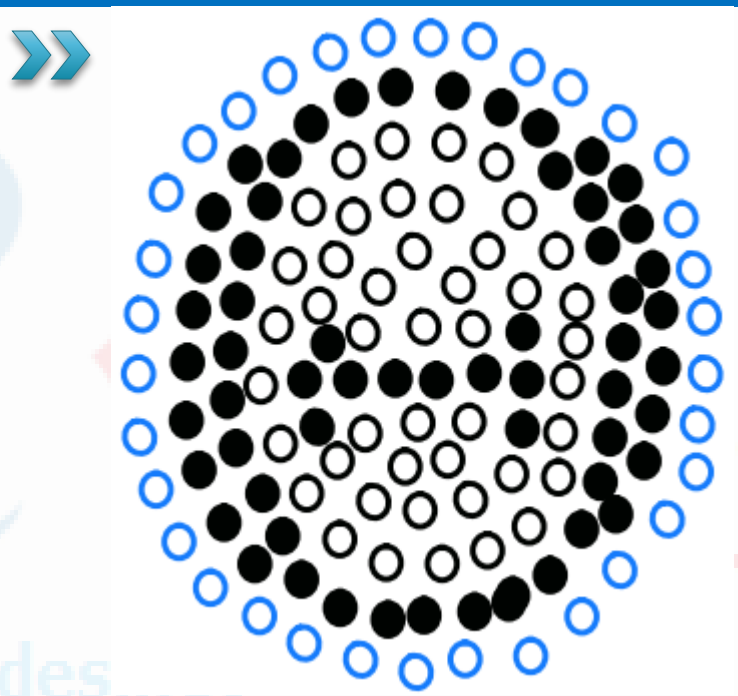
FACULTY OF INFORMATION TECHNOLOGY

DATA STRUCTURES (CTDL)

Semester 1, 2021/2022

Algorithm Analysis



Experimental Studies

- How to study the efficiency of an algorithm?
  - implement the algorithm
  - run the program on various test inputs
  - records the time spent during each execution

```
1  long startTime = System.currentTimeMillis( );        // record the starting time
2  /* (run the algorithm) */
3  long endTime = System.currentTimeMillis( );          // record the ending time
4  long elapsed = endTime − startTime;                  // compute the elapsed time
```

Experimental Studies (cont.)

- Java provides:
  - System.currentTimeMillis(): current time in milliseconds
  - System.nanoTime(): current time in nanoseconds

- Limitations?
  - The measured times reported by both methods currentTimeMillis and nanoTime will vary greatly
    - from machine to machine, and may likely vary
    - from trial to trial, even on the same machine.

Example

- Consider two algorithms for constructing long strings in Java

```
1  /** Uses repeated concatenation to compose a String with n copies of character c. */
2  public static String repeat1(char c, int n) {
3    String answer = "";
4    for (int j=0; j < n; j++)
5      answer += c;
6    return answer;
7  }
8
9  /** Uses StringBuilder to compose a String with n copies of character c. */
10 public static String repeat2(char c, int n) {
11   StringBuilder sb = new StringBuilder( );
12   for (int j=0; j < n; j++)
13     sb.append(c);
14   return sb.toString( );
15 }
```

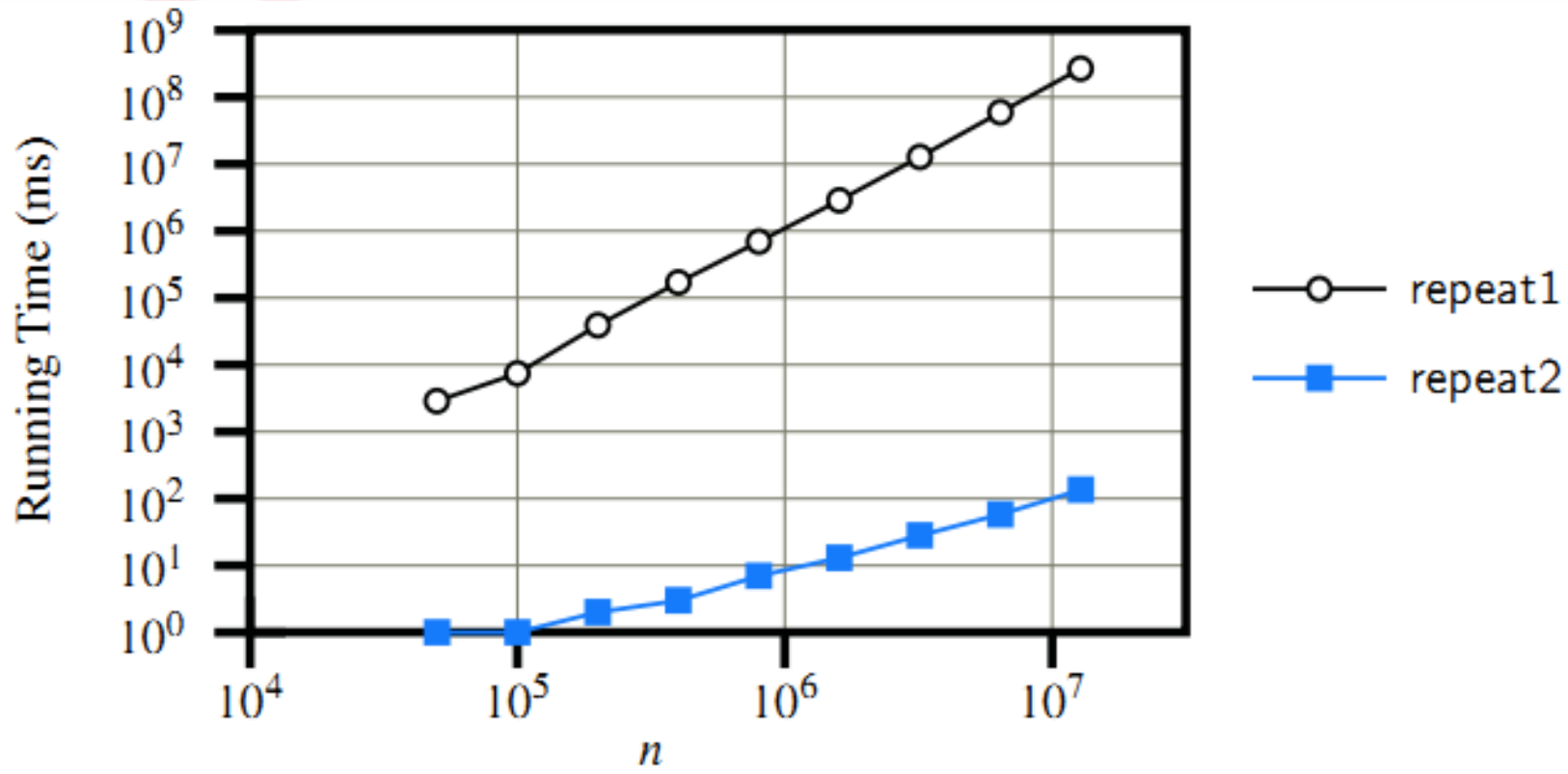
Example (cont.)

- Results of timing experiment on the methods using System.currentTimeMillis( )

n	repeat1 (in ms)	repeat2 (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,874,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421	135

Example (cont.)

- Chart of the results of the timing experiment of both methods



Challenges of Experimental Analysis

- Three challenges of experimental analysis:
  - The experiments are performed in the same hardware and software environments.
  - Experiments can be done only on a limited set of test inputs ➔ they leave out the running times of inputs not included in the experiment (and these inputs may be important).
  - An algorithm must be fully implemented in order to execute it to study its running time experimentally.

Moving Beyond Experimental Analysis

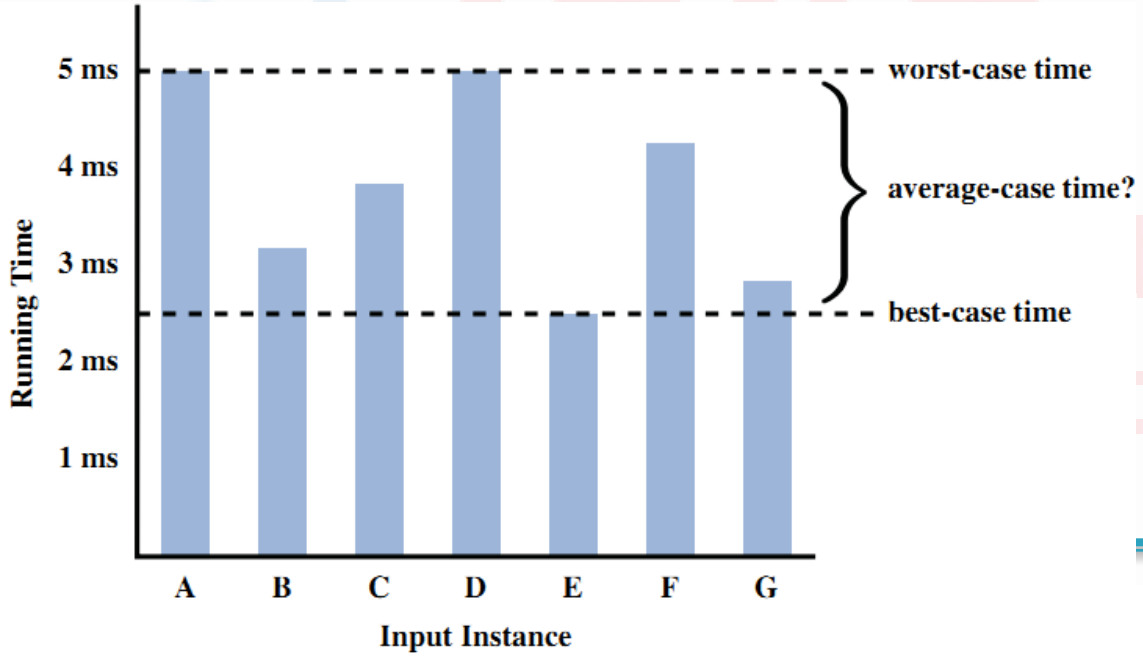
- An approach to analyzing the efficiency of algorithms should:
  - Evaluate the relative efficiency of any two algorithms in a way that is independent of the hardware and software environment.
  - Study a high-level description of the algorithm without need for implementation.
  - Take into account all possible inputs.

Moving Beyond Experimental Analysis (cont.)

- Counting Primitive Operations:
  - Assigning a value to a variable
  - Following an object reference
  - Performing an arithmetic operation (i.e., adding two numbers)
  - Comparing two numbers
  - Accessing a single element of an array by index
  - Calling a method
  - Returning from a method

Moving Beyond Experimental Analysis (cont.)

- Measuring Operations as a Function of Input Size
  - a function f (n) that characterizes the number of primitive operations (performed as a function of the input size n)
- Focusing on the Worst-Case Input



Big-Oh Notation

- For any monotonic functions f(n) and g(n) from the positive integers to the positive integers.

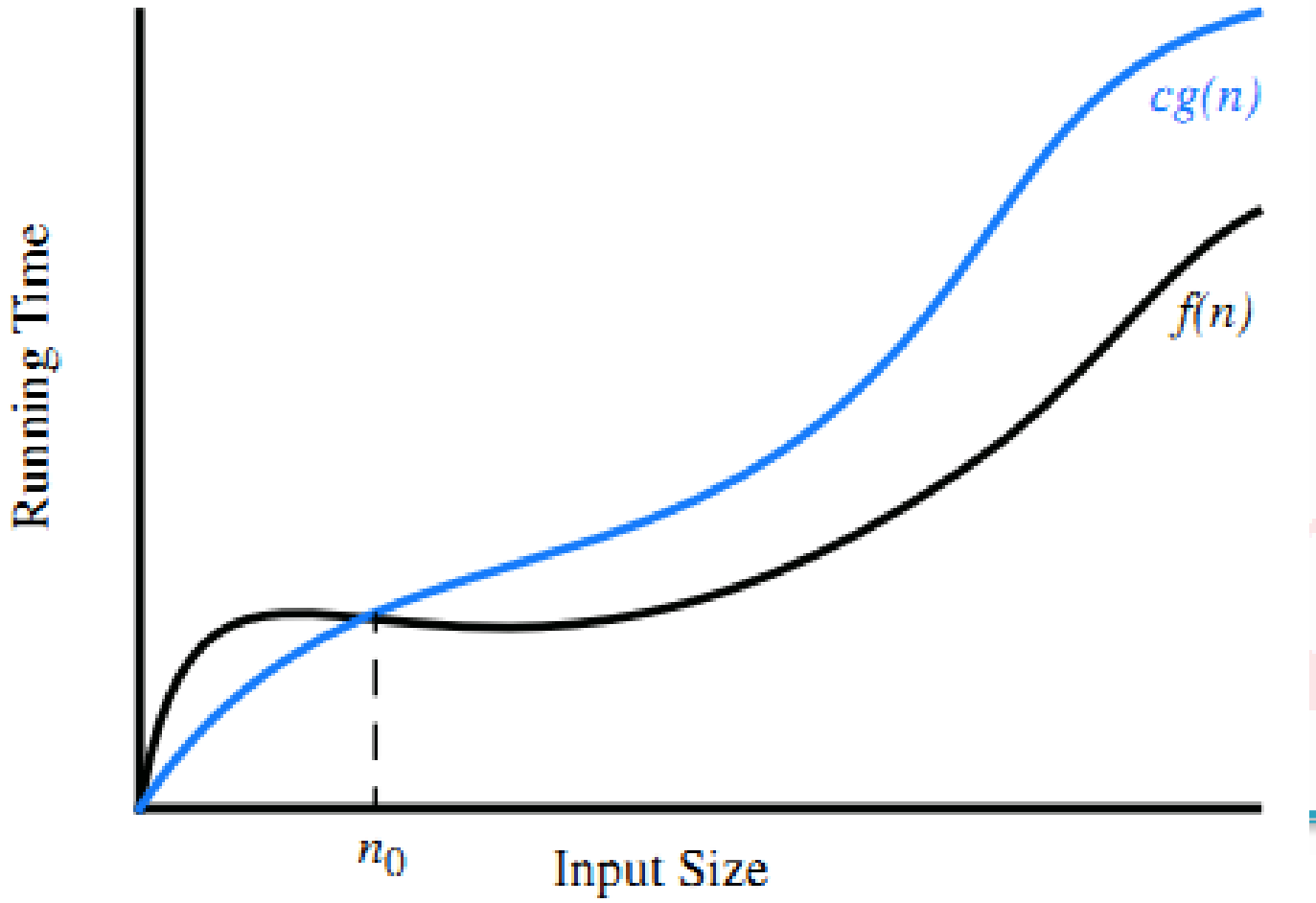
- We say that **f(n) = O(g(n))** when there exist constants **c > 0** and **$n_0 > 0$** such that

$$f(n) \leq c * g(n), \text{ for all } n \geq n_0$$

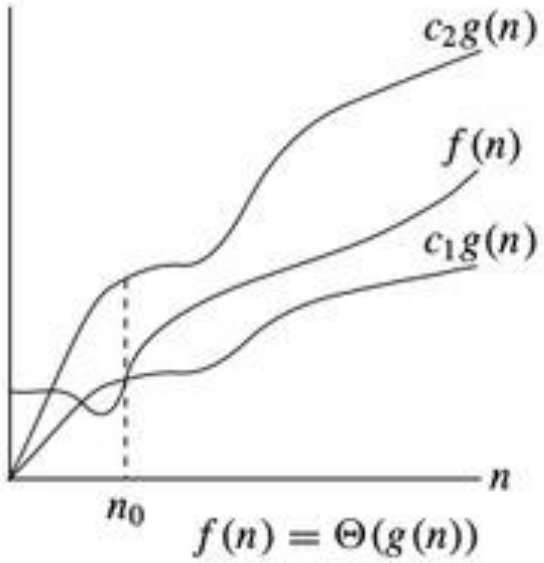
- Function **g(n)** is called an **upper bound** for **f(n)**, for all sufficiently large **$n \to \infty$**

Big-Oh Notation

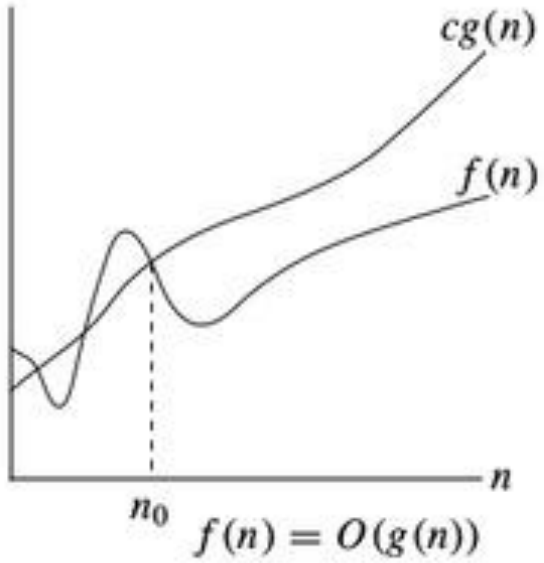
- The function f (n) is O(g(n)), since f (n) ≤ c * g(n) when n ≥ n_0



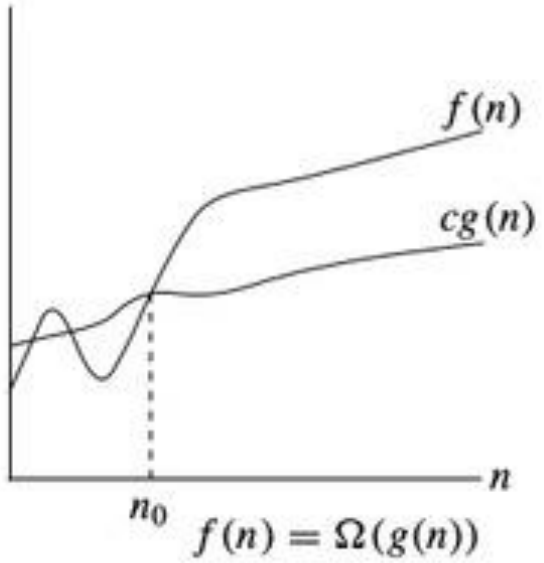
Big-omega, Big-theta



(a)



(b)



(c)

Rules for Big-Oh

- If $T = O(c*f(n))$ for a constant c, then

  $T = O(f(n))$
- If $T_1 = O(f(n))$ and $T_2 = O(g(n))$ then

  $T_1 + T_2 = O(max(f(n), g(n)))$ ➔ the sum rule
- If $T_1 = O(f(n))$ and $T_2 = O(g(n))$ then

  $T_1 * T_2 = O(f(n) * g(n))$ ➔ the product rule
- If f is a polynomial of degree k then

  $f(n) = O(n^k)$

Properties of the Big-Oh Notation

- Ignore constant factors and lower-order terms
  - $5n^4 + 3n^3 + 2n^2 + 4n + 1$ ➔ $O(n^4)$
  - $5n^2 + 3n\log n + 2n + 5$ ➔ $O(n^2)$
  - 2^{n+2} ➔ $O(2^n)$
- Characterizing Functions in Simplest Terms
  - $f(n) = 4n^3 + 3n^2$
    - $O(n^5)$
    - $O(n^4)$
    - **$O(n^3)$**

Example

- Let T (n): the number of units of time taken by a program or an algorithm on any input of size n, $T(n)=f(n)=O(g(n))$.

- Suppose we have a program whose running time is $T(0) = 1$, $T(1) = 4$, $T(2) = 9$, and in general $T(n) = (n+1)^2$

➔$T(n) = O(n^2)$

Since $n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2 = 4n^2$

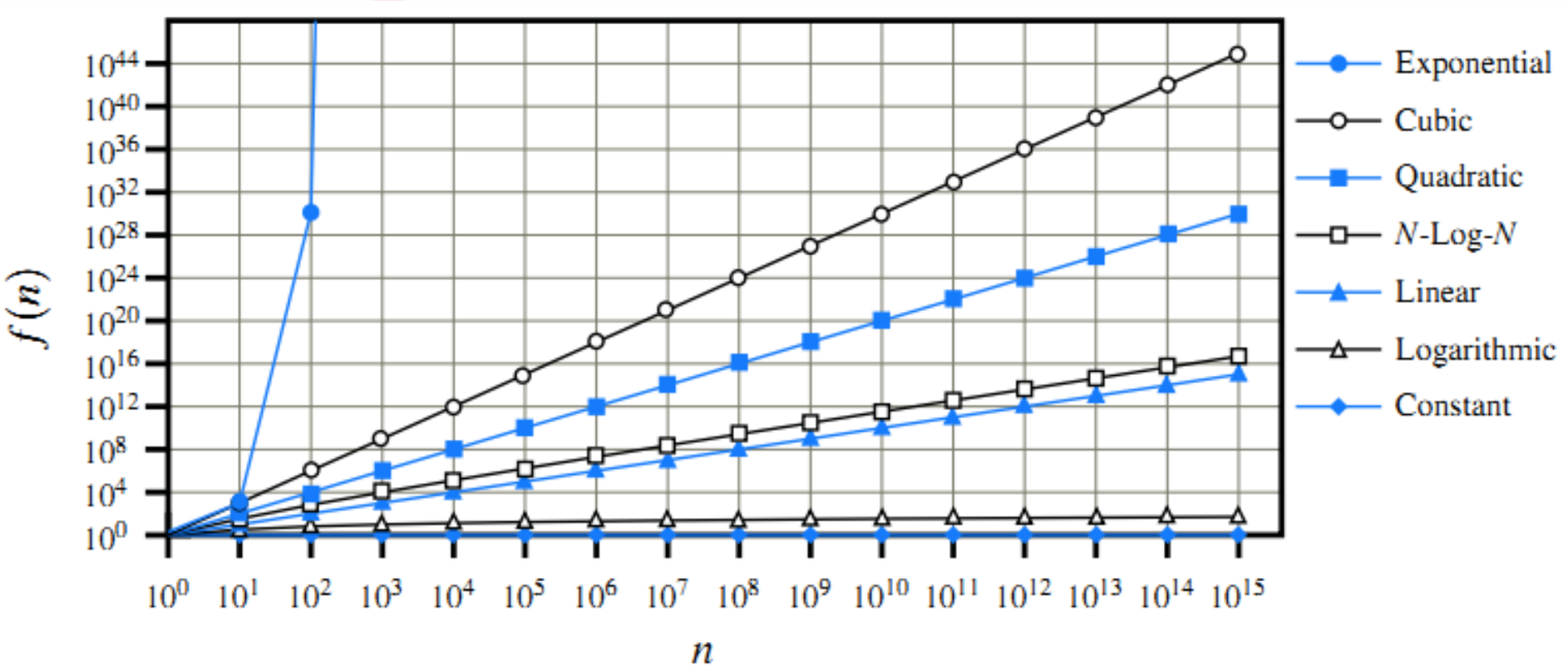
Selected functions

- Constant function
  - $f(n) = c$, $c=10$, $c=27$, $c=2^{10}$
- Logarithm Function
  - $f(n) = \log_b n$, (the most common base for the logarithm function in computer science is 2)
- N-Log-N Function
  - $f(n) = n\log_2 n$
- Quadratic Function
  - $f(n) = n^2$

Selected functions (cont.)

- Cubic Function and Other Polynomials
  - $f(n) = n^3$
  - $f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \ldots + a_d n^d$
- Exponential Function
  - $f(n) = b^n$
- Geometric Sums
  - $f(n) = c$, $c=10$, $c=27$, $c=2^{10}$

Order of Big-Oh



How to Compute Complexity?

How to determine complexity?

- Remove constant:

$$T(n) = O(c.f(n)) = O(f(n))$$

$n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2 = 4n^2$

➔ $T(n)=O(n^2)$

How to determine complexity?

- Sequence of statements

```
statement 1;
statement 2;
...
statement k;
```

T(n) = time(statement 1) + time(statement 2) + ... + time(statement k)

Example

```
1 for ( int i=0; i<n; i++)
2     X[i] = 0;
3 for ( int i=0; i<n; i++)
4       for ( int j=0; j<n; j++)
5           X[i] += i+j;
```

O(n)

$O(n^2)$

$T(n) = \max(O(n), O(n^2)) = O(n^2)$

How to determine complexity?

- If-Then-Else

```
if (condition) then
        block 1 (sequence of statements)
else
        block 2 (sequence of statements)
end if;
```

T(n) = max(time(block 1), time(block 2))

Example

If-Then-Else

```
(1)         if (A[1][1] == 0)
(2)             for (i = 0; i < n; i++)
(3)                 for (j = 0; j < n; j++)
(4)                     A[i][j] = 0;
            else
(5)             for (i = 0; i < n; i++)
(6)                 A[i][i] = 1;
```

$O(n^2)$

$O(n)$

$T(n) = O(n^2)$

How to determine complexity?

- Loops

```
for i in 1 .. n
        loop sequence of statements
end loop;
```

- If we assume the statements are O(1)

T(n)= n* O(1) = O(n)

Example

```
sum = 0;
for( i = 0;  i < n;  i++ )
    sum = sum + i;
```

T(n) = **O(n)**

How to determine complexity?

- Nested loops

```
for i in 1 .. n
        loop for j in 1 .. m
                loop sequence of statements
        end loop;
 end loop;
```

T(n)= O(n * m)

Example

```
sum = 0;
for( i = 0; i < n; i++)
    for( j = 0; j < n; j++)
        sum++;
```

$T(n) = O(n^2)$

Example

```
sum = 0;
for( i = 0 ; i  <  n;  i++)
      for( j = 0 ; j  <  n*n ; j++ )
            sum++;
```

$T(n) = O(n^3)$

Example

- Nested loop:

```
1 int k=0;
2 for ( int i=0; i<n; i++)
3       for ( int j=i; j<n; j++)
4             k++;
```

$T(n) = O(n^2)$

How to determine complexity?

- Statements with function/ procedure call

```
for j in 1 .. n
        loop g(j);
end loop
```

- Suppose: complexity of g(j) = O(n)

$T(n)= O(n^2)$

Example

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

Example

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
        if (a[ i ][ j ] == x) return 1 ;
return -1;
```

Runtime complexity types

- ***Worst-case**:* the maximum number of steps taken on any instance of size a.
- ***Best-case**:* the minimum number of steps taken on any instance of size a.
- ***Average case**:* an average number of steps taken on any instance of size a.

Example

- Let us consider an algorithm of sequential searching in an array of size n.
- *Worst-case: ?*
- *Best-case: ?*
- *Average case: ?*

Example

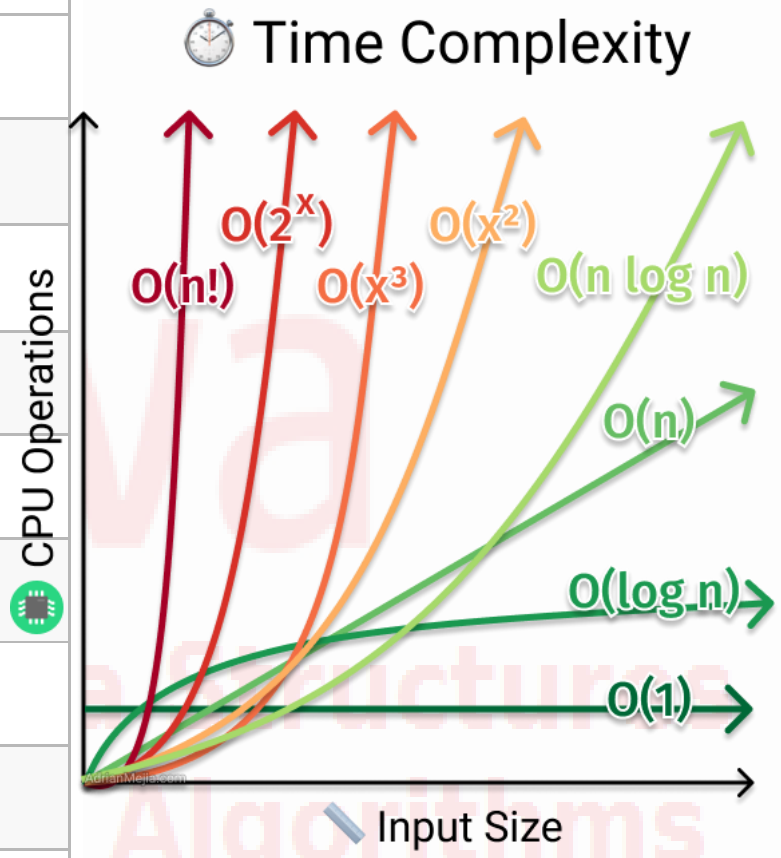
- Let us consider an algorithm of sequential searching in an array of size n.
- *Worst-case:* O(n)
- *Best-case:* O(1)
- *Average case:* O(n/2)=O(n)

Summary

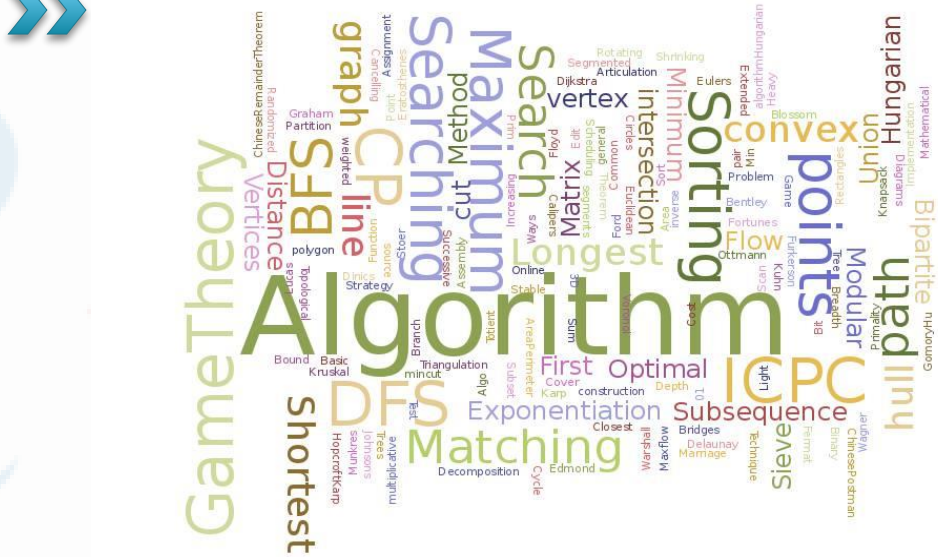
Class	Name	Comments
1	Constant	Algorithm ignores input
log n	Logarithmic	Cuts problem size by constant fraction on each iteration
n	Linear	Algorithm scans its input (at least)
n log n	n-log-n	Some divide and conquer
n^2	Quadratic	Loop inside loop = "nested loop"
n^3	Cubic	Loop inside nested loop
2^n	Exponential	Algorithms generates all subsets of n-element set
n!	Factorial	Algorithms generates all permutations of n-element set

Accepted complexity

Length of Input (N)	Worst Accepted Algorithm
$\leq [10..11]$	$O(N!), O(N^6)$
$\leq [15..18]$	$O(2^N * N^2)$
$\leq [18..22]$	$O(2^N * N)$
≤ 100	$O(N^4)$
≤ 400	$O(N^3)$
$\leq 2K$	$O(N^2 * logN)$
$\leq 10K$	$O(N^2)$
$\leq 1M$	$O(N * logN)$
$\leq 100M$	$O(N), O(logN), O(1)$



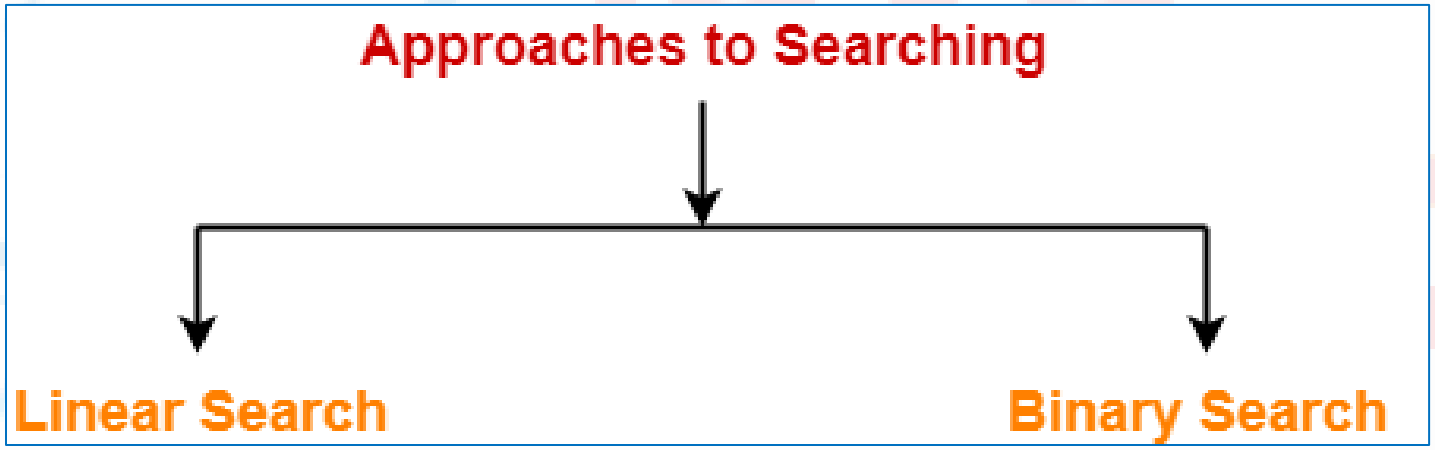
Algorithms



Searching algorithms

- **Linear search**: the list or array is traversed sequentially and every element is checked (**Sequential Search**)

- **Binary search**: (searching in sorted data-structures) The algorithm repeatedly targets the center of the search structure and divide the search space in half (**Interval Search**)

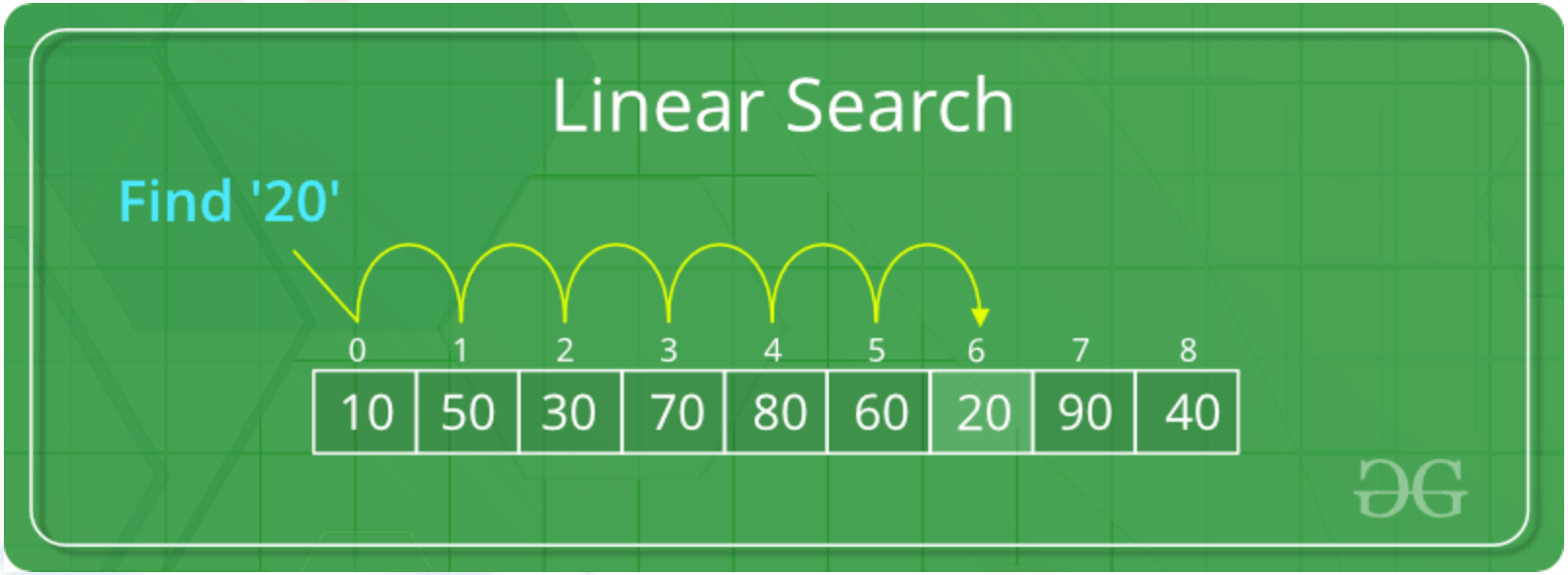


Linear search

```
public int linearSearch(int[] arr, int size, int target)  {
for (int i=0; i<size-1; i++) {
    if(arr[i] == target)
        return i;
    }
    return -1;
}
```

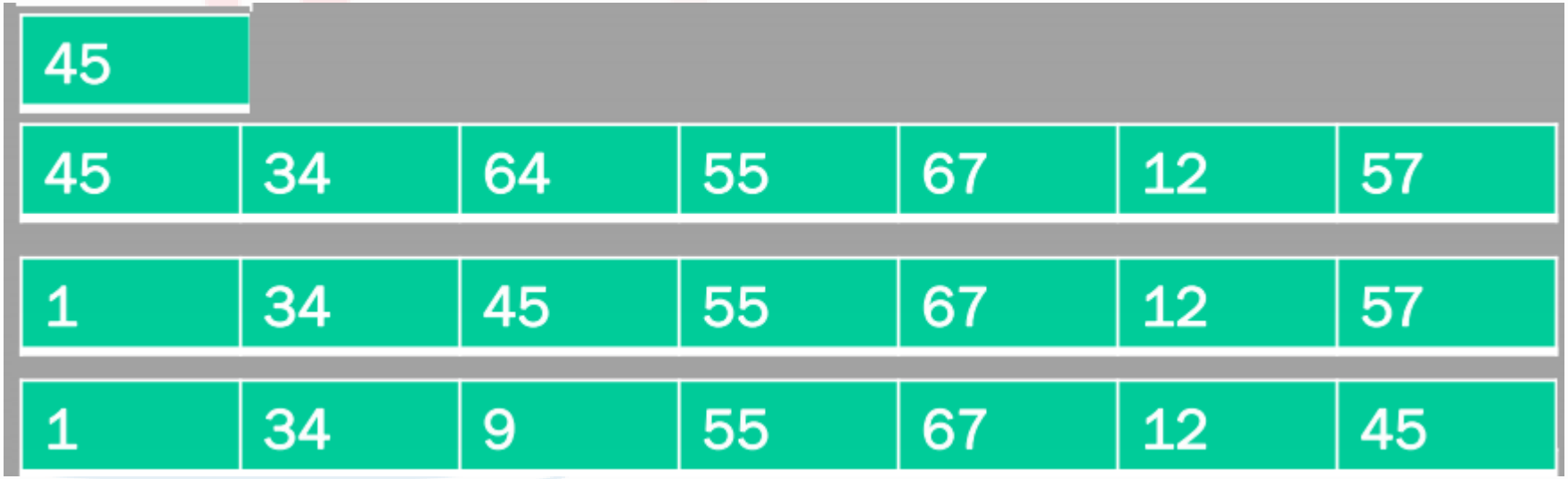
Linear search

- Linear Search to find the element "20" in a given list of numbers



Running time

- How about the running time for searching 45 in these arrays?



Running time

- Best case: 1 comparable time
- Worst case: n comparable times
- AVG case: (n+1) /2 comparable times

Recursive approach

Recursive Array Search

- Searching an array can be accomplished using recursion
- Simplest way to search is a linear search
  - Examine one element at a time starting with the first element and ending with the last
  - On average, (n + 1)/2 elements are examined to find the target in a linear search
  - If the target is not in the list, nelements are examined
- A linear search is O(n)

Recursive Array Search (cont.)

- Base cases for recursive search:
  - Empty array, target can not be found; result is -1
  - First element of the array being searched = target; result is the subscript of first element

- The recursive step searches the rest of the array, excluding the first element

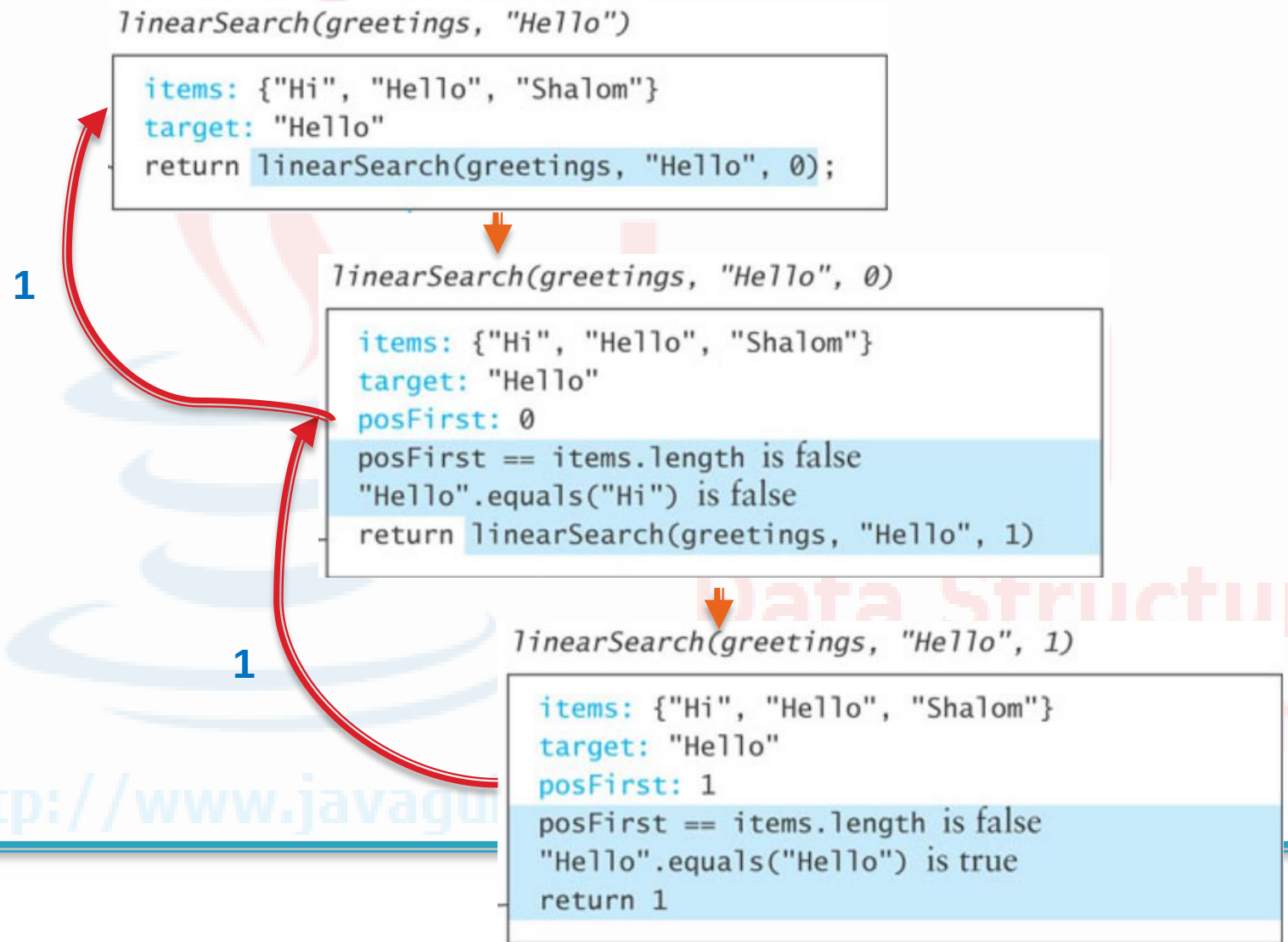
Recursive Array Search (cont.)

- Algorithm for recursive linear array search

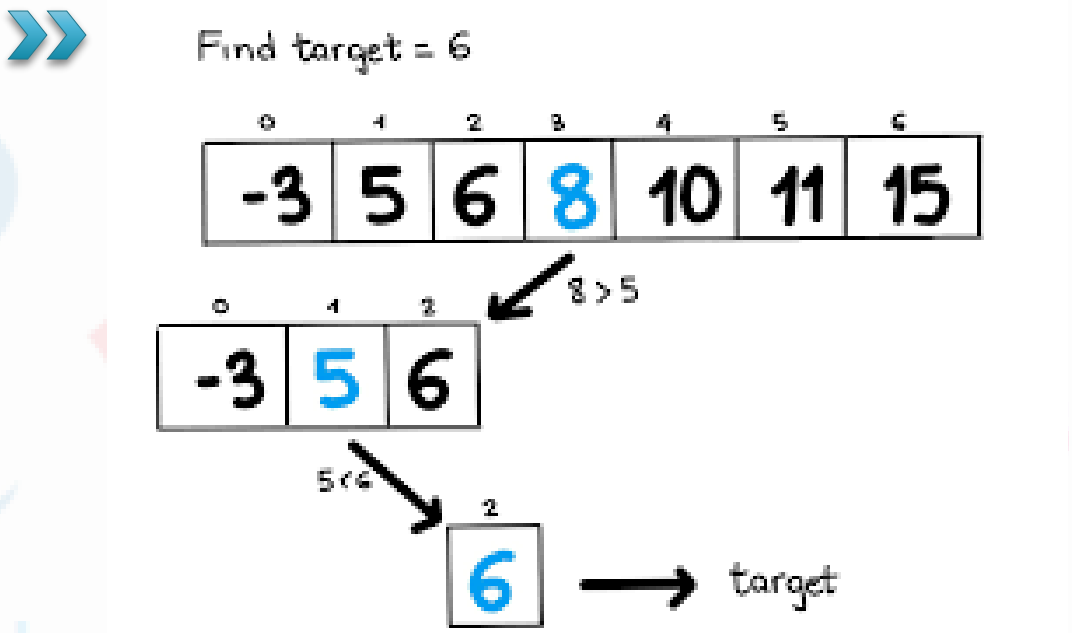
```
if the array is empty
     the result is –1
else if  the first element matches the target
     the result is the position of the first element
else
     search the array excluding the first element and return the result
```

Recursive Array Search (cont.)

- Implementation:



Binary Search

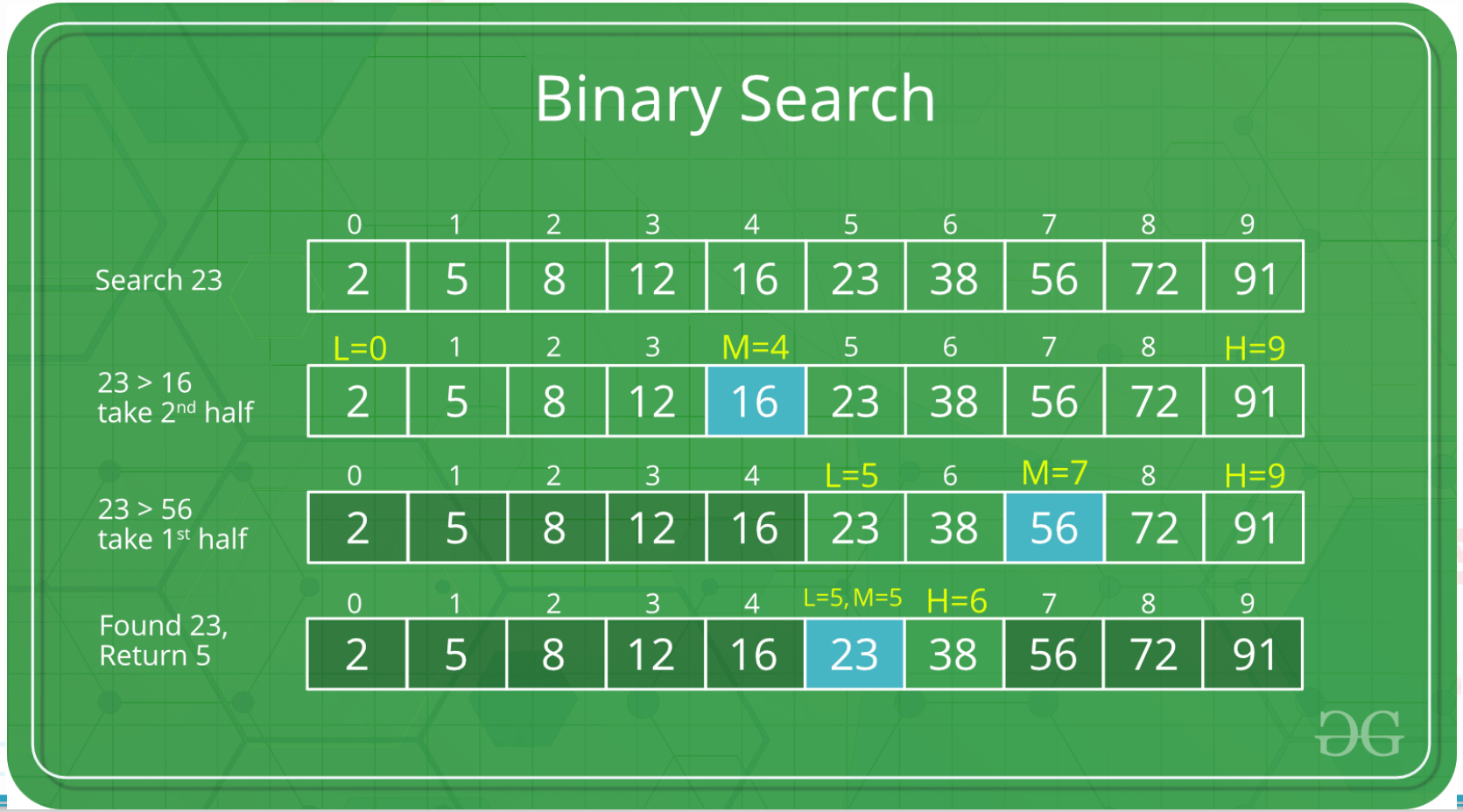


Binary search

- Binary search: searching in sorted data-structures
  - Compare x with the middle element.
  - If x matches with middle element, we return the mid index.
  - Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
  - Else (x is smaller) recur for the left half.

Binary search: Example 1

- Binary Search to find the element "23" in a given list of numbers



Non-recursion for Binary search

```
1  /** Returns true if the target value is found in the data array. */
2  public static boolean binarySearchIterative(int[ ] data, int target) {
3    int low = 0;
4    int high = data.length − 1;
5    while (low <= high) {
6      int mid = (low + high) / 2;
7      if (target == data[mid])                    // found a match
8        return true;
9      else if (target < data[mid])
10       high = mid − 1;                           // only consider values left of mid
11     else
12       low = mid + 1;                            // only consider values right of mid
13   }
14   return false;                                 // loop ended without success
15 }
```

Running time

- Best case: O(1)
- Worst case: $O(\log_2(n))$
- AVG case: $O(\log_2(n))$

Calculating Time complexity

- Suppose that Binary Search terminates after **k** iterations
  - At each iteration, the array is divided by half. So let's say the length of array at any iteration is **n**
  - **Iteration 1**, the length of array = n
  - **Iteration 2**, the length of array = $n/2$
  - **Iteration 3**, the length of array = $(n/2)/2 = n/2^2$
  - ...
  - **Iteration k**, the length of array = $n/2^k$

Calculating Time complexity

- Also, we know that after k divisions, the length of array becomes 1
- Therefore, the length of array = $n/2^k = 1 \Rightarrow n = 2^k$
- Applying log function on both sides:

$\Rightarrow \log_2(n) = \log_2(2^k) \Rightarrow \log_2(n) = k \log_2(2)$

- As ($\log_a(a) = 1$),

➔ $k = \log_2(n)$

- Hence, the time complexity of Binary Search is $\log_2(n)$

Recursive approach

Recursion for Binary Search

- Low = begin of array, high = end of array, mid = (high + low)/2
- STOP CONDITION/BASE CASE
  - Target equals with value of element at middle
  - Element is not found?
- CONTINIOUS CONDITION/RECURSIVE CASE
  - Target larger than mid ➔ Recursive with low= mid+1, high not change
  - Target smaller than mid ➔ Recursive with low not change, high = mid -1

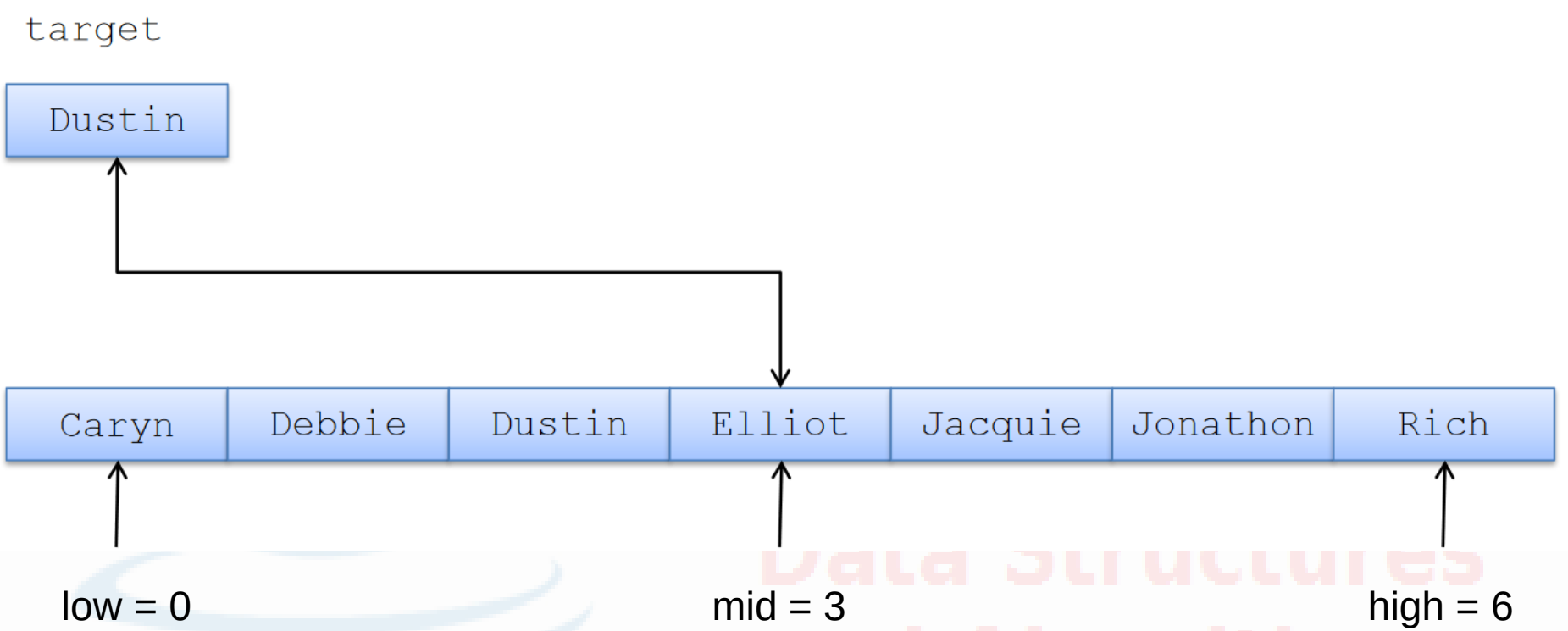
ONLY FOR SORTED ARRAY!!!

Recursion for Binary Search

```
/**
 * Returns true if the target value is found in the indicated portion of the data array.
 * This search only considers the array portion from data[low] to data[high] inclusive.
 */
public static boolean binarySearch(int[ ] data, int target, int low, int high) {
  if (low > high)
    return false;                                          // interval empty; no match
  else {
    int mid = (low + high) / 2;
    if (target == data[mid])
      return true;                                         // found a match
    else if (target < data[mid])
      return binarySearch(data, target, low, mid - 1);     // recur left of the middle
    else
      return binarySearch(data, target, mid + 1, high);    // recur right of the middle
  }
}
```

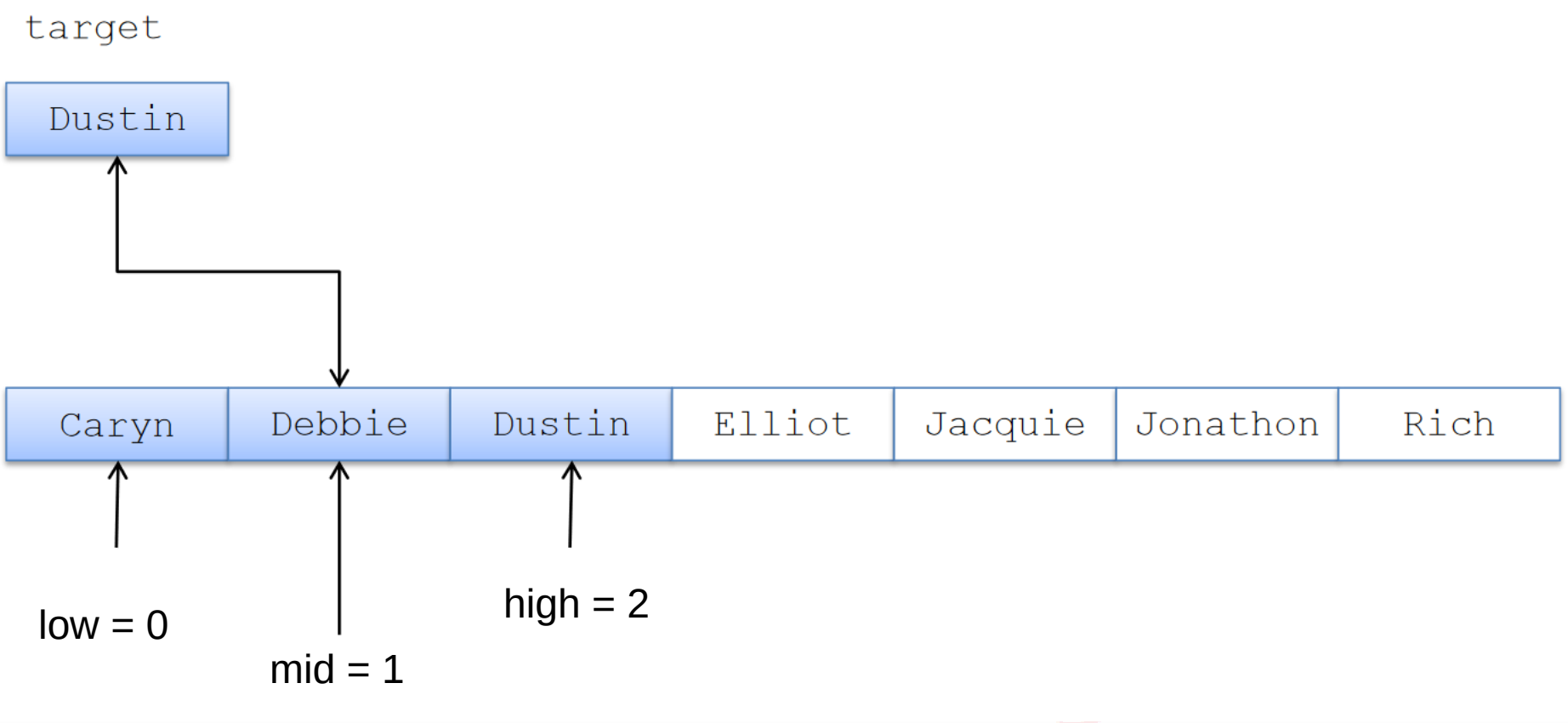
Binary search: Example 2

First call



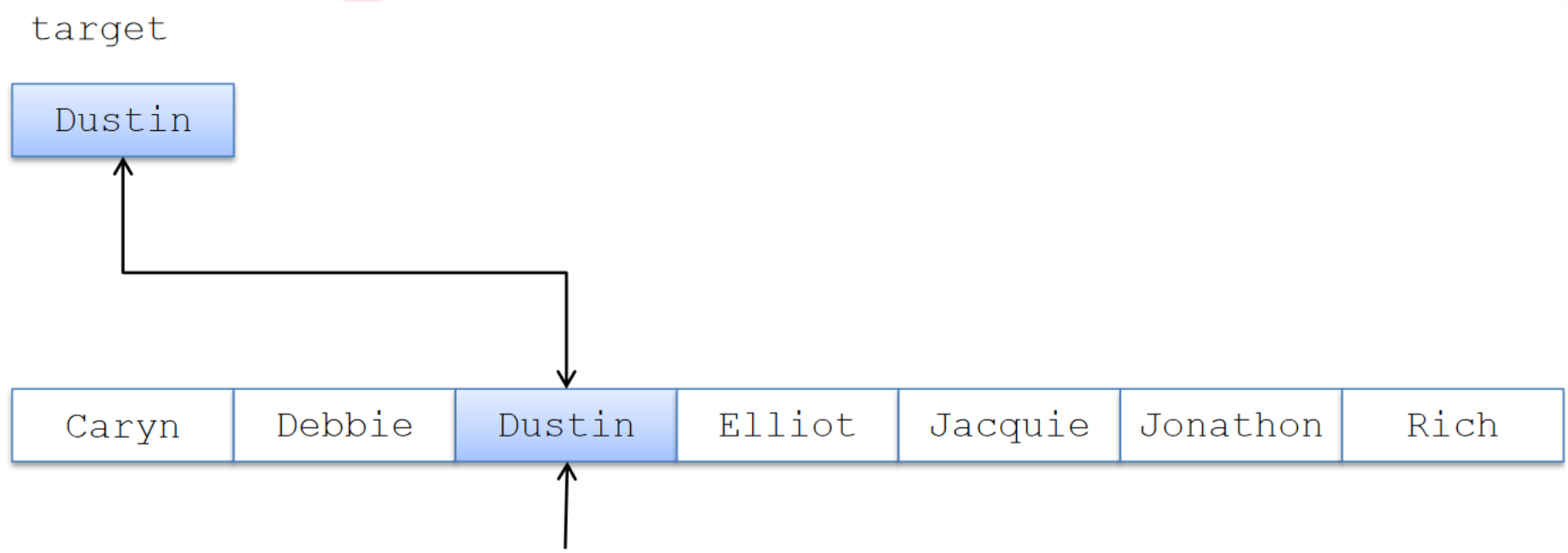
Binary search: Example 2

Second call



Binary search: Example 2

Third call



Sorting algorithms

- Bringing Order to the World

Why Study Sorting?

- When an input is sorted, many problems become easy (e.g. searching, min, max, k-th smallest)
- Sorting has a variety of interesting algorithmic solutions that embody many ideas
  - Iterative
  - Recursive
  - Divide-and-conquer

How to sort?

- For a given array: int[] arr = {5, 3, 9, 10, 4};
- How to sort arr (ascending order)?

Selection Sort

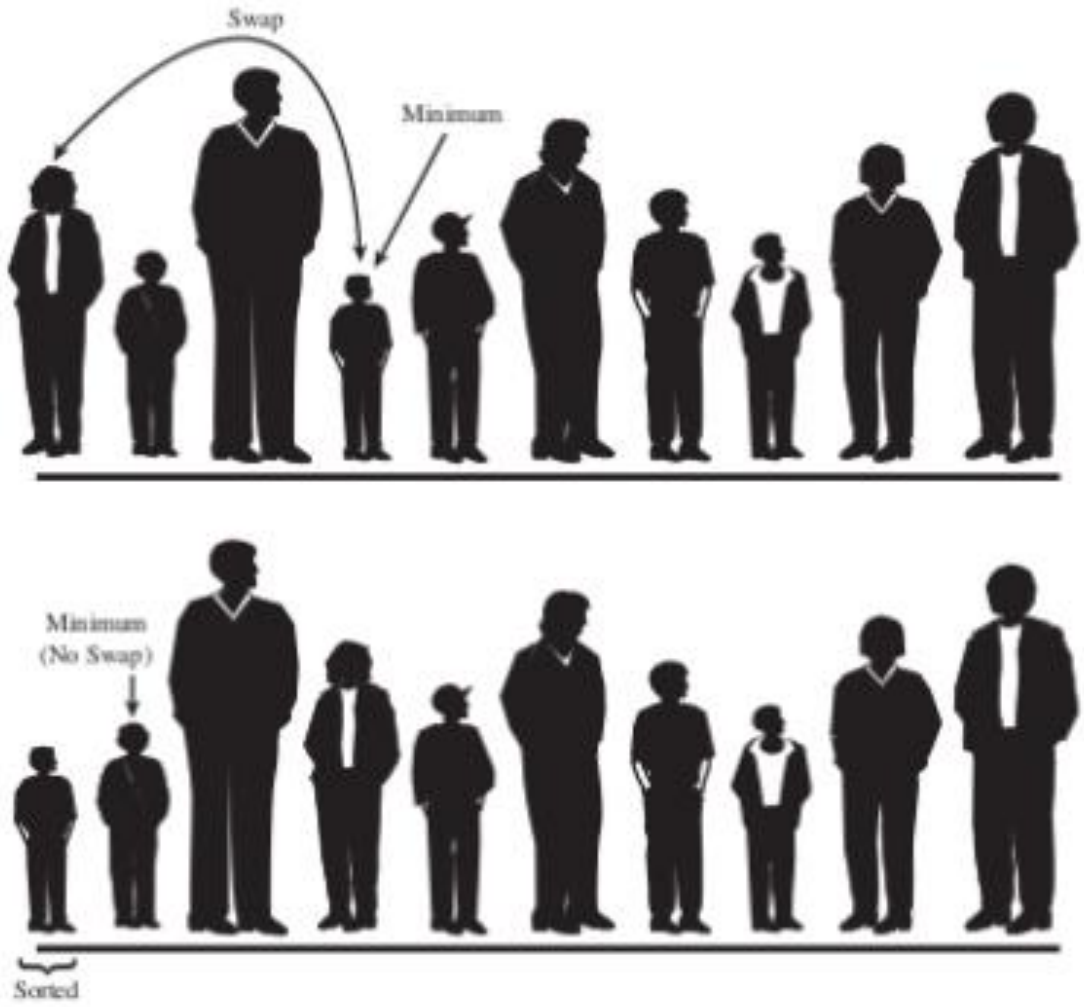
Selection sort: idea

- Finding the smallest (or largest depending on the sorting order) element in the unsorted sublist
- Exchanging it with the leftmost unsorted element (putting in sorted order)
- Moving the sublist boundaries one element to the right.

Selection sort: step by step

- Given an array of n items
  - **Step 1.** Find the largest/smallest item x, in the range of [0…n−1]
  - **Step 2.** Swap x with the $(n-1)^{th}$ item
  - **Step 3.** Reduce **n** by 1 and go to Step 1

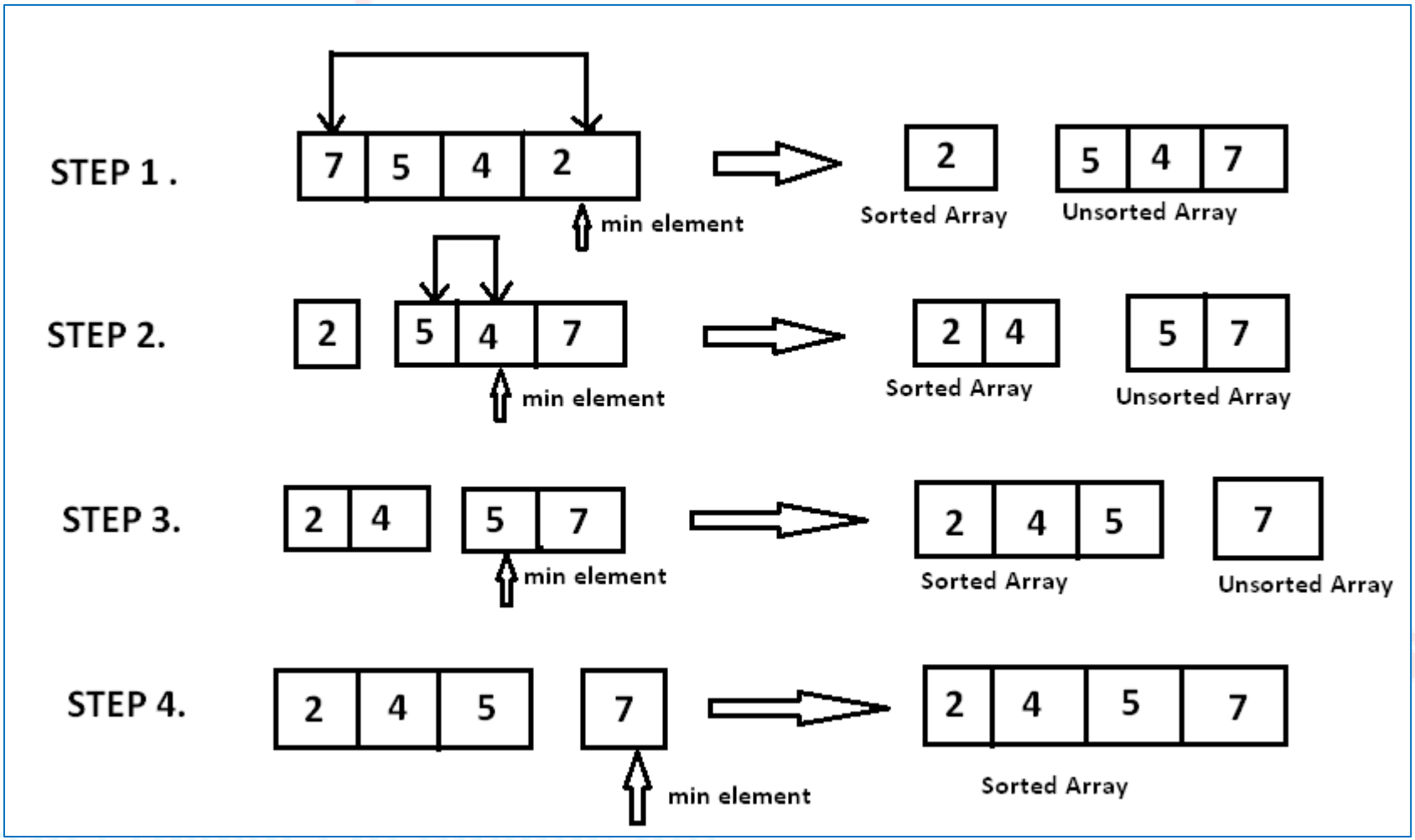
Selection sort algorithm



Running time

- Best case: $O(n^2)$
- Worst case: $O(n^2)$
- AVG case: $O(n^2)$

Example 1

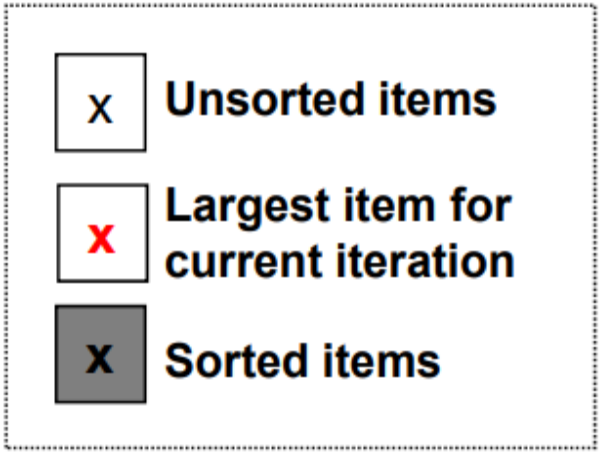


Example 2

29	10	14	37	13
29	10	14	13	37
13	10	14	29	37
13	10	14	29	37
10	13	14	29	37

37 is the largest, swap it with the last element, i.e. 13.

Q: How to find the largest?



Sorted!

How to implement Selection sort?

Given an array of n items (X).

- **Step 1**: i = 0// first index
- **Step 2**: Finding X[min] or X[max] in X[i] ... X[n-1]
- **Step 3**: Swap X[i] to X[min], if min or max equal i, quit this step.
- **Step 4**:

If i <= n-1, then i = i +1, go to Step 2.

Else, stop - finish sorting array.

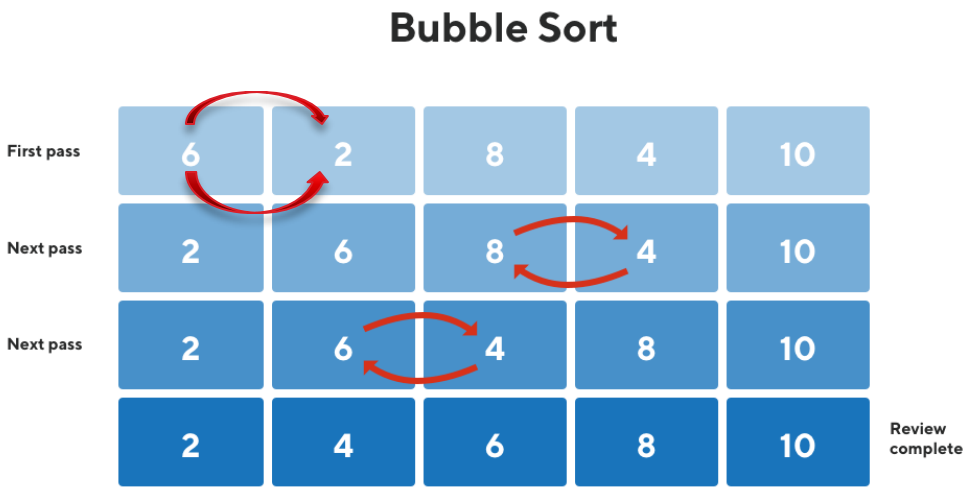
Selection sort: recursive approach

```
public class SelectionSort {
    private static void swap(int[] a, int i, int j) {
        // switch value at index i to value at index j
    }

    public static int[] selectionSort_Min(int[] array, int stepNum) {
        if (stepNum > array.length - 1) {
            return array;
        } else {
            for (int j = stepNum; j < array.length; j++) {
                // Find the index of the minimum value
                // swap
            }
        }
        return selectionSort_Min(array, stepNum + 1);
    }
}
```

Selection sort: iterative approach

```
public class SelectionSort {
    private static void swap(int[] a, int i, int j) {
        // switch value at index i to value at index j
    }

    public static int[] selectionSort_Min(int[] array) {
        for (int i = 0; i < array.length - 1; i++) {
            for (int j = i + 1; j < array.length; j++) {
                // Find the index of the minimum value
                // swap
            }
        }
        return array;
    }
}
```

Bubble sort



Bubble sort

- Comparing the adjacent pair,
- If they are in not right order, then they swapped each other position.
- When there are no elements swapped in one full iteration of element list, then it indicates that bubble sort is completed.

Bubble sort: idea

- Given an array of n items (X).
- Steps
  - take two adjacent elements at a time and compare them.
    - if X[i] > X[i+1]
      - swap
  - repeat until no more swaps are possible.

Example

- Sort the following array by ascending order:

$$X = \{5, 12, 3, 9, 16\}$$

Example

- **Iteration 1:**
- 5, 12, 3, 9, 16
  - The array stays the same because 5 is less than 12.
- 5, 12, 3, 9, 16
  - 3 and 12 are switched because 3 is less than 12 ➔ 5, 3, 12, 9, 16
- 5, 3, 12, 9, 16
  - 9 and 12 are switched since 9 is less than 12 ➔ 5, 3, 9, 12, 16
- 5, 3, 9, 12, 16
  - 12 and 16 do not switch because 12 is less than 16

Example

- **Iteration 2:**
- 5, 3, 9, 12, 16
  - 3 is less than 5, so they switch ➔ 3, 5, 9, 12, 16
- 3, 5, 9, 12, 16
  - 5 is less than 9 so they remain in the same places
- 3, 5, 9, 12, 16
  - 12 is greater than 9 so they do not switch places
- 3, 5, 9, 12, 16
  - 12 and 16 are in numerical order so they don't switch

Example

- **Iteration 3:**
- 3, 5, 9, 12, 16
  - 3 is less than 5, so they do not switch
- 3, 5, 9, 12, 16
  - 5 is less than 9 so they remain in the same places
- 3, 5, 9, 12, 16
  - 12 is greater than 9 so they do not switch places
- 3, 5, 9, 12, 16
  - 12 and 16 are in numerical order so they don't switch

Bubble sort: iterative approach

```
// sort by ascending order
public static void bubbleSort(int[] array) {
    // (array.length - 1) iterarion
    for (int k = 0; k < array.length - 1; k++) {
        // last k items are already sorted, so inner loop can
        // avoid looking at the last k items
        for (int i = 0; i < array.length - 1 - k; i++) {
            if (array[i] > array[i + 1]) {
                // SWAP
            }
        }
        // the algorithm can be stopped if the inner loop
        // didn't do any swap
    }
}
```

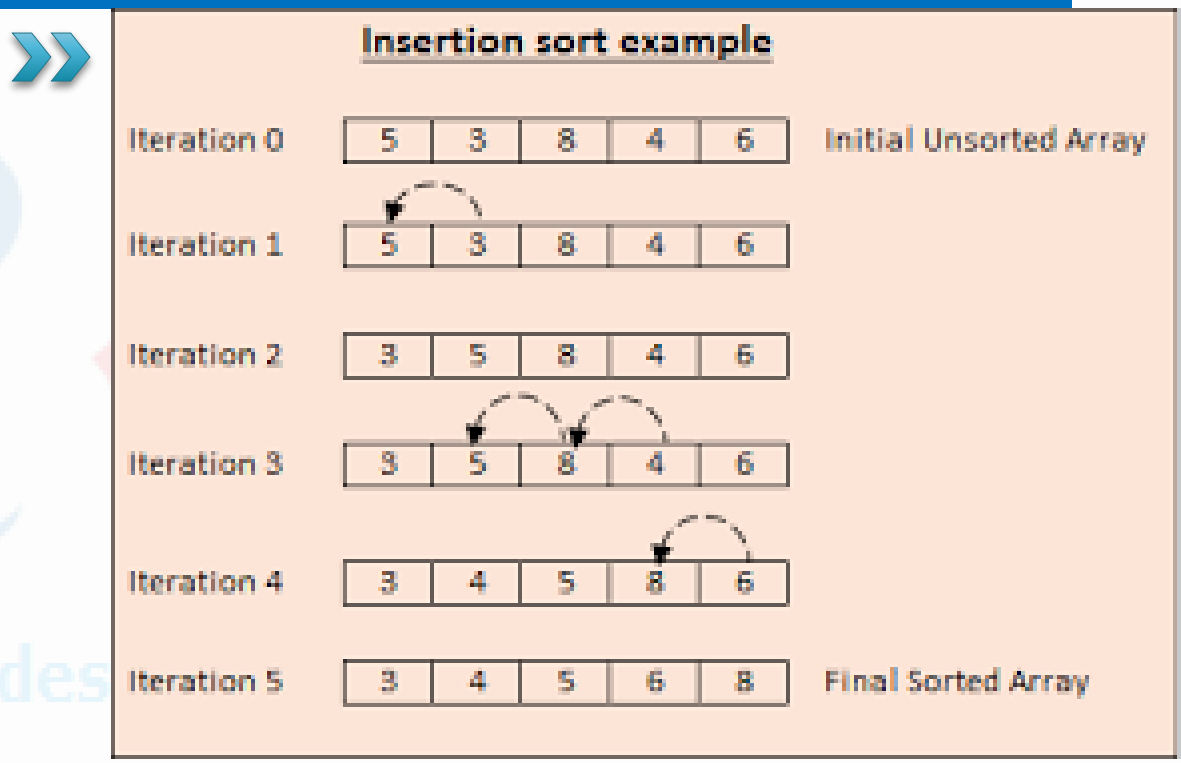
Bubble sort: recursive approach

```
public static void recursiveBubbleSort(int[] array, int n) {
    // Base case
    if (n == 1)
        return;
    // One pass of bubble sort. After
    // this pass, the largest element
    // is moved (or bubbled) to end.
    for (int i = 0; i < n - 1; i++)
        if (array[i] > array[i + 1]) {
            // swap arr[i], arr[i+1]
        }

    // Largest element is fixed,
    // recur for remaining array
    recursiveBubbleSort(array, n - 1);
}
```

Bubble sort: analysis

- Worst-case
  - Input is reversely sorted
  - Running time: $O(n^2)$
- Best-case
  - Input is already in ascending order
  - The algorithm returns after a single outer iteration
  - Running time: O(n)

Insertion sort

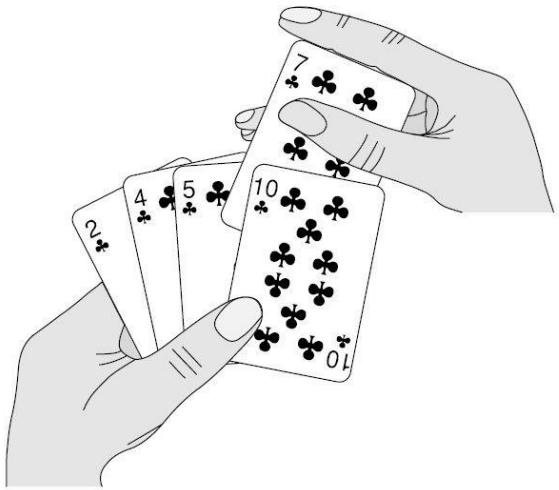


Insertion sort

- Insertion sort is a simple sorting algorithm that builds the final sorted array one item at a time.
- Insertion sort keeps a prefix of the array sorted.
- At each step, this prefix is grown by inserting the next value into it at the correct place. Eventually, the prefix is the entire array, which is therefore sorted.

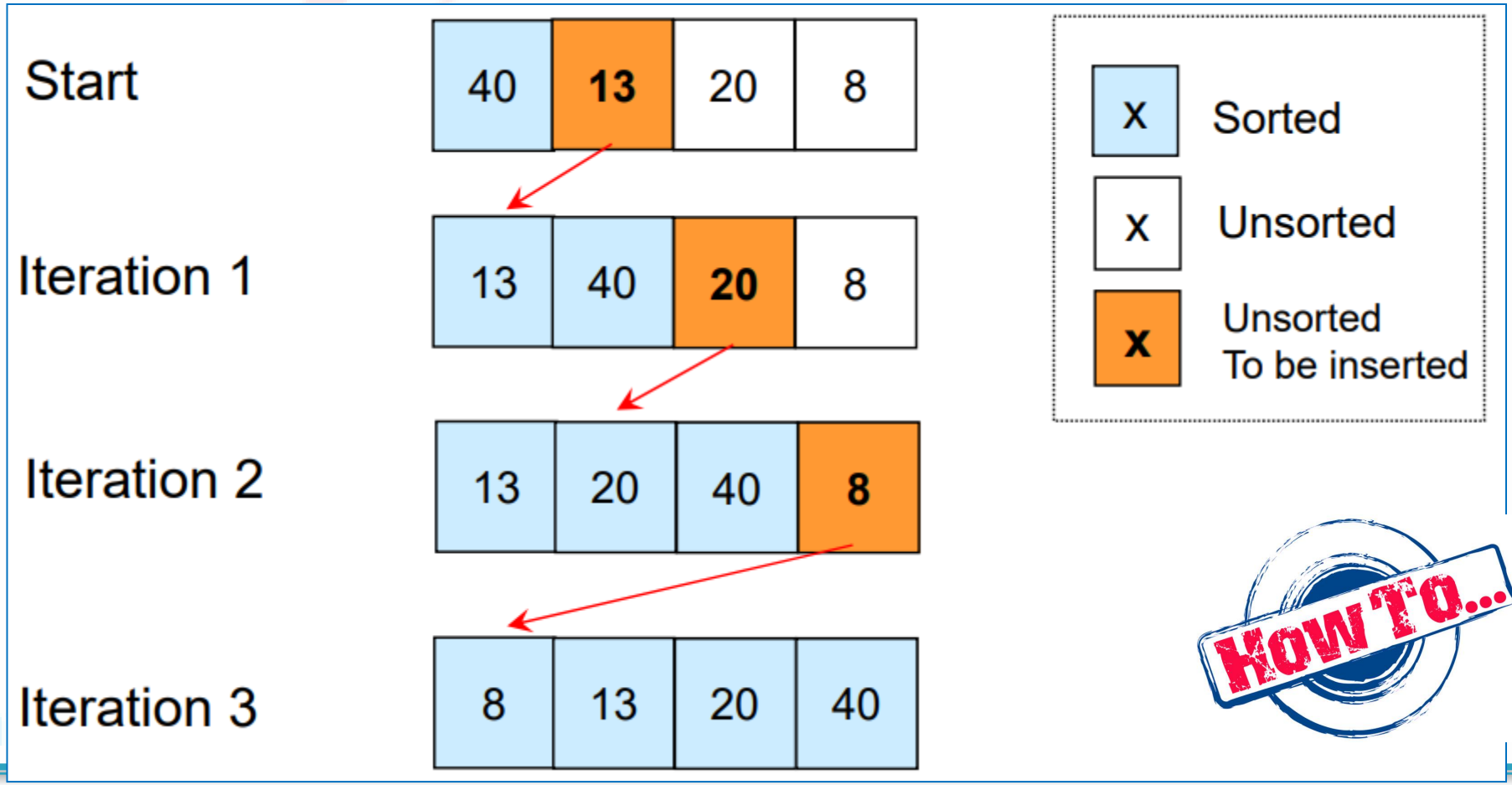
Insertion sort (cont.)

- 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

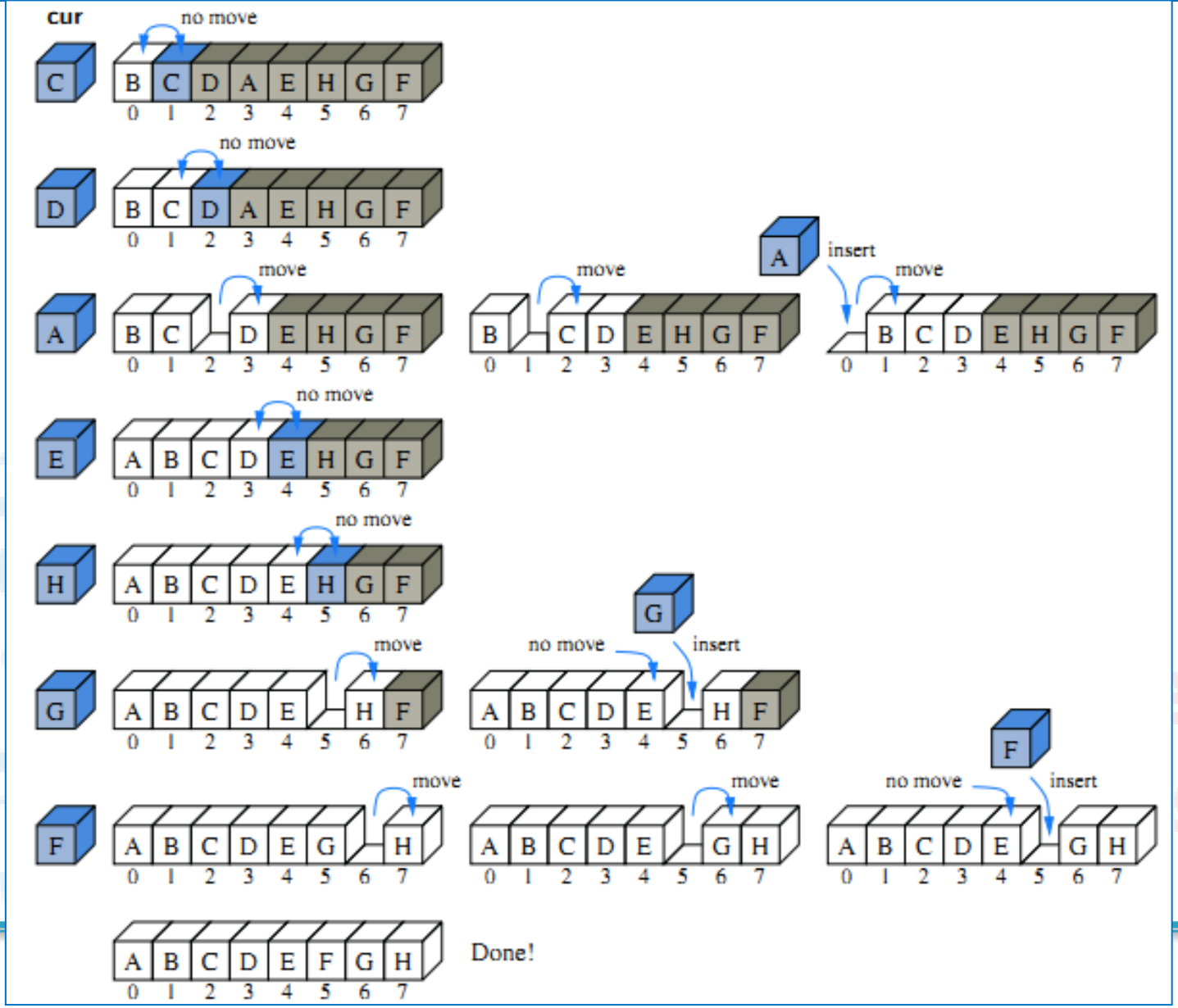


Example

- For a given array X = {40, 13, 20, 8}, sort X by ascending order?



Example


cur
no move
C
B C D A E H G F
0 1 2 3 4 5 6 7
no move
D
B C D A E H G F
0 1 2 3 4 5 6 7
move
A
B C D E H G F
0 1 2 3 4 5 6 7
move
B C D E H G F
0 1 2 3 4 5 6 7
A
insert
move
B C D E H G F
0 1 2 3 4 5 6 7
no move
E
A B C D E H G F
0 1 2 3 4 5 6 7
no move
H
A B C D E H G F
0 1 2 3 4 5 6 7
G
move
A B C D E H F
0 1 2 3 4 5 6 7
no move
insert
A B C D E H F
0 1 2 3 4 5 6 7
F
move
A B C D E G H
0 1 2 3 4 5 6 7
move
A B C D E G H
0 1 2 3 4 5 6 7
no move
insert
A B C D E G H
0 1 2 3 4 5 6 7
A B C D E F G H
0 1 2 3 4 5 6 7
Done!

Insertion sort: iterative approach

- Algorithm InsertionSort(X):
  - Input: An array X of n comparable elements
  - Output: The array X with elements rearranged in nondecreasing order

For k from 0 to n−1 do

Insert X[k] at its proper location within X[0], X[1], . . ., X[k].

Insertion sort: iterative approach

```
/** Insertion-sort of an array of characters into nondecreasing order */
public static void insertionSort(char[ ] data) {
  int n = data.length;
  for (int k = 1; k < n; k++) {                    // begin with second character
    char cur = data[k];                            // time to insert cur=data[k]
    int j = k;                                     // find correct index j for cur
    while (j > 0 && data[j−1] > cur) {             // thus, data[j-1] must go after cur
      data[j] = data[j−1];                         // slide data[j-1] rightward
      j−−;                                         // and consider previous j for cur
    }
    data[j] = cur;                                 // this is the proper place for cur
  }
}
```

Insertion sort: recursive approach

- Base Case:
  - If array size is 1 or smaller, then return.
- Recursive case:
  - Recursively sort first n-1 elements.
- Insert last element at its correct position in sorted array.

Insertion sort: recursive approach

```
public static void insertionSortRecursive(int[] array, int n) {
    // Base case
    if (n <= 1)
        return;
    // Sort first n-1 elements
    insertionSortRecursive(array, n - 1);
    // Insert last element at its correct position
    // in sorted array.
    int last = array[n - 1];
    int j = n - 2;
    // Move elements of array[0..i-1], that are greater than key,
    // to one position
    // ahead of their current position
    while (j >= 0 && array[j] > last) {
        array[j + 1] = array[j];
        j--;
    }
    array[j + 1] = last;
}
```

Insertion sort: analysis

- Best case:
  - The input is already sorted
  - Running time: O(n)

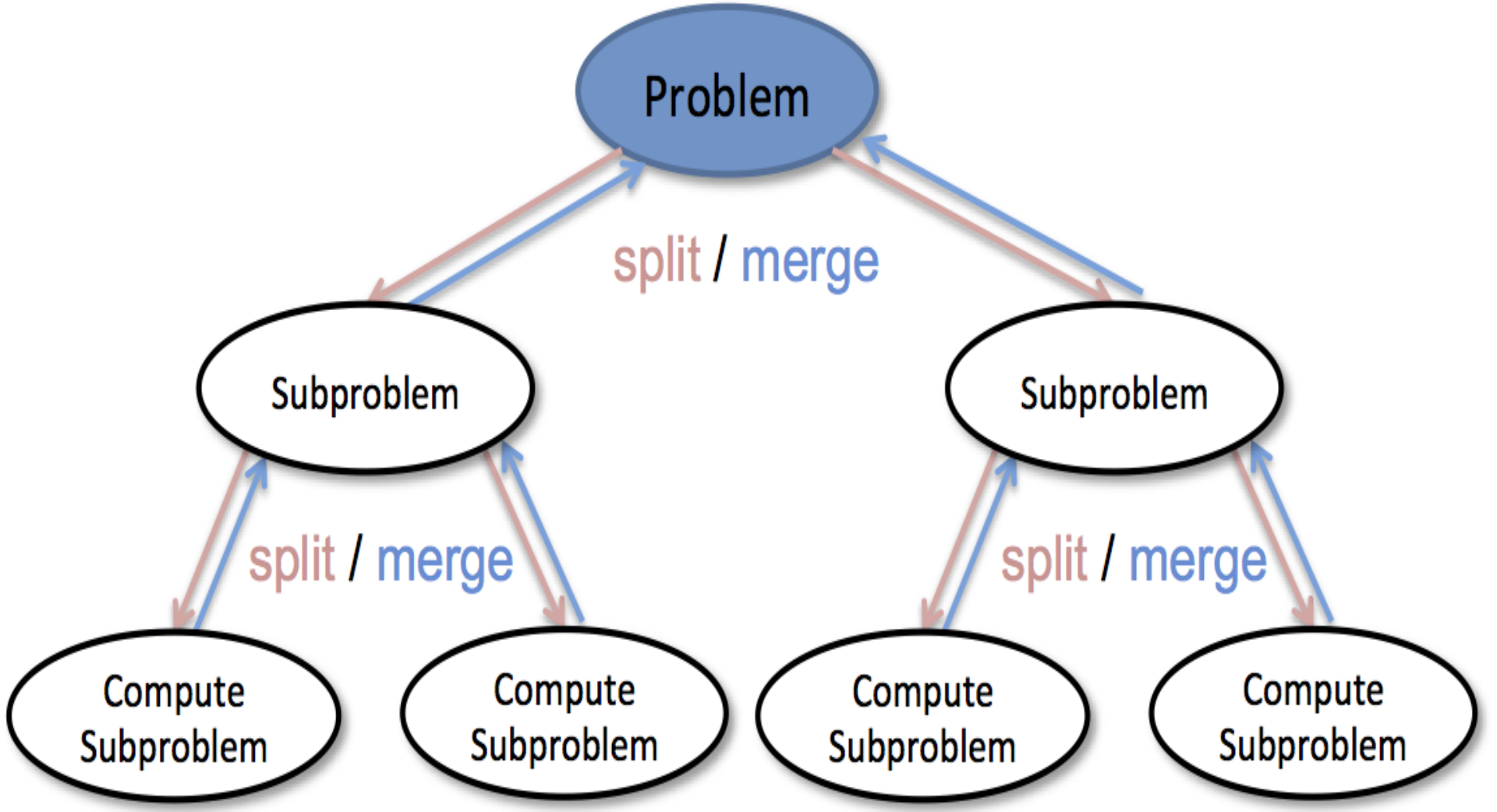
- Worst case:
  - The input is reversely sorted
  - Running time: $O(n^2)$

Merge sort

Divide-and-Conquer

- **Divide**:
  - If the input size is smaller than a certain threshold (i.e., one or two elements), solve the problem directly using a straightforward method and return the solution so obtained.
  - Otherwise, divide the input data into two or more disjoint subsets.
- **Conquer**:
  - Recursively solve the sub problems associated with the subsets.
- **Combine**:
  - Take the solutions to the sub problems and merge them into a solution to the original problem

Divide-and-Conquer



Merge sort

- To sort a sequence S with n elements using the three divide-and-conquer steps:
- **Divide**:
  - If S has zero or one element, return S immediately; it is already sorted.
  - Otherwise (S has at least two elements), divides the n elements into two sub sequences (S_1, S_2) having size n / 2 elements each.
- **Conquer**: Recursively sort sequences S_1 and S_2.
- **Combine**: merge the two sub sequences S_1, S_2 which are sorted to produce the sorted answer.

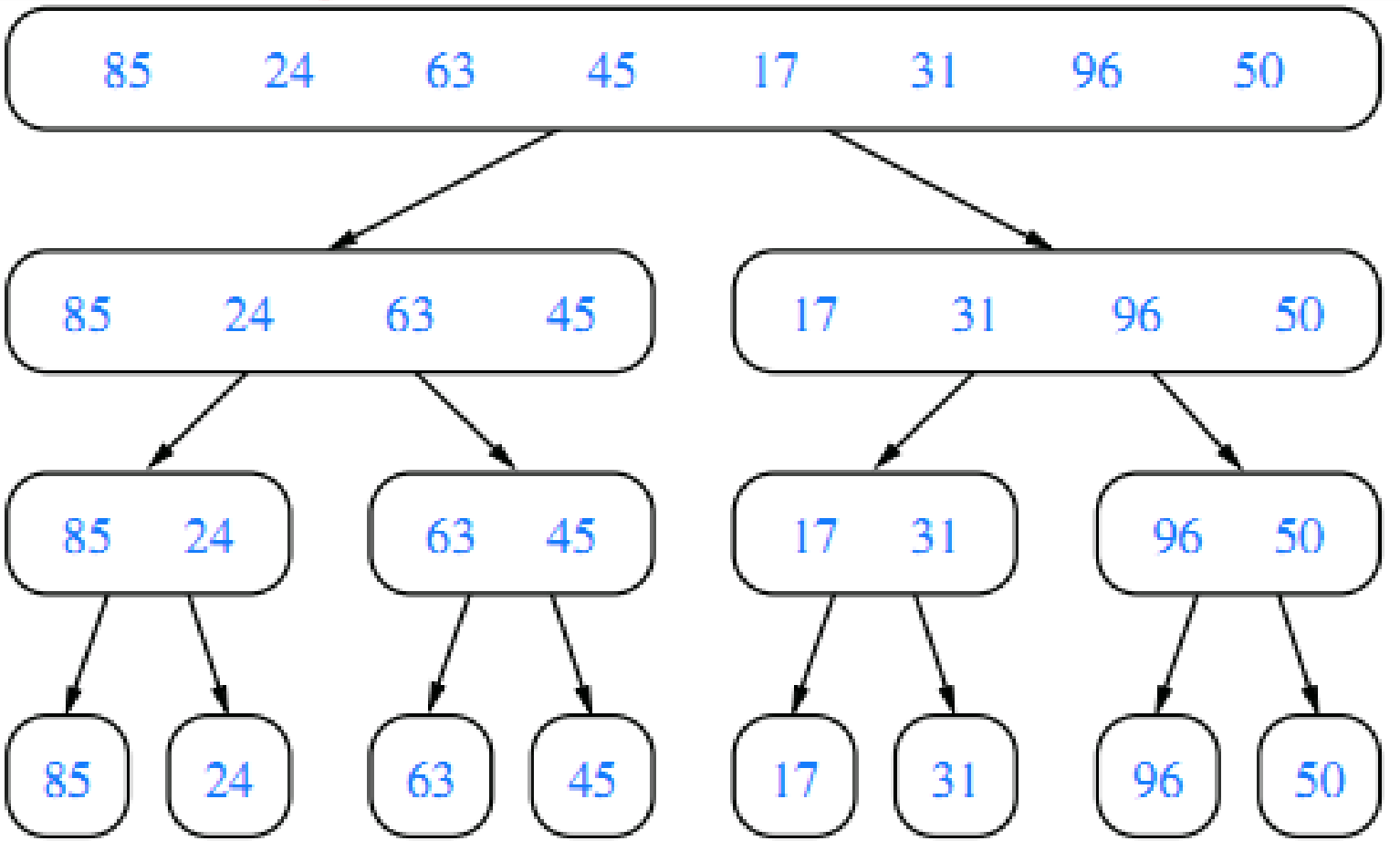
Merge sort: idea

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

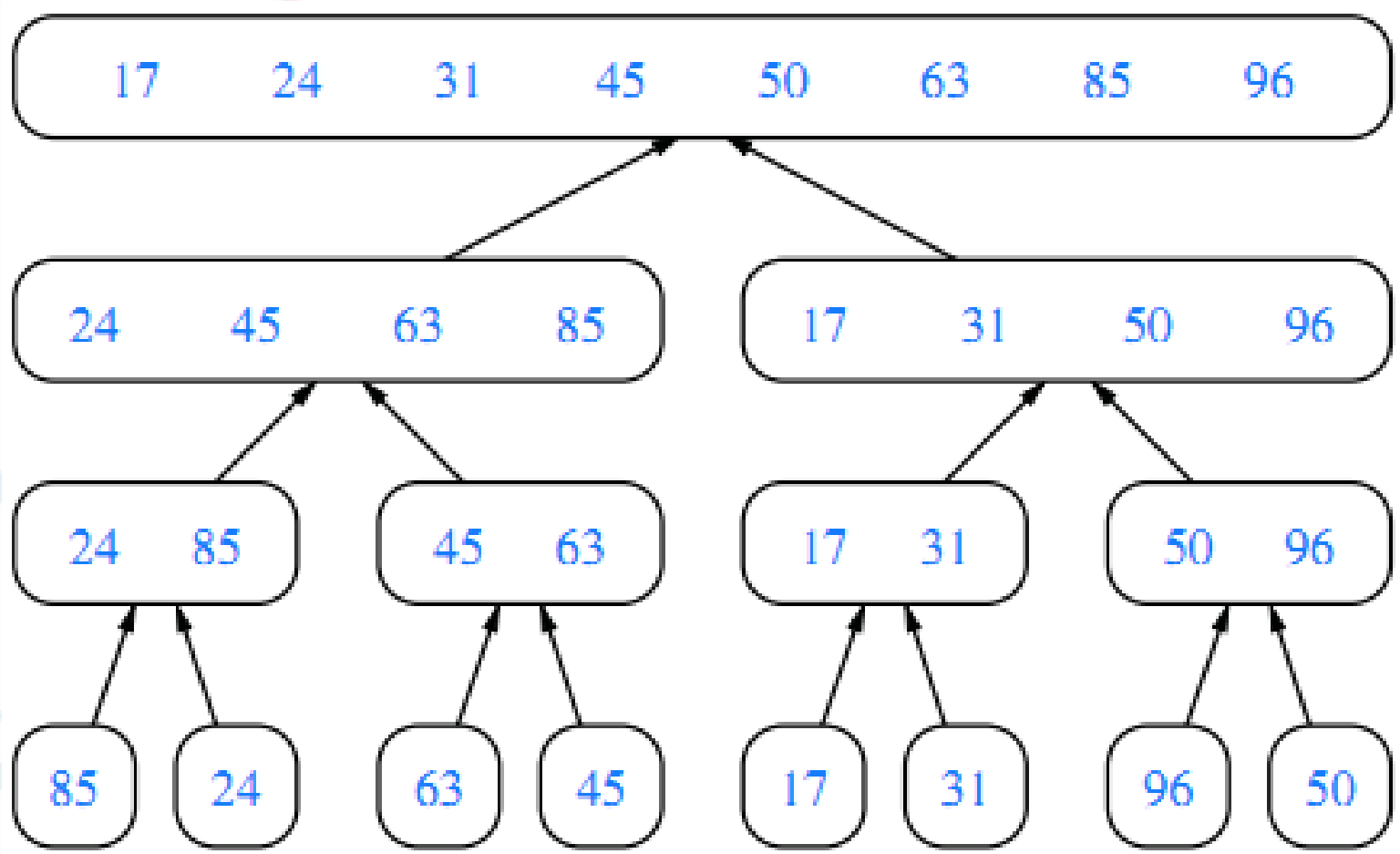
Example

- How to sort the following array using Merge sort?

Example: Step 1

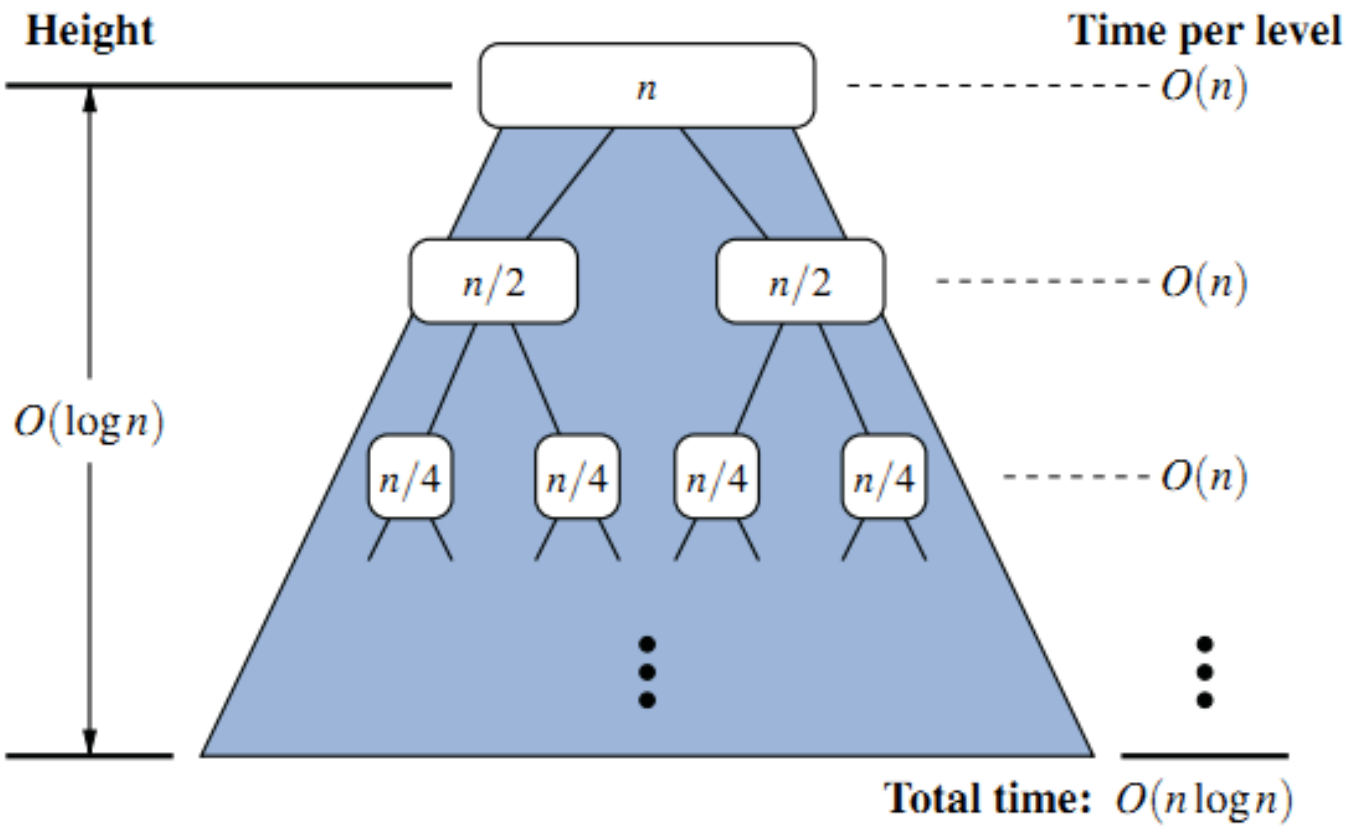


Example: Step 2



Running Time

- This algorithm splits the items to be sorted into 2 groups, recursively sorts each group, and merges them into a final sorted array. The Run time is **O(n log n).**



Quick sort

QUICK SORT

```
public static void quicksort(char[] items, int left, int right)
{
   int i, j;
   char x, y;

   i = left; j = right;
   x = items[(left + right) / 2];

   do
   {
      while ((items[i] < x) && (i < right)) i++;
      while ((x < items[j]) && (j > left)) j--;

      if (i <= j)
      {
         y = items[i];
         items[i] = items[j];
         items[j] = y;
         i++; j--;
      }
   } while (i <= j);

   if (left < j) quicksort(items, left, j);
   if (i < right) quicksort(items, i, right);
}
```

Mergesort limitations?

- Mergesort divides the input array X[low..high] into two pieces of similar size without looking at the contents:
  - X_1[low..mid]
  - X_2[(mid + 1)..high]

  (mid = (low + high) / 2)
- After sorting the two pieces, we have to merge them, costing us an extra **n** operations.
- Can we arrange it so that we don't have to merge at the end?

Quick sort: idea

- **Divide**:
  - If S has at least two elements (nothing needs to be done if S has zero or one element), select a specific element ***x*** from S (pivot). Then, put elements in S into three sequences:
  - L, storing the elements in S less than x
  - E, storing the elements in S equal to x
  - G, storing the elements in S greater than x

Quick sort: idea (cont.)

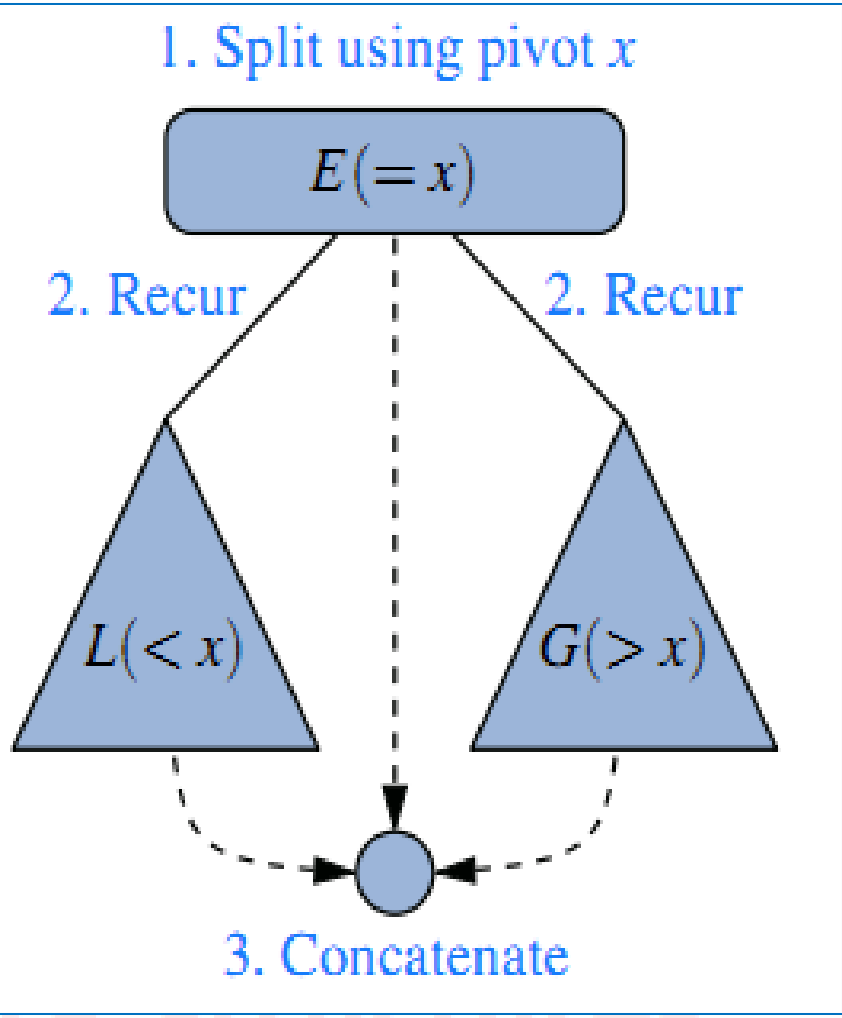
- **Conquer**: Recursively sort sequences L and G.

- **Combine**: Put back the elements into S in order by first inserting the elements of L, then those of E, and finally those of G.

Quick sort

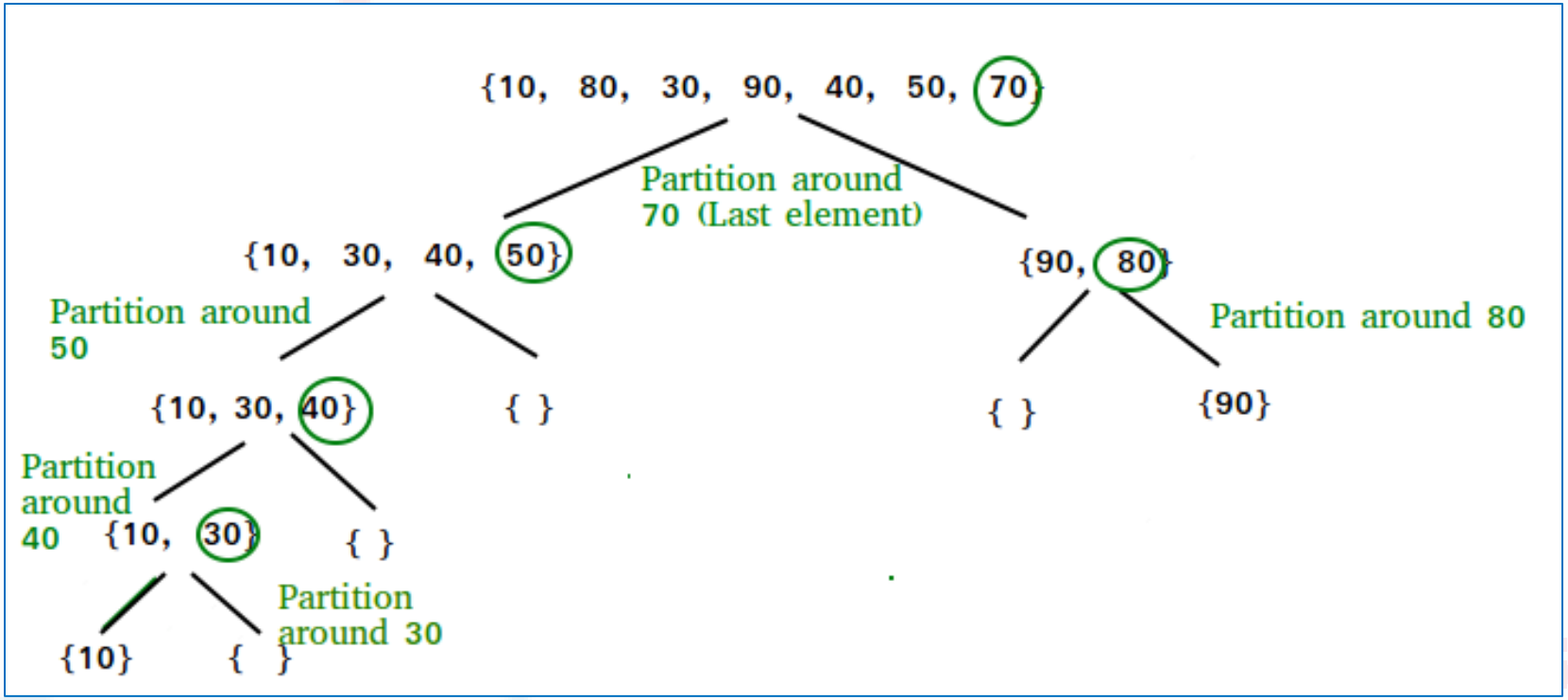
```
function quicksort(array)
  var L, E, G – (array or list)
  if length(array) ≤ 1
  return array
 select a pivot value from array
for each x in array
      if x < pivot then append x to L
      if x = pivot then append x to E
      if x > pivot then append x to G
  return concatenate(quicksort(L), E, quicksort(G))
```

partition



HOW TO CHOOSE **PIVOT**?

Example



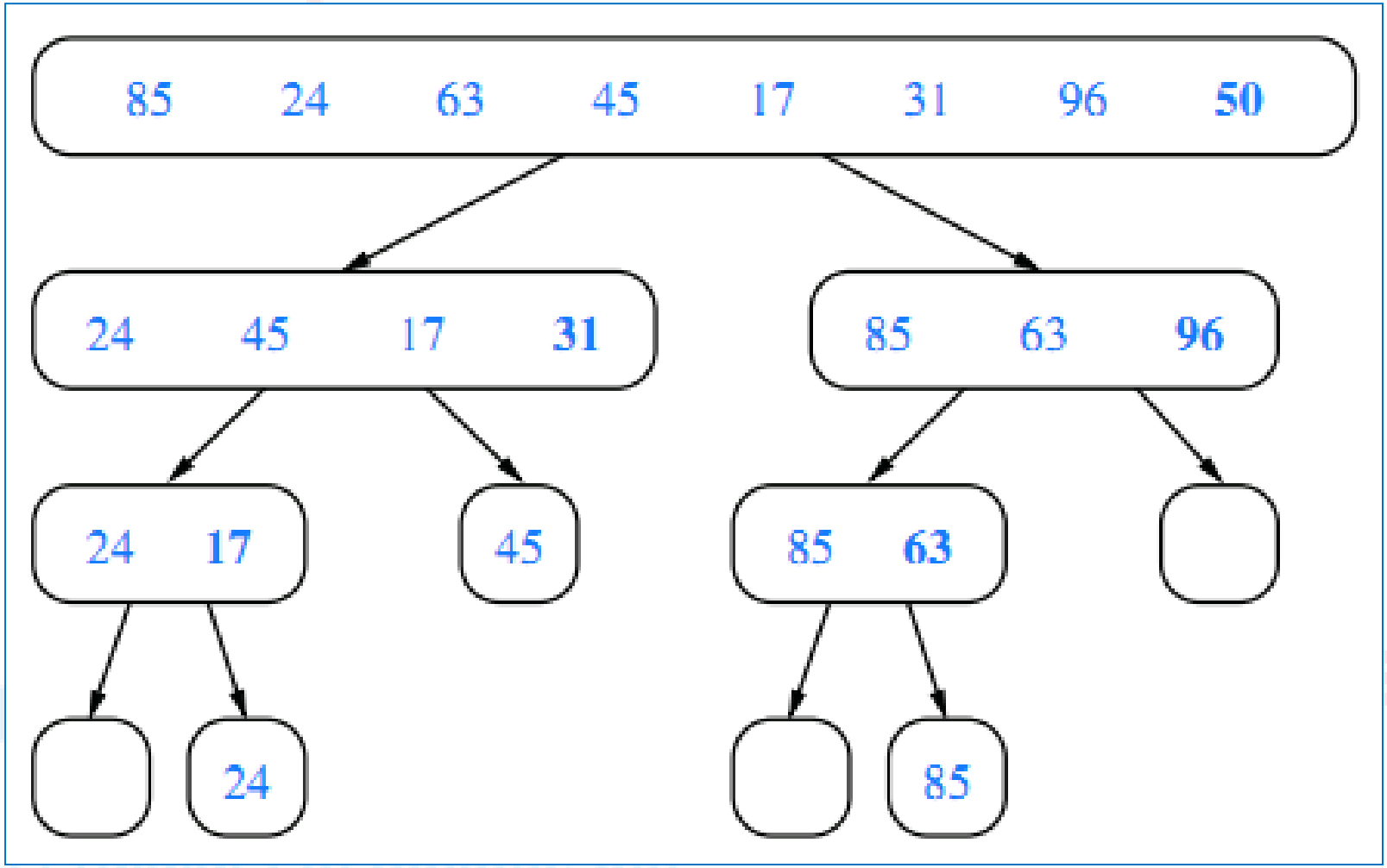
How to pick Pivot?

- There are many different versions of quickSort that pick pivot in different ways:
  - First element
  - Last element
  - Random element
  - Median-of-three elements
    - Pick three elements (leftmost, rightmost, center), and find the median *x* of these elements.

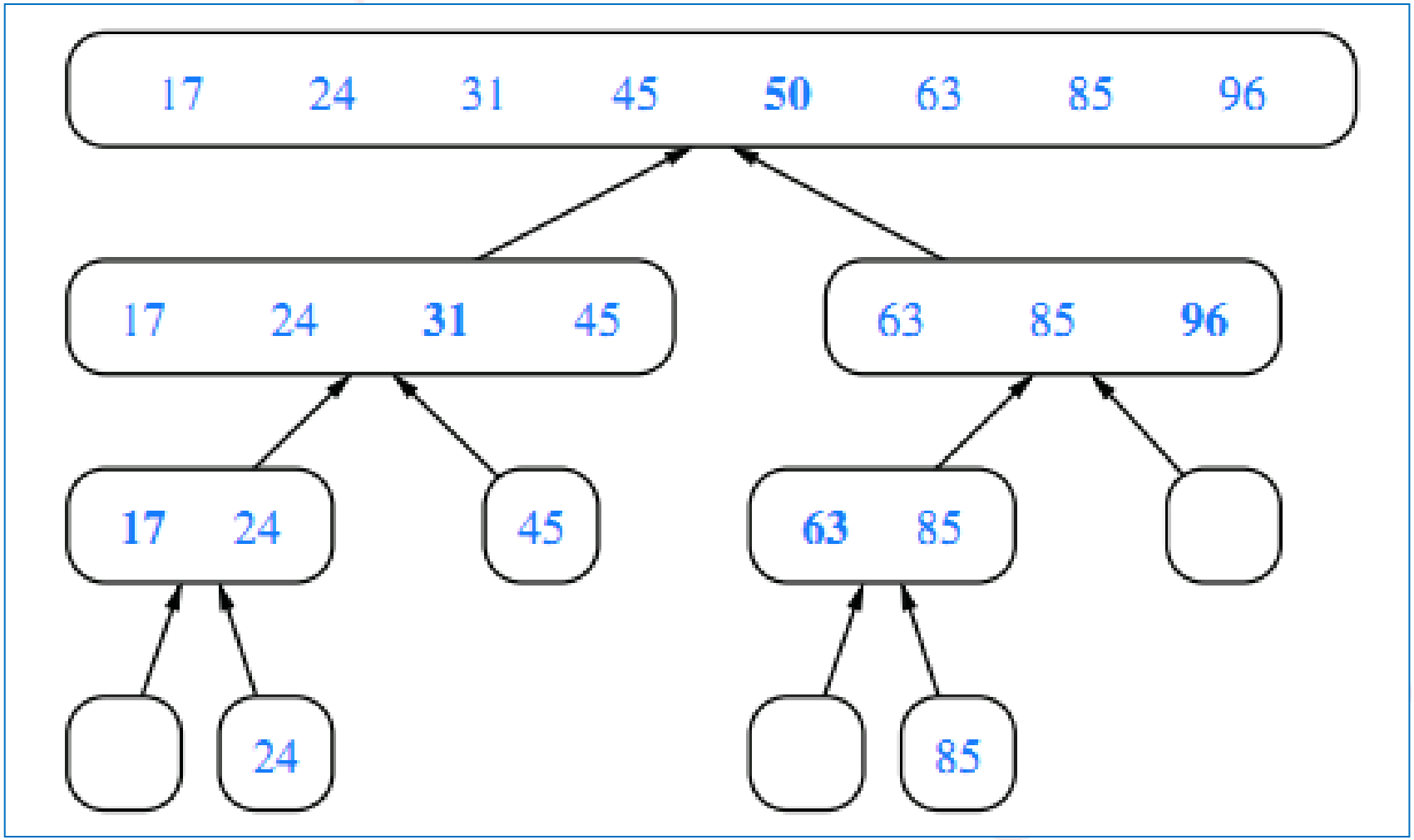
Median-of-three

- Let input `S = {6, 1, 4, 9, 0, 3, 5, 2, 7, 8}`
- `left=0` and `S[left] = 6`
- `right=9` and `S[right] = 8`
- `center = (left+right)/2 = 4` and `S[center] = 0`
- Pivot
  - = Median of `S[left]`, `S[right]`, and `S[center]`
  - = median of 6, 8, and 0
  - = `S[left] = 6`

Example

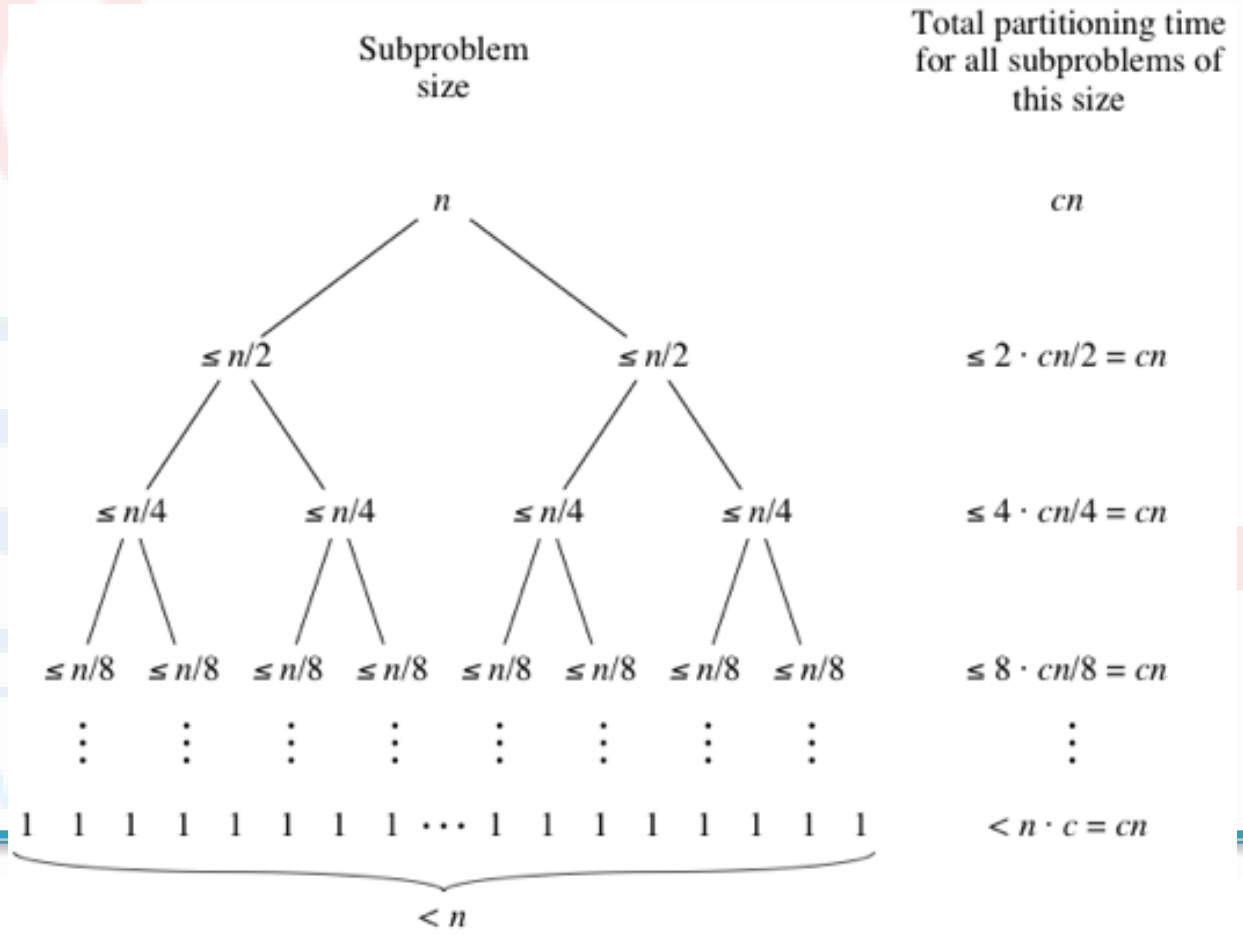


Example



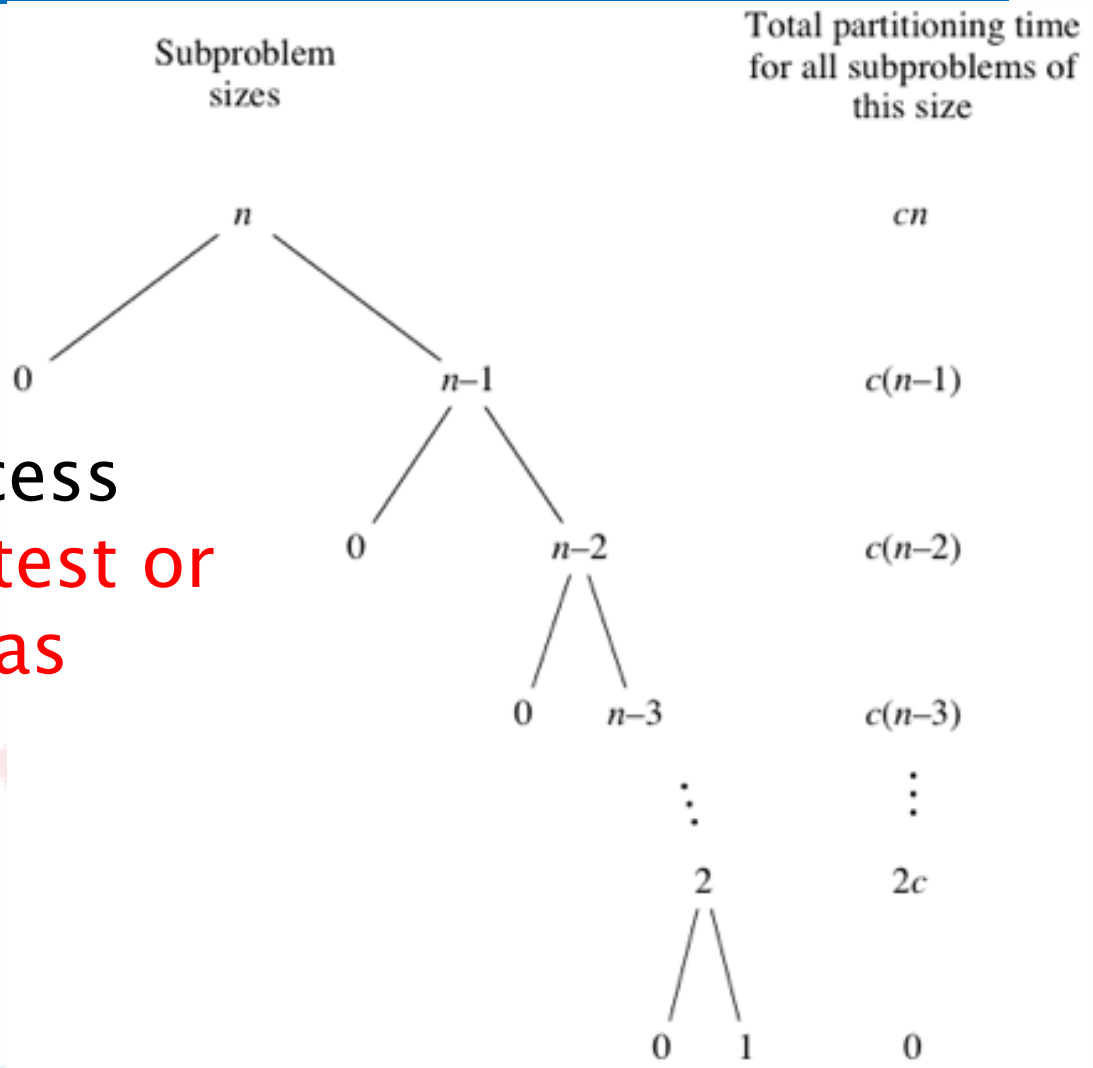
The best case

- The partitions are as evenly balanced as possible: their sizes either are equal or are within 1 of each other



The worst case

- The partition process always picks greatest or smallest element as pivot.



Quick sort: analysis

- Advantage:
  - Fastest known sorting algorithm in practice
  - Average case running time: O(n log n)
  - Best case running time: O(n log n)

- Disadvantage:
  - Sorts $O(n^2)$ in the worst case
  - The worst case doesn't happen often … sorted

FACULTY OF INFORMATION TECHNOLOGY

Java
Data Structures