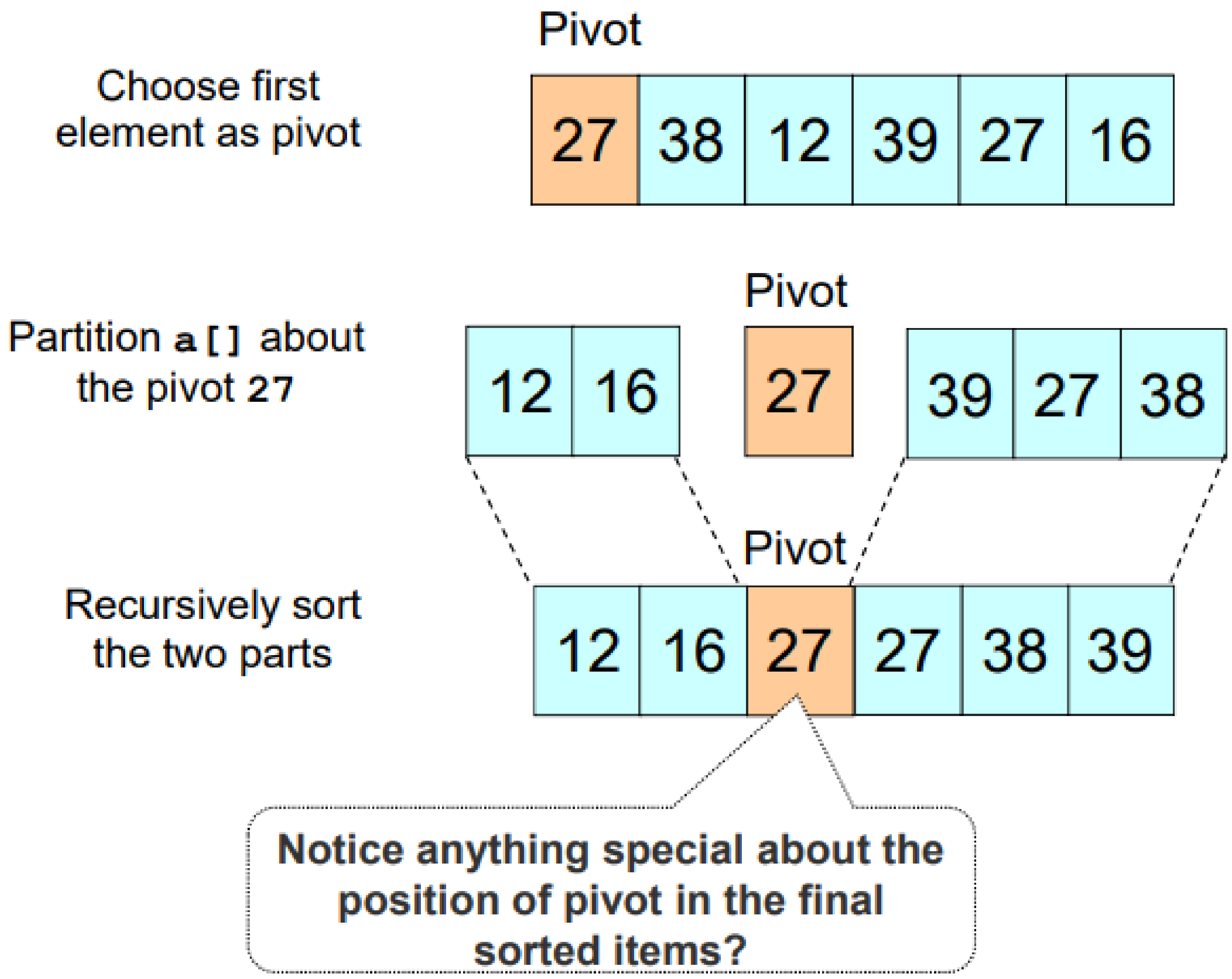
Quick sort

Quick Sort is a divide-and-conquer algorithm

- Divide step
  - Choose an item Choose an item p (known as (known as pivot) and partition the) and partition the items of a[i...j] into two parts
  - Items that are smaller than p
  - Items that are greater than or equal to p
  - Recursively sort the two parts
- Conquer step
  - Do nothing!

In comparison, Merge Sort spends most of the time in conquer step but very little time in divide step

Quick sort: Divide Step



Quick sort

Example:

7	2	1	6	8	5	3	4

Average case: O(nlogn)

Worst case: O(n^2)

In-place algorithm

Radix sort

- Treats each data to be sorted as a character string
- It is not using comparison, i.e. no comparison between the data is needed
- In each iteration
- Organize the data into groups according to the next character in each data
- The groups are then “concatenated” for next iteration

Radix sort

Original Integer	53	89	150	36	366	233
Grouped by the third digit	150	53	633	233	36	89
Grouped by the second digit	633	233	36	150	53	89
Grouped by the first digit	36	53	89	150	233	633

Radix sort

- For each iteration
- We go through each item once to place them into group
- Then go through them again to concatenate the groups
- Complexity is O(n)
- Number of iterations is d, the maximum number of digits (or maximum number of characters)
- Complexity is thus O(dn)

- Radix sort is an In-place and stable sorting algorithm

Time complexities: Summary

	Worst Case	Best Case	In-place?	Stable?
Selection Sort	$O(n^2)$	$O(n^2)$	Yes	No
Insertion Sort	$O(n^2)$	$O(n)$	Yes	Yes
Bubble Sort	$O(n^2)$	$O(n^2)$	Yes	Yes
Bubble Sort 2	$O(n^2)$	$O(n)$	Yes	Yes
Merge Sort	$O(n \lg n)$	$O(n \lg n)$	No	Yes
Quick Sort	$O(n^2)$	$O(n \lg n)$	Yes	No
Radix sort	$O(dn)$	$O(dn)$	No	yes

Summary

Comparison-Based Sorting Algorithms

- Iterative Sorting
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
- Recursive Sorting
  - Merge Sort
  - Quick Sort

Non-Comparison Comparison-Based Sorting Algorithms Based Sorting Algorithms

  - Radix Sort

Properties of Sorting Algorithms

  - In-Place
  - Stable

Parallel sort

Parallel Sorts

Odd-Even Transposition Sort

- Odd-Even Transposition Sort is a parallel version of bubble sort

9	7	3	8	5	6	4	1	Phase 1(Odd)
7	9	3	8	5	6	1	4	Phase 2(Even)
7	3	9	5	8	1	6	4	Phase 3(Odd)
3	7	5	9	1	8	4	6	Phase 4(Even)
3	5	7	1	9	4	8	6	Phase 5(Odd)
3	5	1	7	4	9	6	8	Phase 6(Even)
3	1	5	4	7	6	9	8	Phase 7(Odd)
1	3	4	5	6	7	8	9	

Odd-Even Transposition Sort

Time Complexity

- $O(n^2)$

```
void oddEvenSort(int arr[], int n)
{
    bool isSorted = false; // Initially array is unsorted

    while (!isSorted) {
        isSorted = true;

        // Perform Bubble sort on odd indexed element
        for (int i = 1; i <= n - 2; i = i + 2) {
            if (arr[i] > arr[i + 1]) {
                swap(arr[i], arr[i + 1]);
                isSorted = false;
            }
        }

        // Perform Bubble sort on even indexed element
        for (int i = 0; i <= n - 2; i = i + 2) {
            if (arr[i] > arr[i + 1]) {
                swap(arr[i], arr[i + 1]);
                isSorted = false;
            }
        }
    }

    return;
}
```

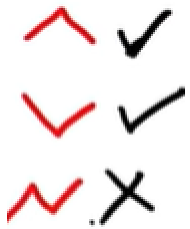
Bitonic Sort

- A parallel sorting algorithm which performs O(n2 log n) comparisons.
- It performs better for the parallel implementation because elements are compared in predefined sequence which must not be depended upon the data being sorted.
- The predefined sequence is called Bitonic sequence

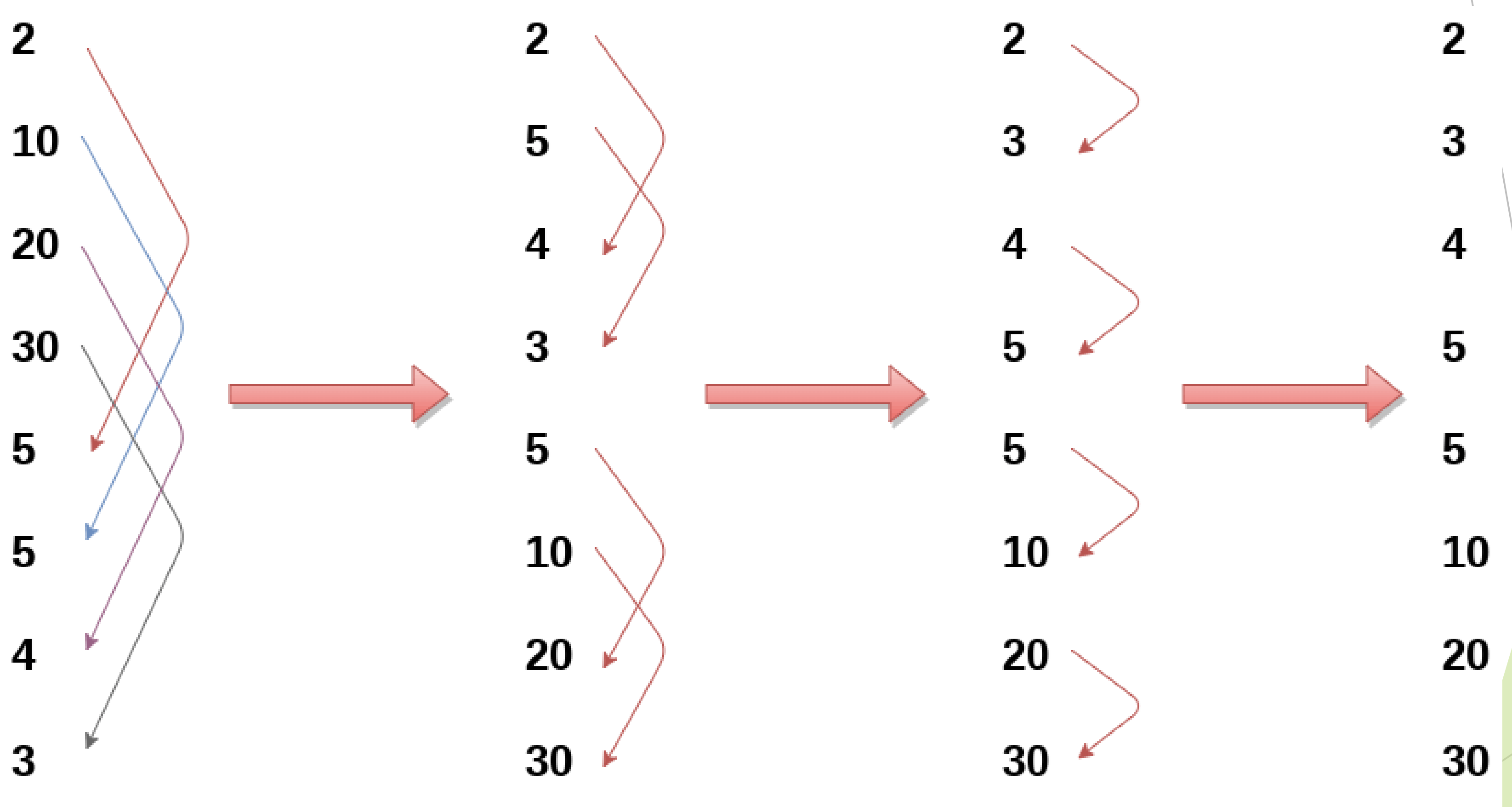
to understand Bitonic sort

- Bitonic sequence is the one in which, the elements first comes in increasing order then start decreasing after some particular index. An array A[0... i ... n-1] is called

$$A[0] < A[1] < A[2] \ldots. A[i-1] < A[i] > A[i+1] > A[i+2] > A[i+3] > \ldots > A[n-1]$$



Bitonic Sort



Complexity	Best Case	Average Case	Worst Case
Time Complexity	O(log ^{2} n)	O(log ^{2} n)	O(log ^{2} n)

Thank you for your patience