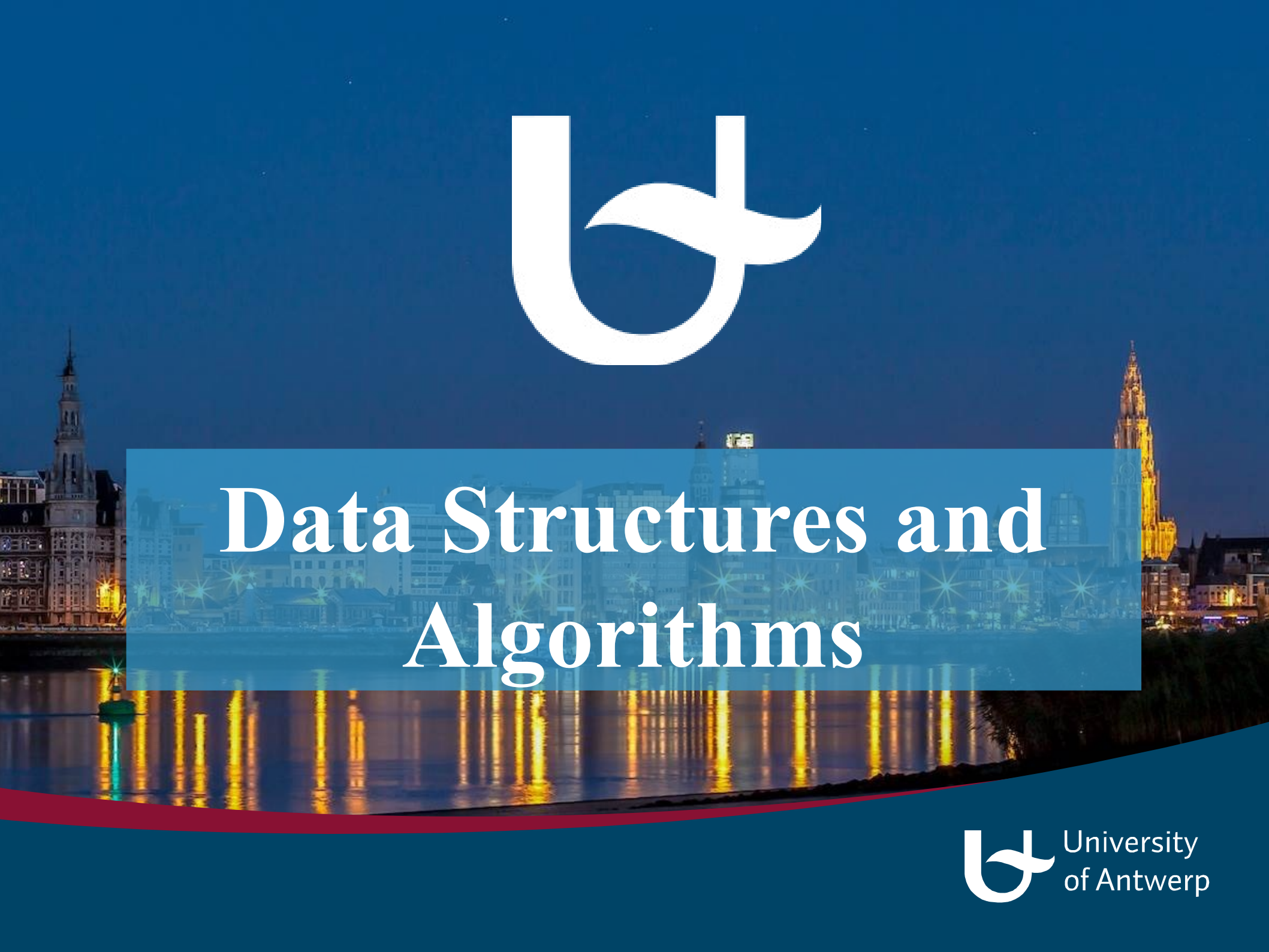


Data Structures and Algorithms

University of Antwerp

Searching and Sorting

Lecturer: Dr. Ali Anwar

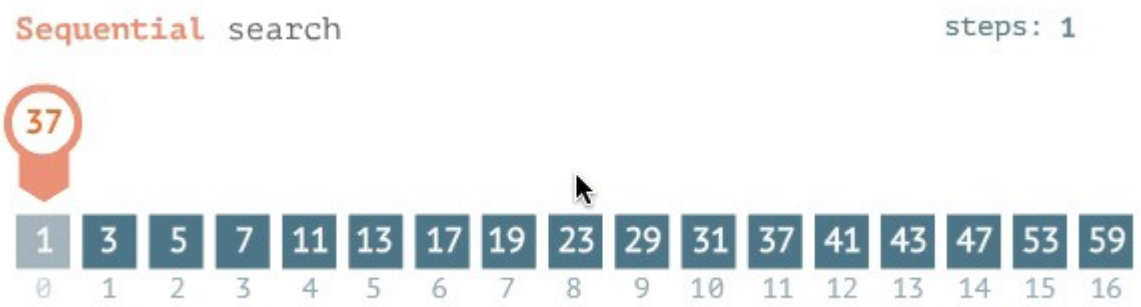
Objectives

Introduction of Algorithms:

- **Searching**
  - Linear search
  - Binary search
- **Sorting**
  - Merge sort
  - Quick sort
  - Bubble sort
  - Insertion sort
  - Selection sort

Part 1

Searching Algorithms



Example of Linear Search

Given an array a and integer x, find an integer i such that

1. *if there is no j such that $a[j]$ is x, then i is -1,*
2. *otherwise, i is any j for which $a[j]$ is x.*

Meaning:

1) if x does not occur in the array *a*, then *i* should be -1

2) if it does occur - then *i* should be a position where it occurs.

Example of Linear Search (cont.)

1. Start from the leftmost element of array and one by one compare *x* with each element of *a*
2. If *x* matches with an element, return the index
3. If *x* doesn't match with any of elements, return -1.
4. Complexity is O(n)

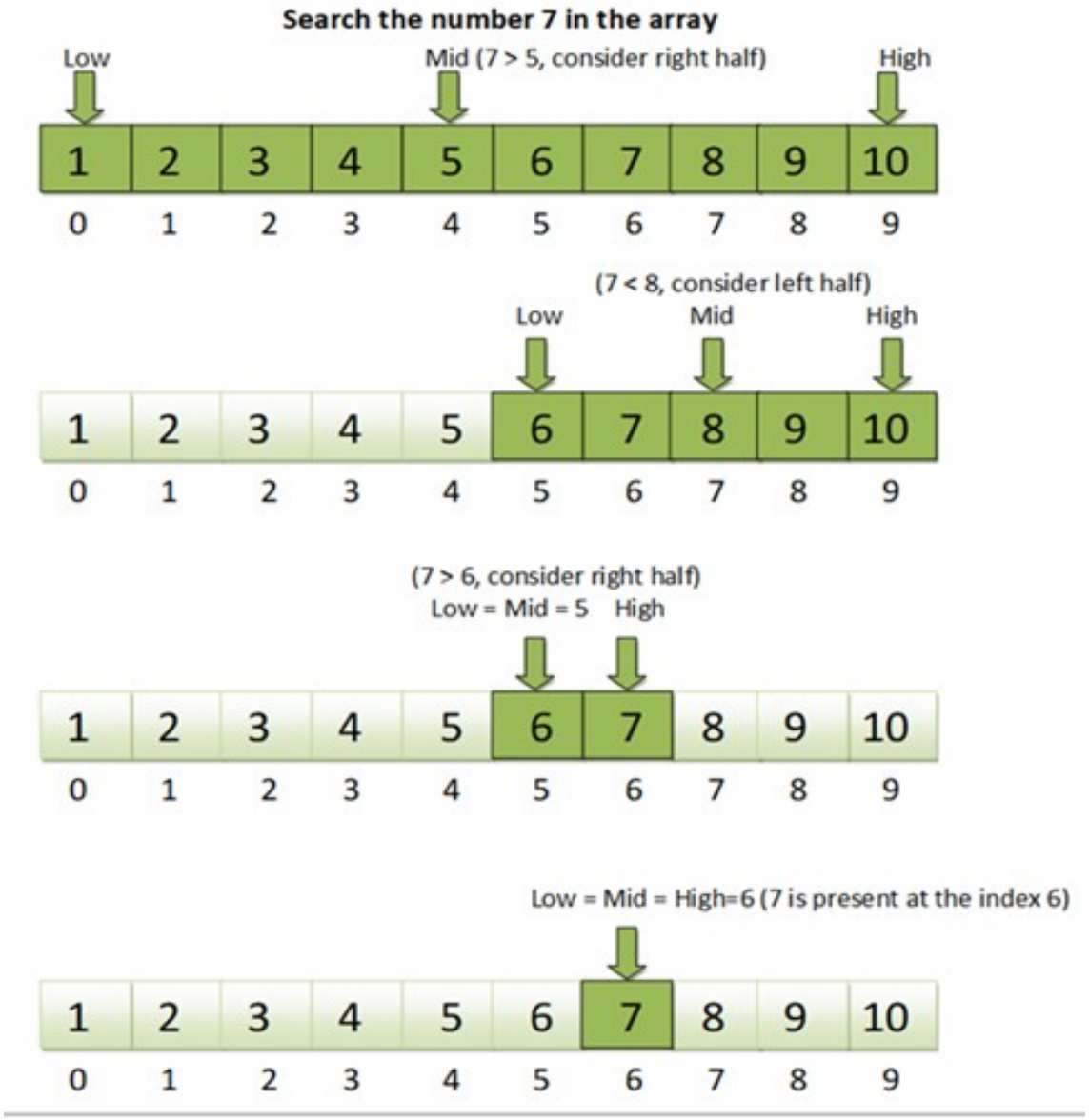
```
#include <iostream>
using namespace std;

int search(int arr[], int n, int x)
{
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}
```

Binary Search

Binary Search works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed down the possible locations to just one.

- Complexity is $n / 2^k = 1$ or O(logn)



C++ Implementation of Binary Search

```
int binarySearch(int array[], int x, int low, int high)
{
        while (low <= high)
        {
                int mid = low + (high - low) / 2;

                if (array[mid] == x)
                        return mid;

                if (array[mid] < x)
                        low = mid + 1;

                else
                        high = mid - 1;
        }

        return -1;
}
```

Part 2

Sorting Algorithms

Sorting

- Sorting is arranging the elements in a list or collection in increasing or decreasing order of some property
- The list should be homogenous (all the elements in the list should be of same type)
- Example:
  - Input: 2, 3, 9, 4, 6
  - Output:
    - 2, 3, 4, 6, 9 (increasing order)
    - 9, 6, 4, 3, 2 (decreasing order)
    - 2, 3, 9, 4, 6 (increasing order of number of factors)

Sorting (contd.)

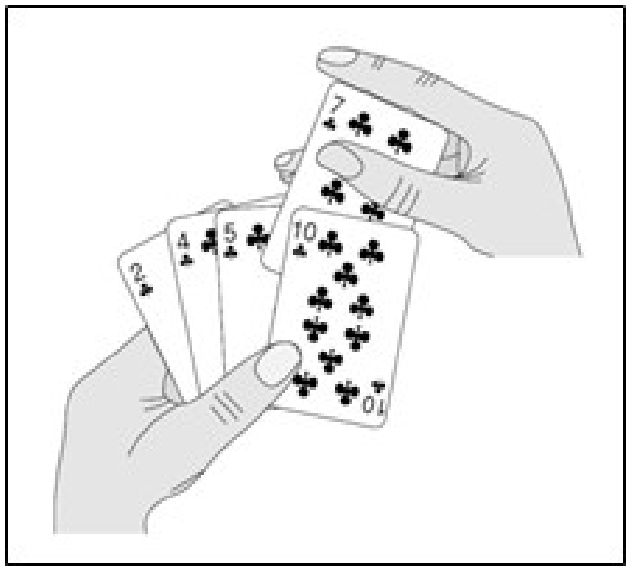
- Input to the sorting algorithms:
  - a collection of records stored in an array
- Some examples include:
  - Search results on websites
  - Language Dictionaries

Analyzing Sorting Algorithms

- **Running time**: measure the number of comparisons made between keys.
- In some situations: measure the number of swap operations, when the records are “large”

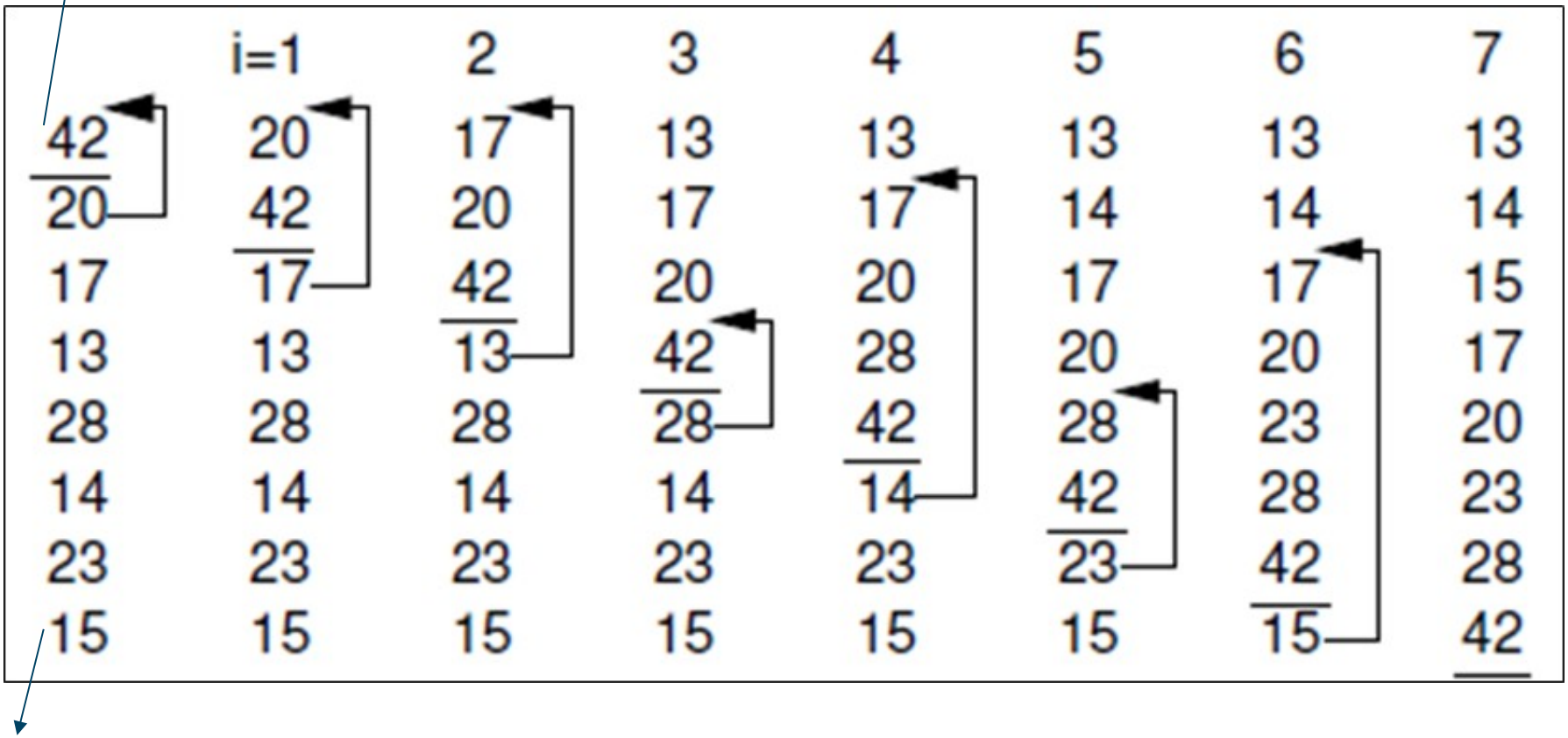
Insertion Sort

- Real life example: when assembling playing cards.
- Insertion sort iterates through a list of records.
- Each record is inserted in turn at the correct position within a sorted list composed of those records already processed.



Example of Insertion Sort

Unsorted set



Sorted set

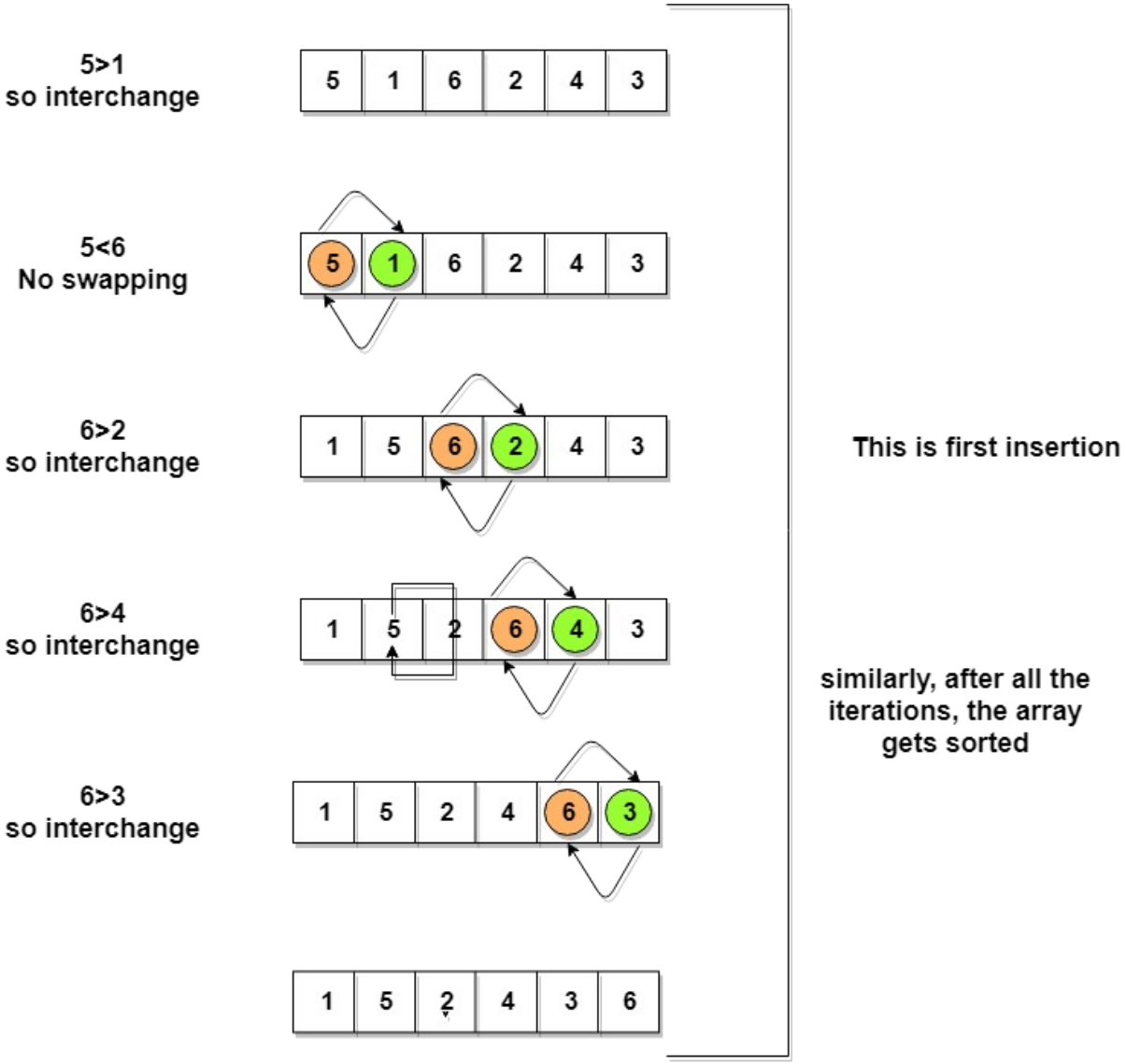
Insertion Sort – Running Time

- Worst case: $O(n^2)$
  - In case of descending sorted array => n scans and n swaps
  - Requires two for loops for sorted and unsorted sets
- Best case or Almost Sorted case: O(n)
  - In case of ascending sorted array => n scans only
- Average case for a random array: $O(n^2)$

Bubble Sort

- We scan the array from left to right multiple times
  - Each time the scan happens, it is called a pass
- Algorithm:
  - First iteration of the inner for loop moves through the record array from bottom to top, comparing adjacent keys
  - The second pass through the array repeats this process, but not to the top (-1)
  - → elements ‘bubble’ to the top

Example of Bubble Sort



Source: https://bit.ly/34sO15r

Bubble Sort – C++ Code

- Algorithm compares each item with every other item in some list
- Time complexity $O(n^2)$

```
for ( i = 1 ; i < n ; i++ )
   for ( j = n-1 ; j >= i ; j-- )
      if ( a[j] < a[j-1] )
         swap a[j] and a[j-1]
```

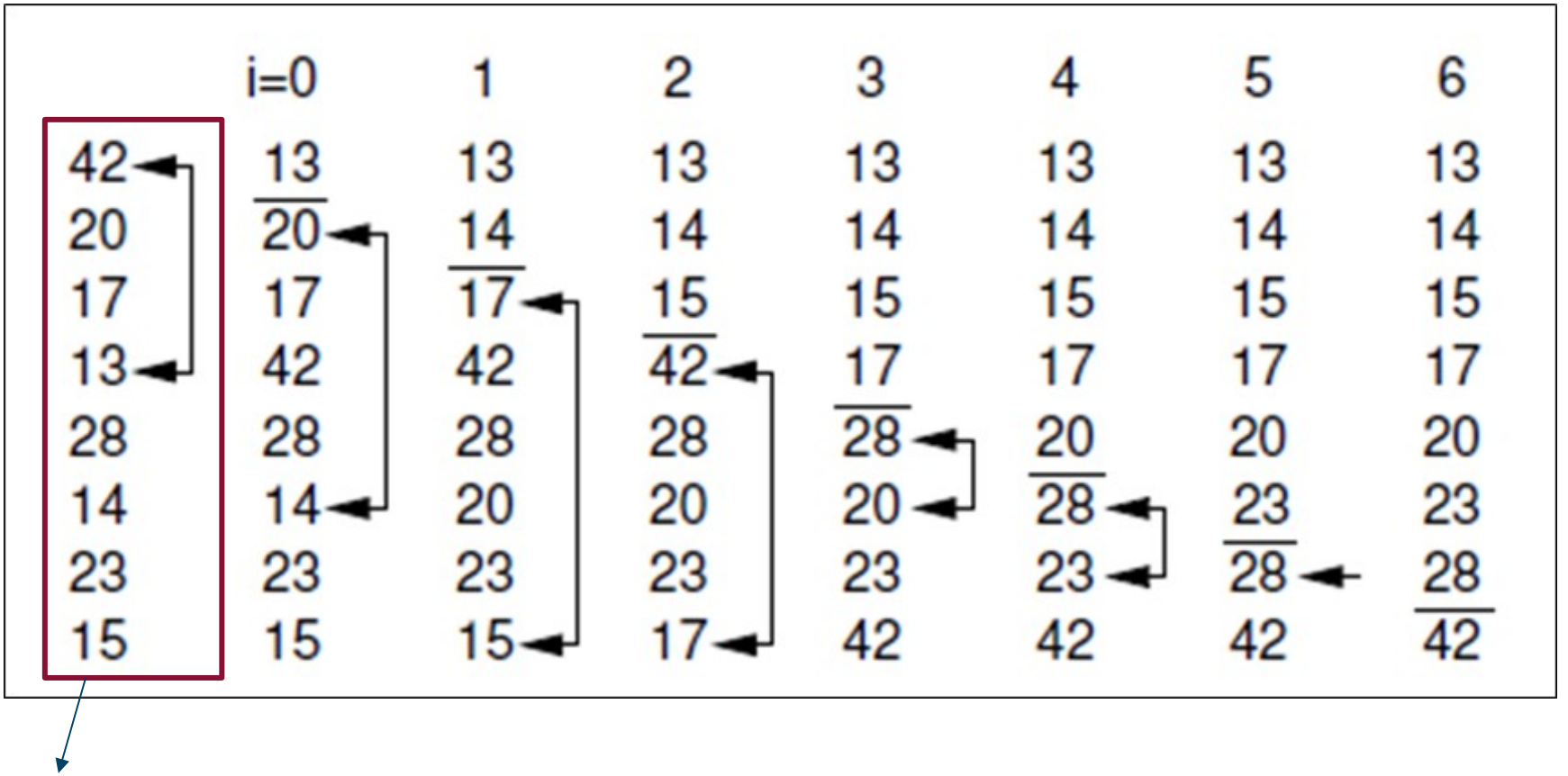
Bubble Sort – Running Time

- outer loop: n - 1
- inner loop: n - *1*
- running time: $(n-1)^2$
- # swaps ~ $O(n^2)$

Selection Sort

- Selection Sort:
  - first finds the smallest key in an unsorted list and places that value on top
  - then repeats with the second smallest, and so on.
- Algorithm: The *i*-th pass of selection sort "selects" the *i*-th smallest key in the array, placing that record into position *i*.

Example of Selection Sort



Scan the element for minimum, compare with top and swap

Selection Sort – Running Time

- Just like in bubble sort:
  - running time (#compares) ~ $O(n^2)$
- But we remember the position of the ‘next smallest’ element, and perform one swap
  - useful when the cost of swap is high

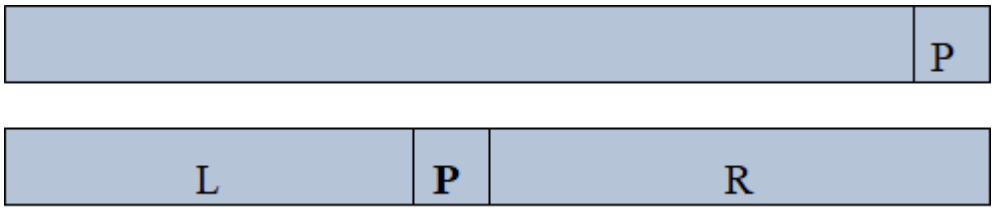
Quick Sort

- Typical "divide and conquer" strategy:
  - *Divide*: divide the input data S in two disjoint subsets S_1 and S_2
  - *Recur*: solve the subproblems associated with S_1 and S_2
  - *Conquer*: combine the solutions for S_1 and S_2 into a solution for S

- Fastest general-purpose in-memory sorting algorithm in the average case & space efficient.
- It is most practical sorting algorithm because of its efficiency

Quick Sort Algorithm

- Pick an element, called a pivot, from the array.
  - *typically: middle element (or first, last)*
- Partitioning: re-order the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it. After this partitioning, the pivot is in its final position. This is called the partition operation.



- Recursively apply the above steps to Left & Right subarray.

QuickSort in Psuedocode

```
// Sorts a (portion of an) array, divides it into partitions, then sorts those
algorithm quicksort(A, lo, hi) is
  // Ensure indices are in correct order
  if lo >= hi || lo < 0 then
    return

  // Partition array and get the pivot index
  p := partition(A, lo, hi)

  // Sort the two partitions
  quicksort(A, lo, p - 1) // Left side of pivot
  quicksort(A, p + 1, hi) // Right side of pivot
```

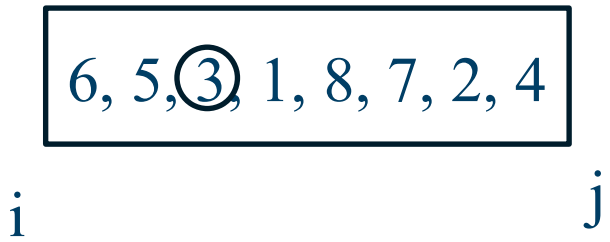
Source: https://en.wikipedia.org/wiki/Quicksort

Simulation of QuickSort

- We start with an array A
- We select a Pivot
- We initialize the iterators
- We do conditional iteration of i and j
- We do recursion

do i++ while A[i] < pivot
do j– while A[j] > pivot

if i>=j then return j
swap A[i] with A[j]

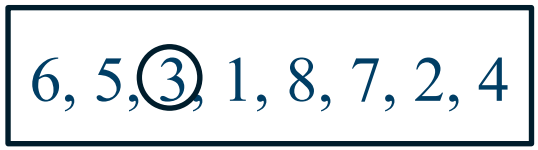


Simulation of QuickSort

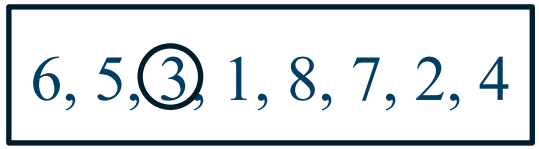
do i++ while A[i] < pivot
do j– while A[j] > pivot

if i>=j then return j
swap A[i] with A[j]

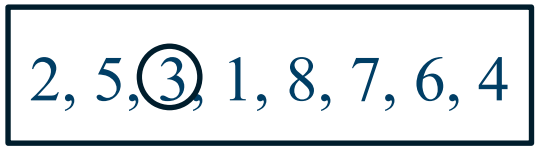
Step 1



Step 2



//swap

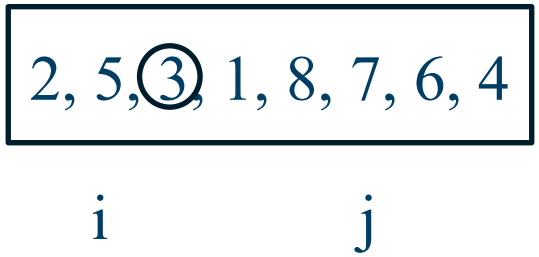


Simulation of QuickSort

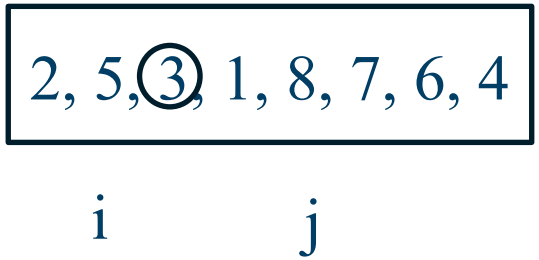
do i++ while A[i] < pivot
do j– while A[j] > pivot

if i>=j then return j
swap A[i] with A[j]

Step 3



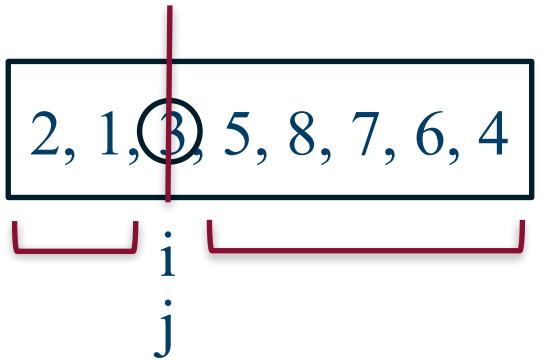
Step 4



Step 5

Simulation of QuickSort

Step 6



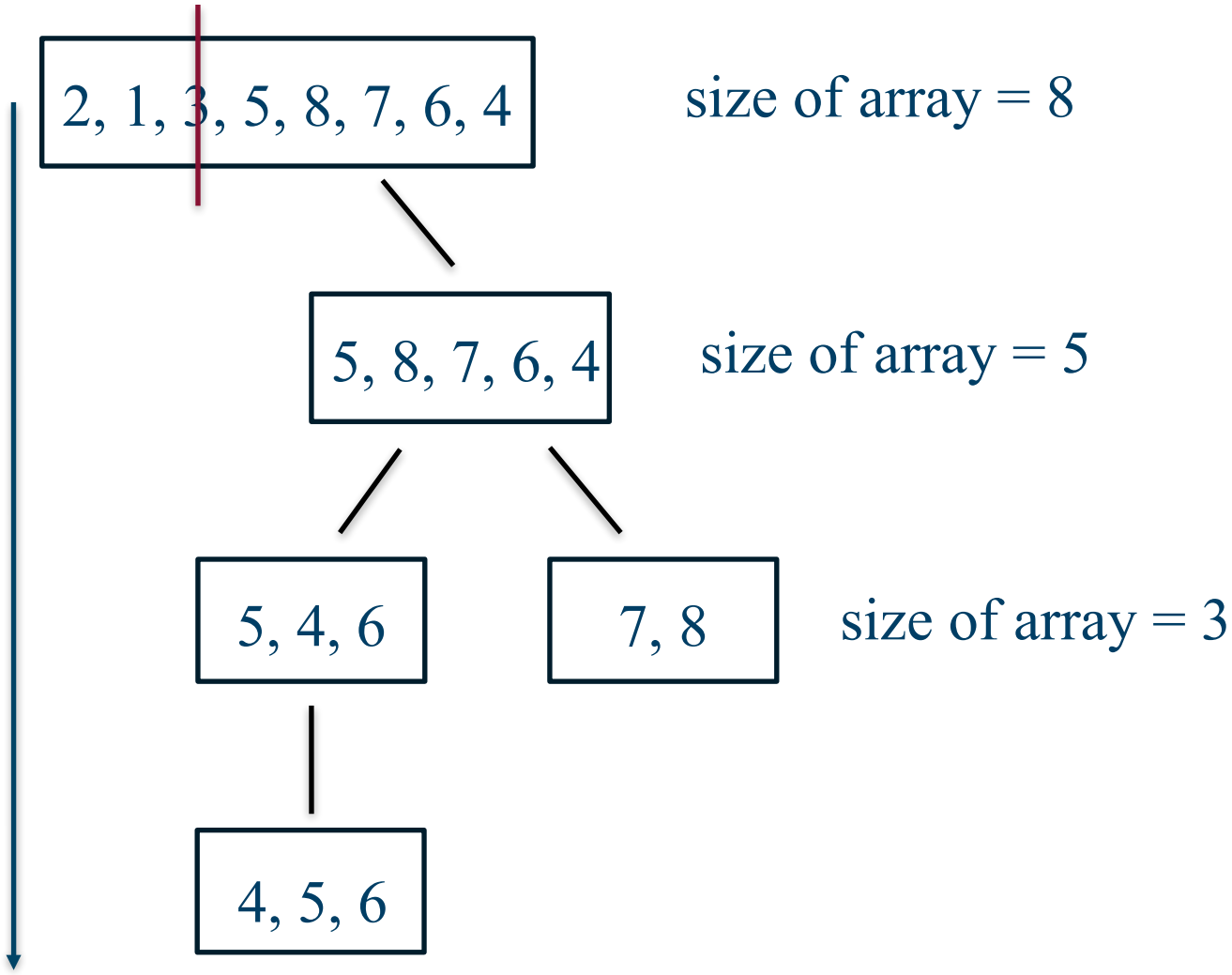
do i++ while A[i] < pivot
do j– while A[j] > pivot

if i>=j then return j
swap A[i] with A[j]

- Iterators i and j have crossed, partition at pivot
- Notice that all the elements greater than pivot are on the right side, and the elements smaller on left side
- Apply QuickSort on left- and right-hand sides recursively

Analysis of QuickSort

At each partition,
the array divides into half
upon which
we do n swaps in worst case



Quick Sort – C++ Code

- Algorithm picks a pivot and move all smaller items before it, while all greater elements after it (=partition), recursively repeated on lesser and greater sub-lists until their size is one or zero

- Time complexity O($n\ log(n)$) – for best case = pivot partitions exactly at the middle

```
quicksort(array a, int left, int right) {
   if ( left < right ) {                          c1
      pivotindex = partition(a,left,right)        n
      quicksort(a,left,pivotindex-1)              n/2
      quicksort(a,pivotindex+1,right)             n/2
   }
}
```

Quick Sort – Running Time

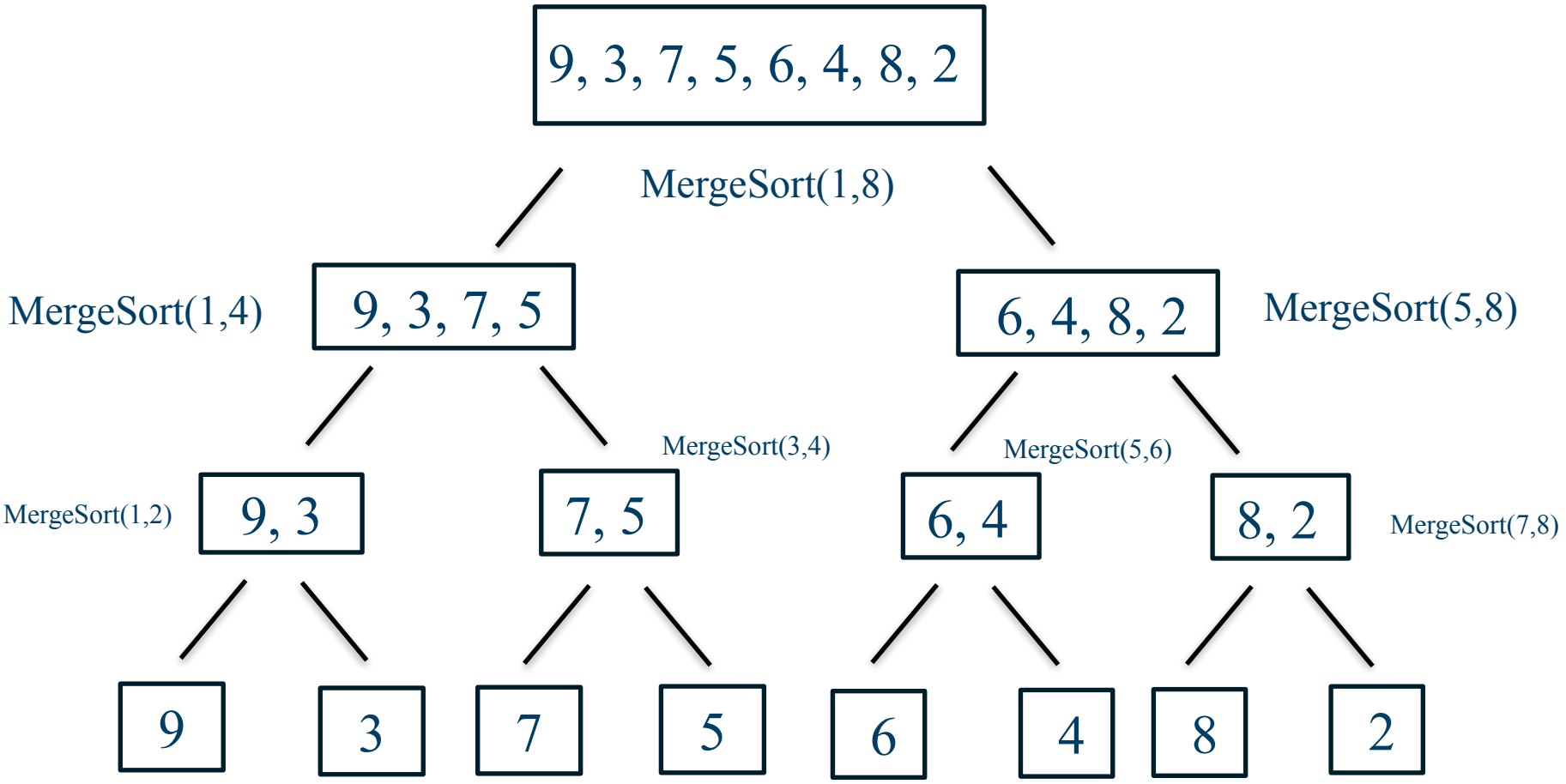
- Worst case: occurs when 'pivot' does a poor job of breaking the array (unbalanced array)
  - $O(n^2)$
- Best case: occurs when 'pivot' always breaks the array into two equal halves
  - *log n* levels for partitioning, each level *n/2*
  - $O(n \log n)$

Merge Sort

- Similar to Quick Sort, but does not do in place swaps (higher space complexity)
- Algorithm splits a list into two similar sized lists (left and right) and sorts each list and then merges the sorted lists back together
- Time complexity O(n *log*(n)) – Worst Case

```
mergesort(array a, int left, int right) {
   if ( left < right ) {
      mid = (left + right) / 2
      mergesort(a, left, mid)
      mergesort(a, mid+1, right)
      merge(a, left, mid, right)
   }
}
```

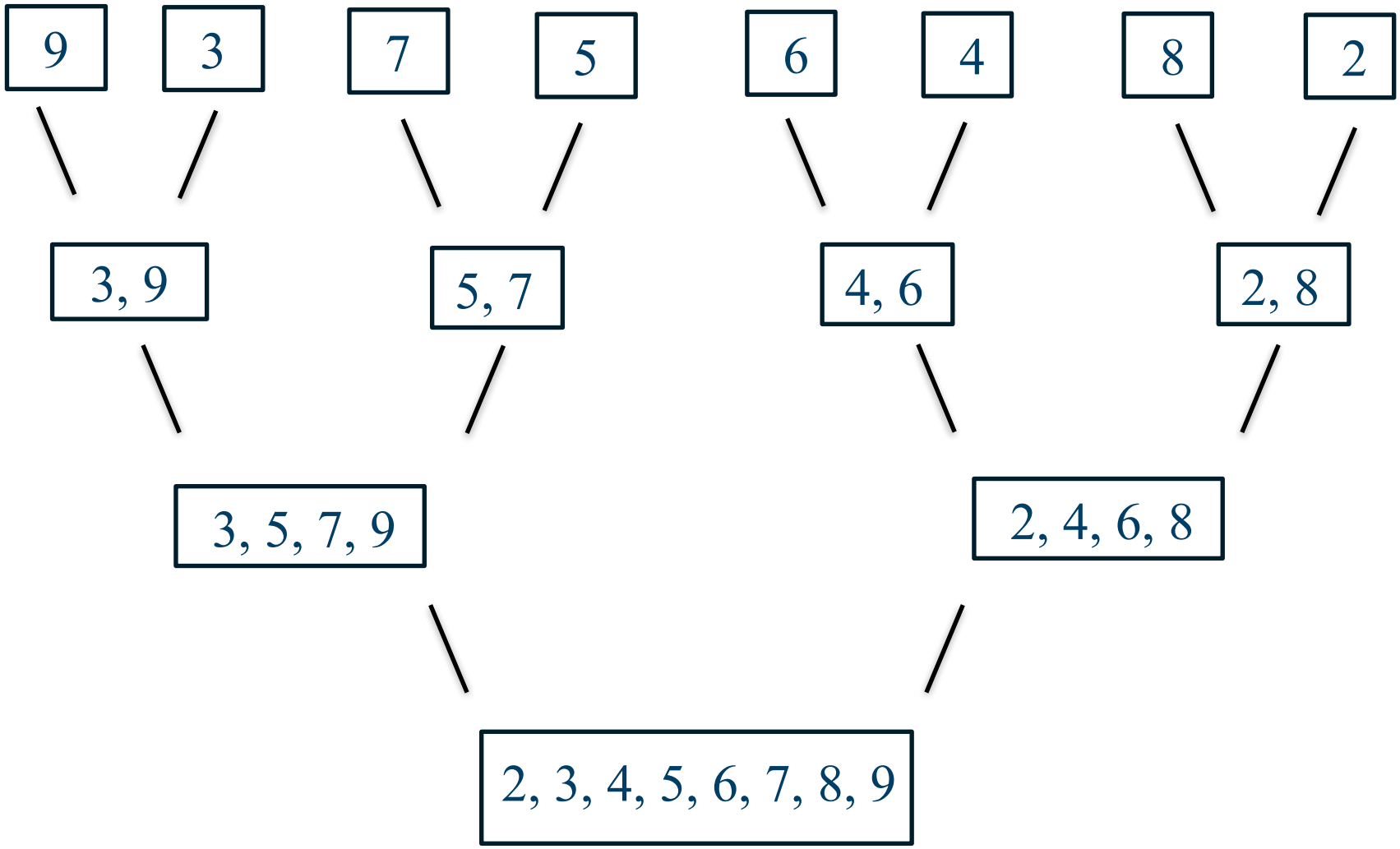
Simulation of Merge Sort



Compare and Merge

Simulation of Merge Sort

Compare and Merge



Merging Procedure for arrays

A: 2, 8, 15, 18

i

B: 5, 9, 12, 17

j

C: 2

k

Compare A[i] with B[j]
Copy smaller value to C[k]

Merging Procedure for arrays

A: 2, 8, 15, 18

i

B: 5, 9, 12, 17

j

C: 2, 5,

k

Compare A[i] with B[j]
Copy smaller value to C[k]

Merging Procedure for arrays

A: 2, 8, 15, 18

i

B: 5, 9, 12, 17

j

C: 2, 5, 8,

k

Compare A[i] with B[j]
Copy smaller value to C[k]

Merging Procedure for arrays

A: 2, 8, 15, 18

i

B: 5, 9, 12, 17

j

C: 2, 5, 8, 9,

k

Compare A[i] with B[j]
Copy smaller value to C[k]

Merging Procedure for arrays

A: 2, 8, 15, 18

i

B: 5, 9, 12, 17

j

C: 2, 5, 8, 9, 12

k

Compare A[i] with B[j]
Copy smaller value to C[k]

Merging Procedure for arrays

A: 2, 8, 15, 18

i

B: 5, 9, 12, 17

j

C: 2, 5, 8, 9, 12, 15

k

Compare A[i] with B[j]
Copy smaller value to C[k]

Merging Procedure for arrays

A: 2, 8, 15, 18

i

B: 5, 9, 12, 17

j

C: 2, 5, 8, 9, 12, 15, 17

k

Compare A[i] with B[j]
Copy smaller value to C[k]

Merging Procedure for arrays

A: 2, 8, 15, 18

i

B: 5, 9, 12, 17

j

C: 2, 5, 8, 9, 12, 15, 17, 18

k

Compare A[i] with B[j]
Copy smaller value to C[k]

Time Complexity of MergeSort

Recursion takes O(log(n))

Merging works in O(n)

The worst-case complexity if O(nlog(n))

What is the space complexity?

Summary of Sorting Algorithms

Algorithm	Time	Notes
Selection sort	$O(n^2)$	• In-place • Slow (good for small inputs)
Insertion sort	$O(n^2)$	• In-place • Slow (good for small inputs)
Quick sort	$O(n^2)$ expected	• In-place, randomized • Fastest (good for large inputs)
Merge sort	$O(n \log n)$	• Sequential data access • Fast (good for huge inputs)