SE102 ADT and Problem solving

Announcement

- Midterm exam date
- 10th Friday January 8-11am Room ILC-B302, B303
- 3 Hours (closed book)
- Dictionary is allowed
- Grading is curved base.
- Friday is day-off for us (voted)

Agenda

- Quicksort
- Big-O
- Problem solving – Java String - Palindrome
- Midterm review
  - *Recursive
  - **Sample of code (Will upload to the Mango system)

Quick sort

- Another **divide and conquer** strategy.
- The basic idea is to find a "*pivot*" item in the array to compare all other items against.
- Then shift items such that all of the items before the pivot are less than the pivot value and all the items after the pivot are greater than the pivot value.
- After that, recursively perform the same operation on the items before and after the pivot.

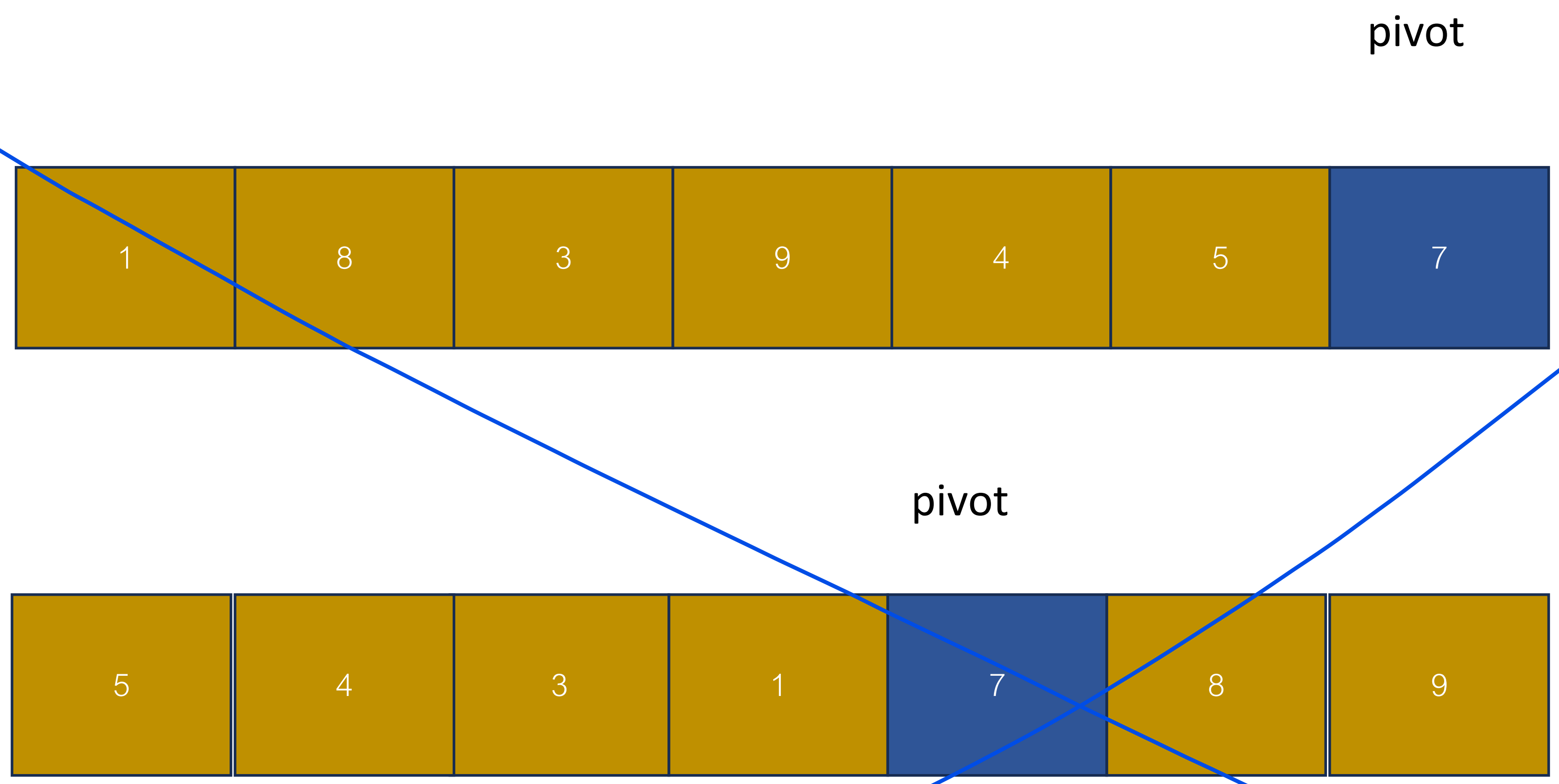
Quick Sort (Algorithm)

- Quicksort uses a divide and conquer strategy, but does not require the O(N) extra space that MergeSort does
  - Partition array into left and right sub-arrays
    - Choose an element of the array, called pivot
    - the elements in left sub-array are all less than pivot
    - elements in right sub-array are all greater than pivot
  - Recursively sort left and right sub-arrays
  - Concatenate left and right sub-arrays in O(1) time

The Steps of Quick Sort

1	8	3	9	4	5	7

The Steps of Quick Sort - Partitioning



Do quick sort at left part

Do quick sort at right part

The Steps of Quick Sort

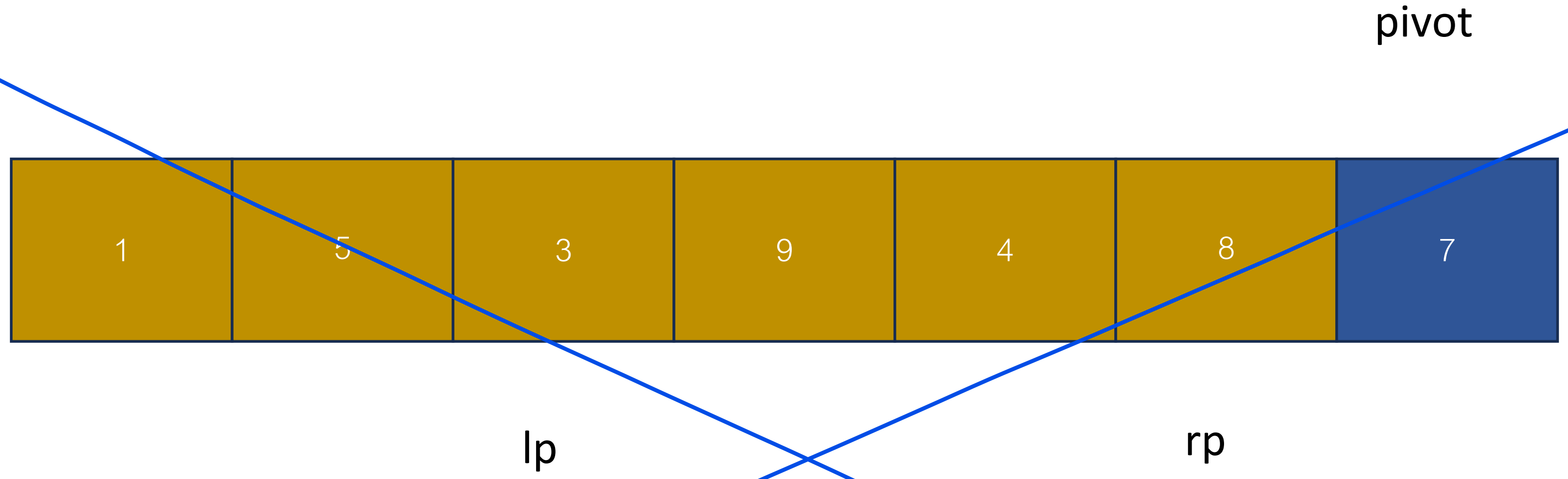
pivot

1	5	3	9	4	8	7

lp rp

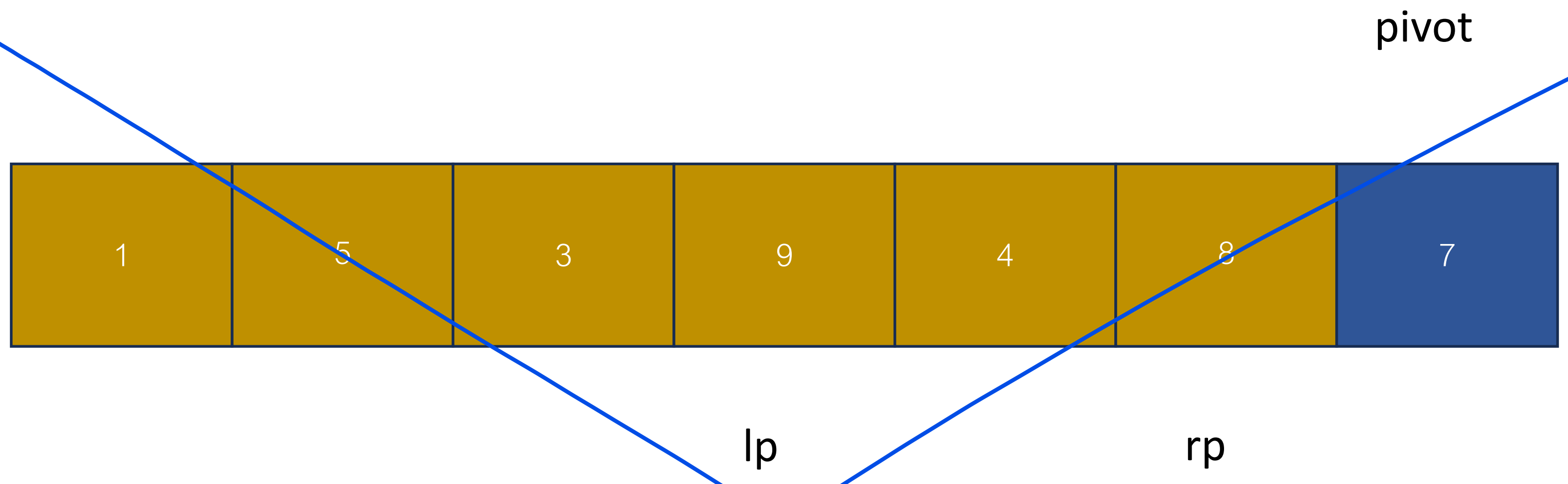
Move lp to the right

The Steps of Quick Sort



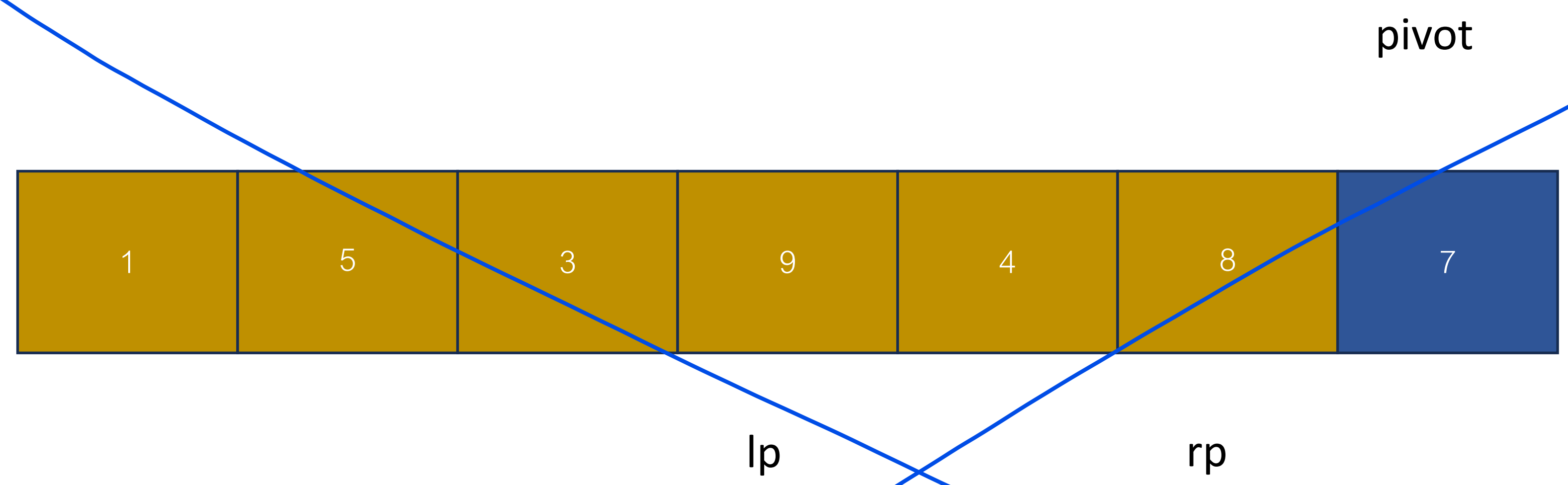
lp < 7, move lp to the right

The Steps of Quick Sort



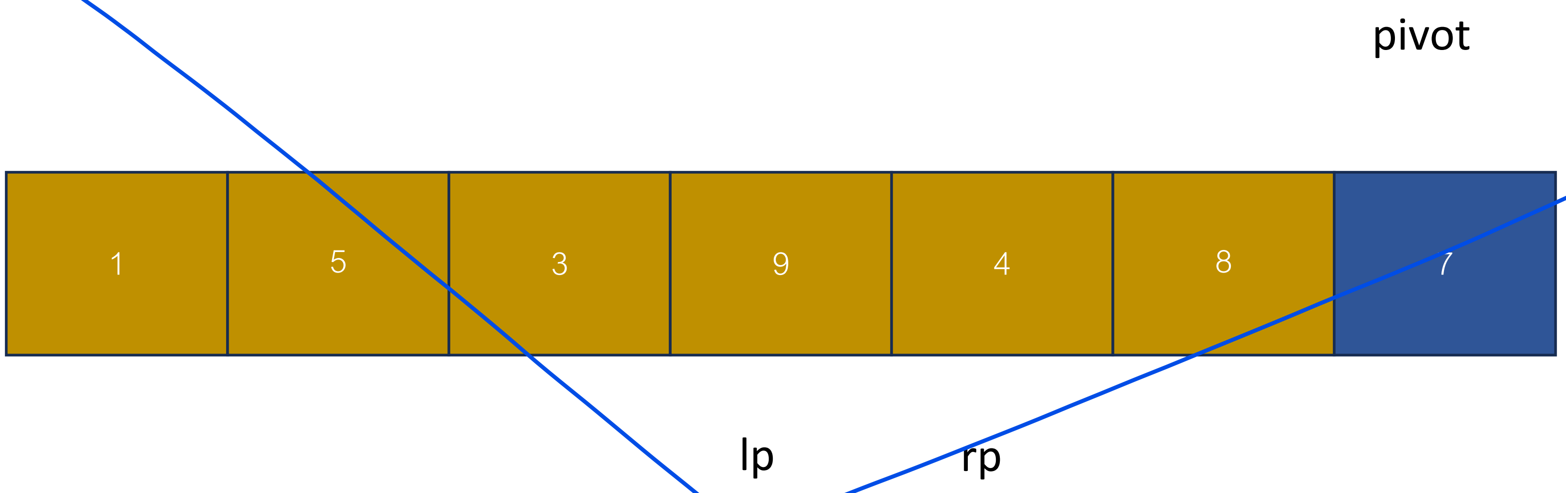
lp > 7, stop

The Steps of Quick Sort

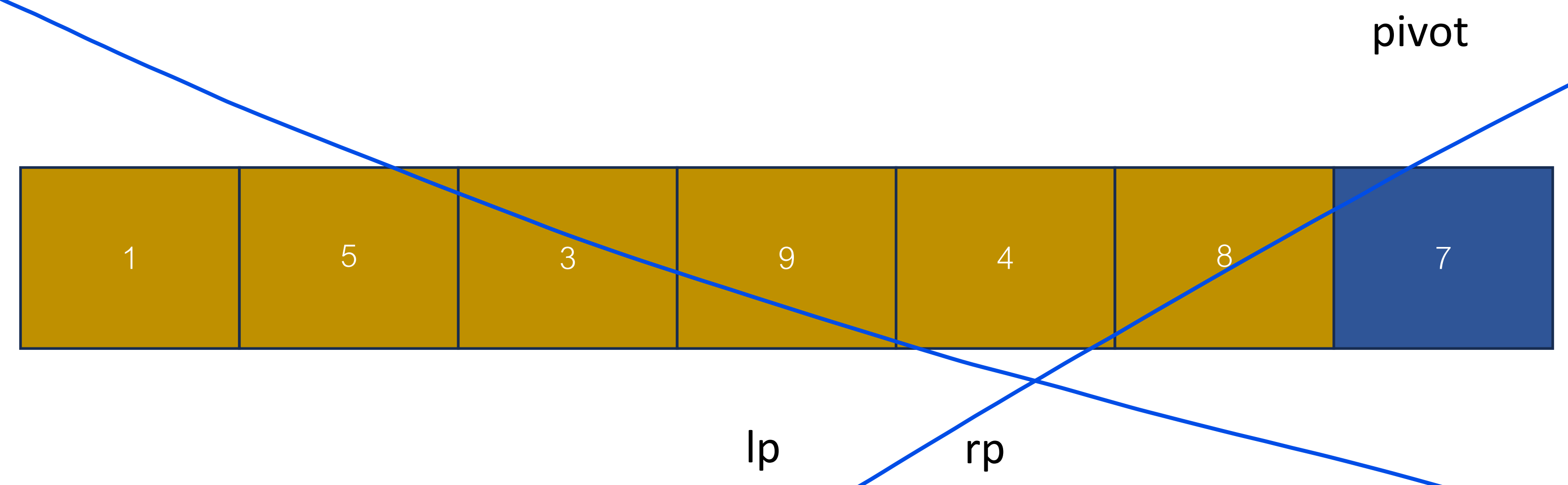


rp > 7, move to the left

The Steps of Quick Sort

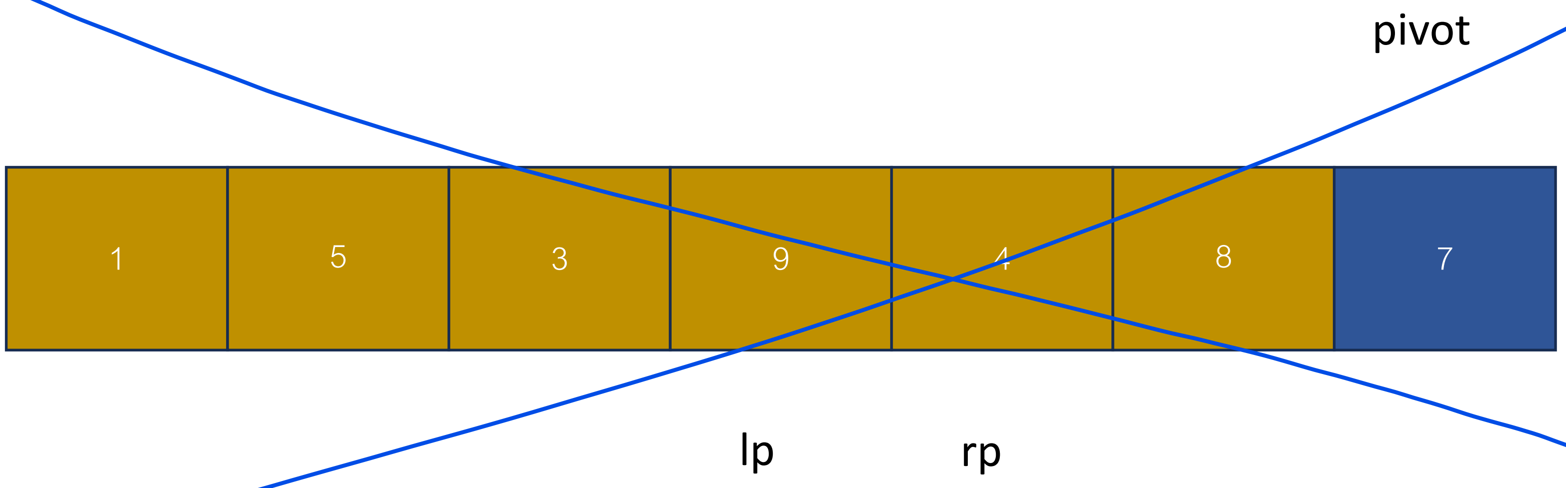


The Steps of Quick Sort



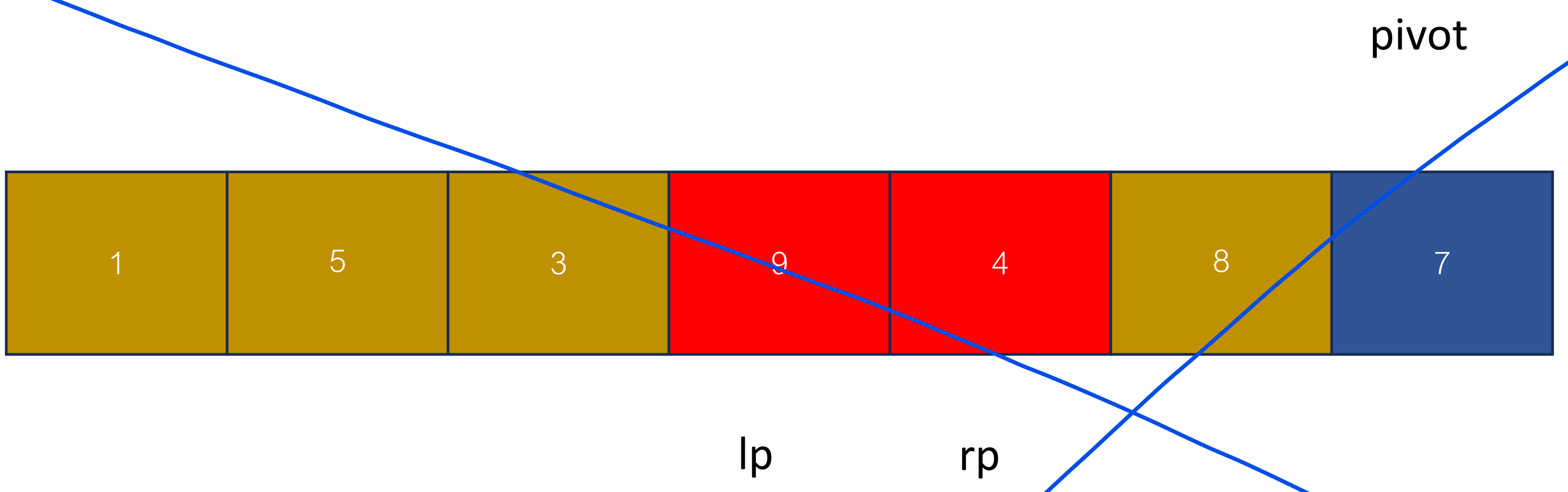
rp < 7, stop here

The Steps of Quick Sort



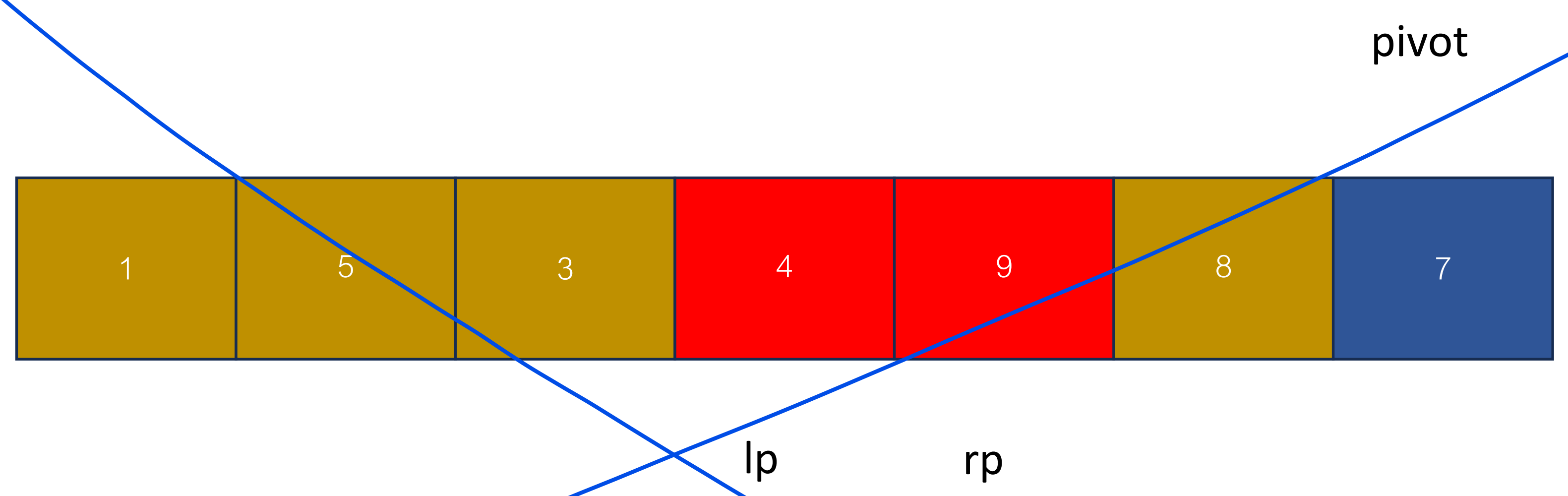
Swap

The Steps of Quick Sort



Swap

The Steps of Quick Sort



The Steps of Quick Sort

1	5	3	4	9	8	7

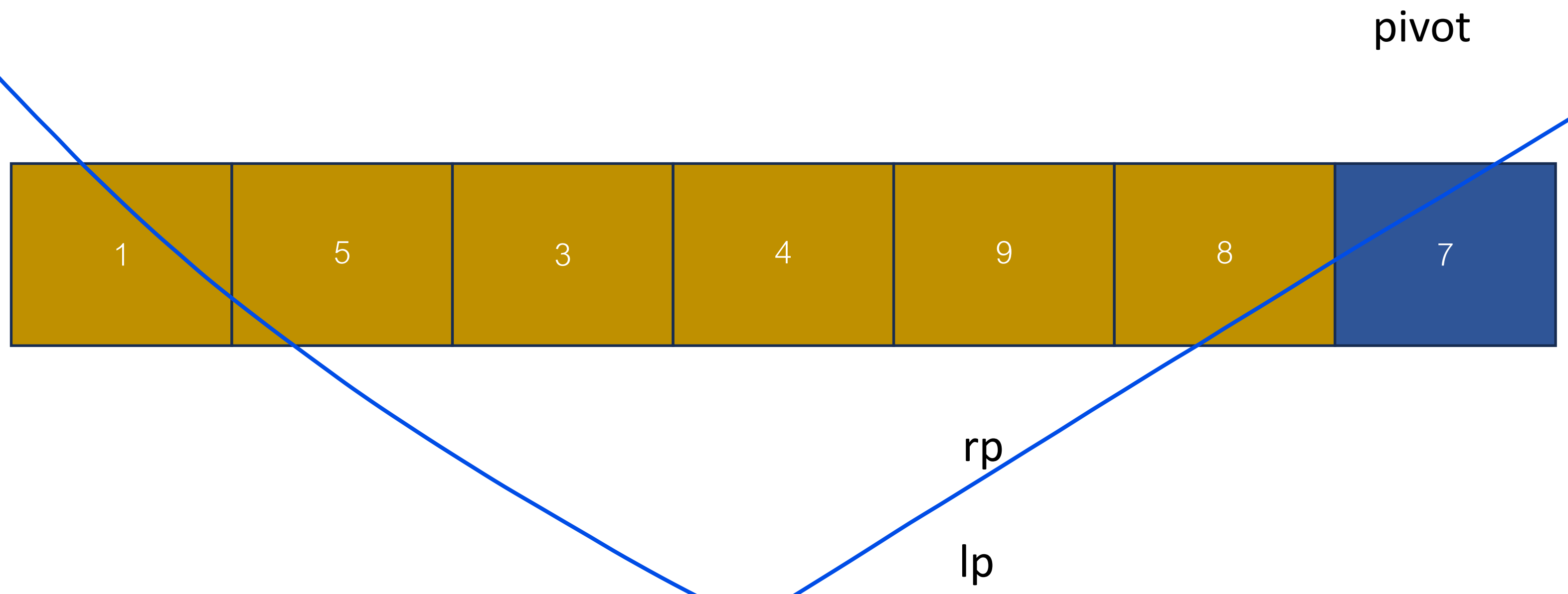
pivot

lp

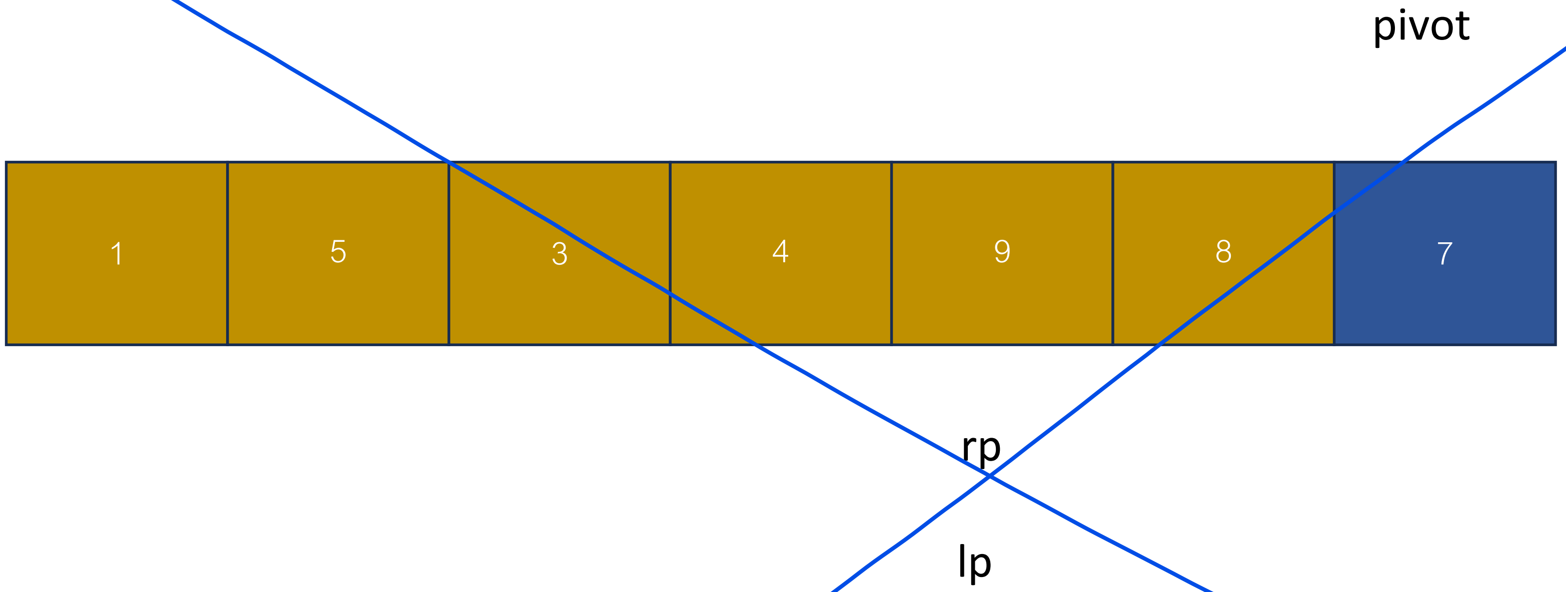
rp

Then move lp to the right

The Steps of Quick Sort



The Steps of Quick Sort



lp = rp, the swap this location with pivot

The Steps of Quick Sort

pivot

rp

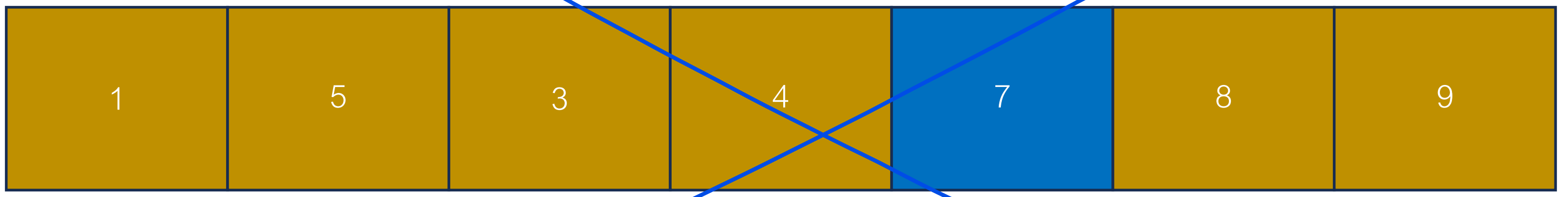
lp

lp = rp, the swap this location with pivot

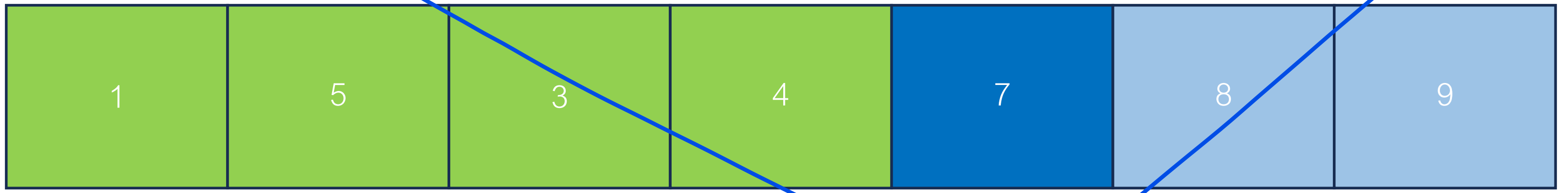
The Steps of Quick Sort

pivot

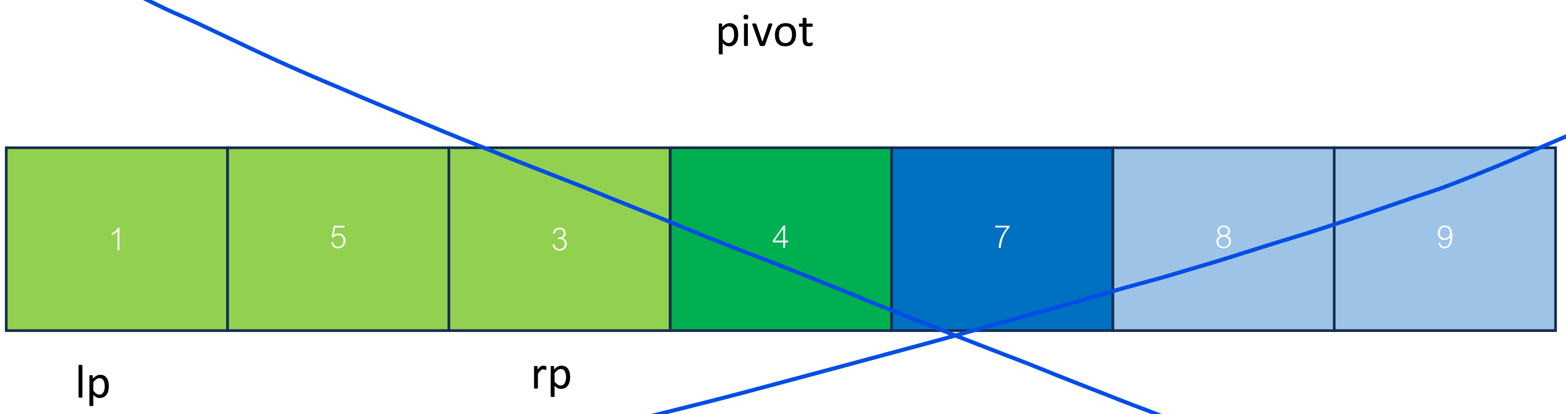
The Steps of Quick Sort



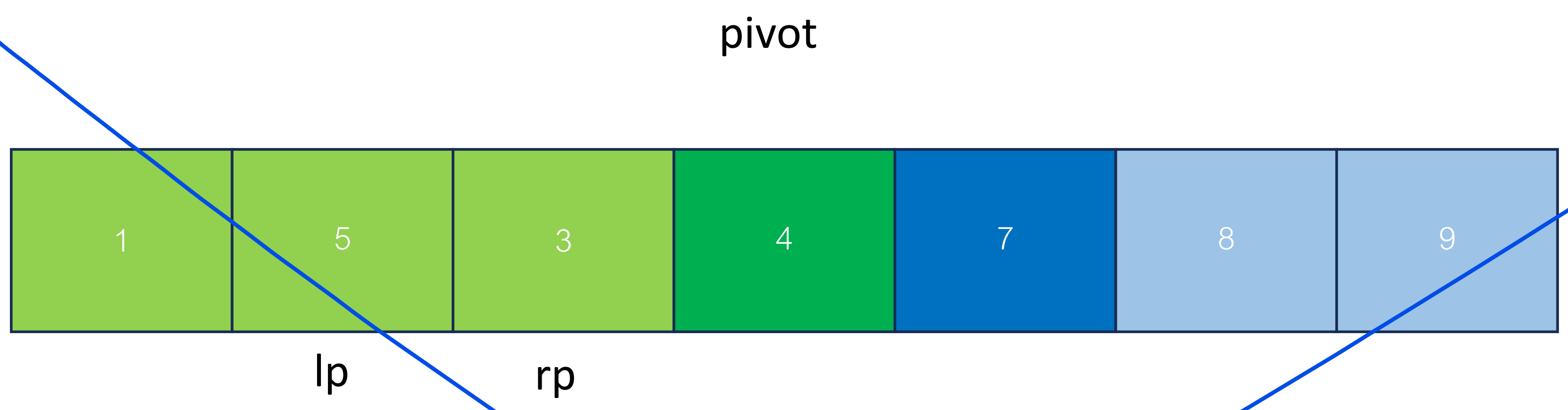
The Steps of Quick Sort



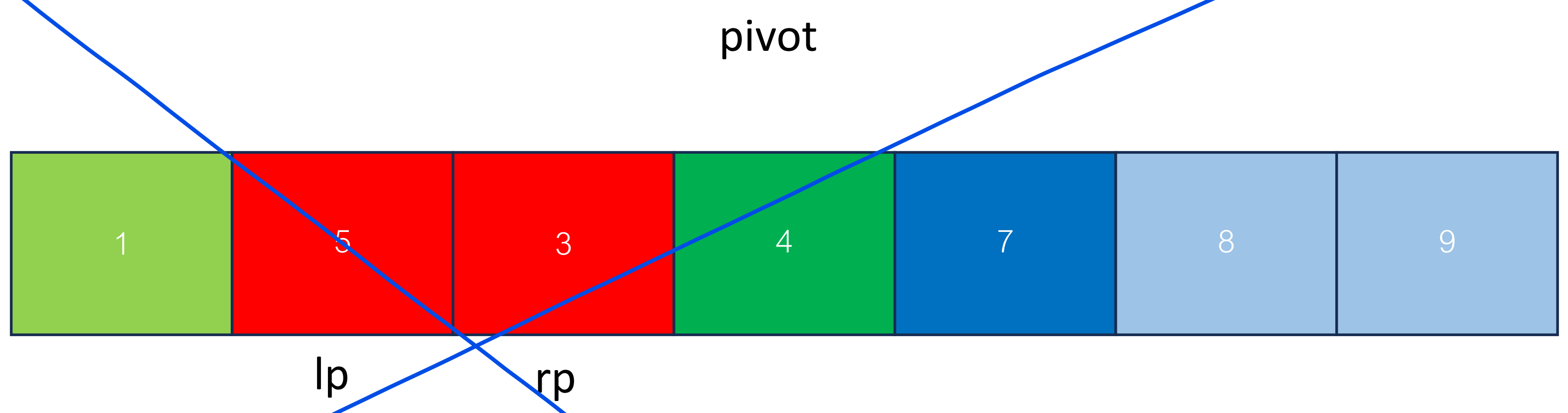
The Steps of Quick Sort



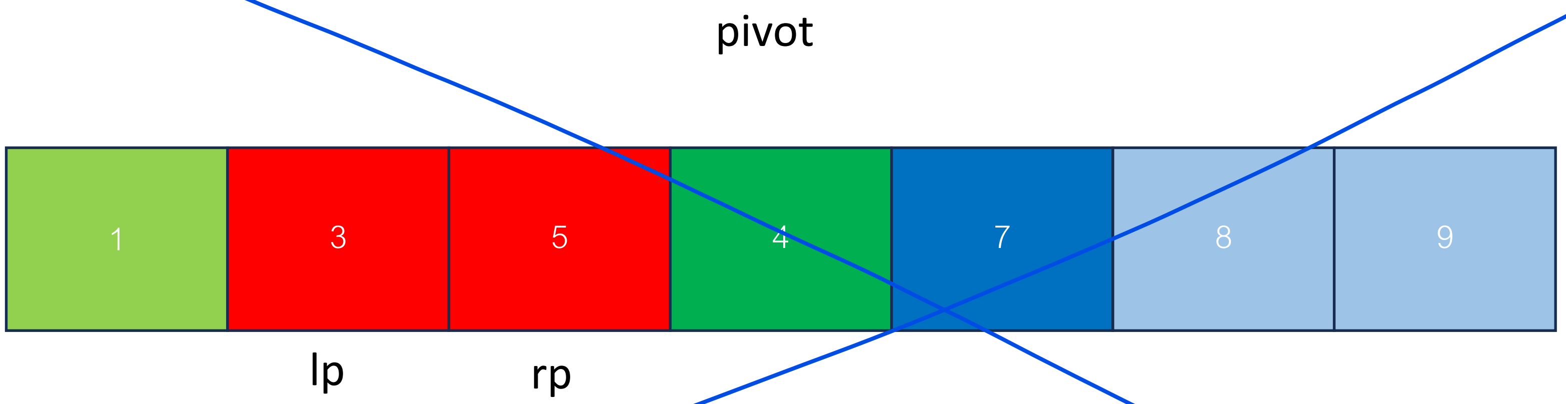
The Steps of Quick Sort



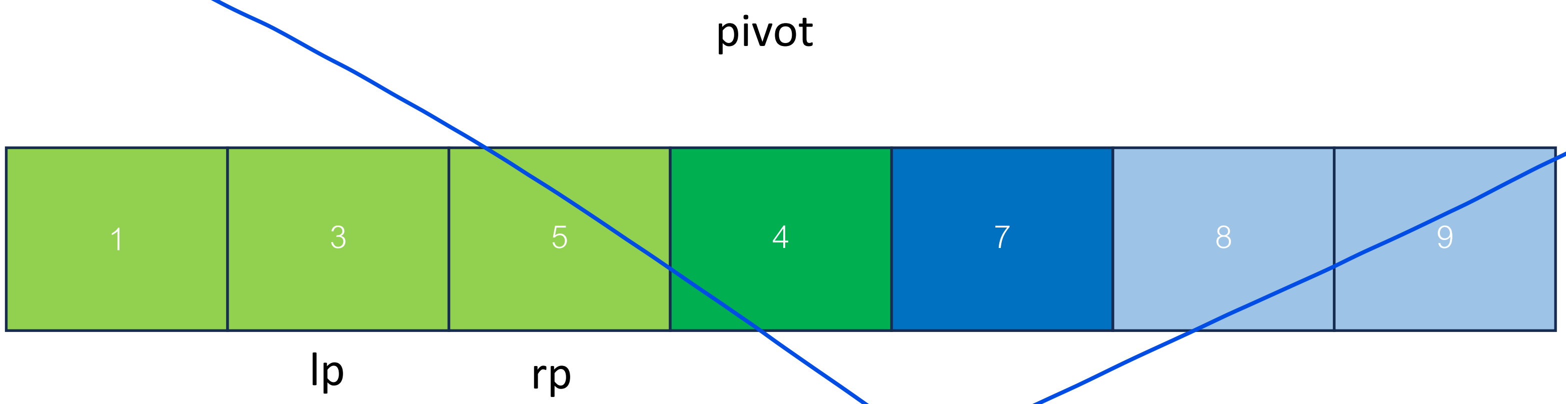
The Steps of Quick Sort



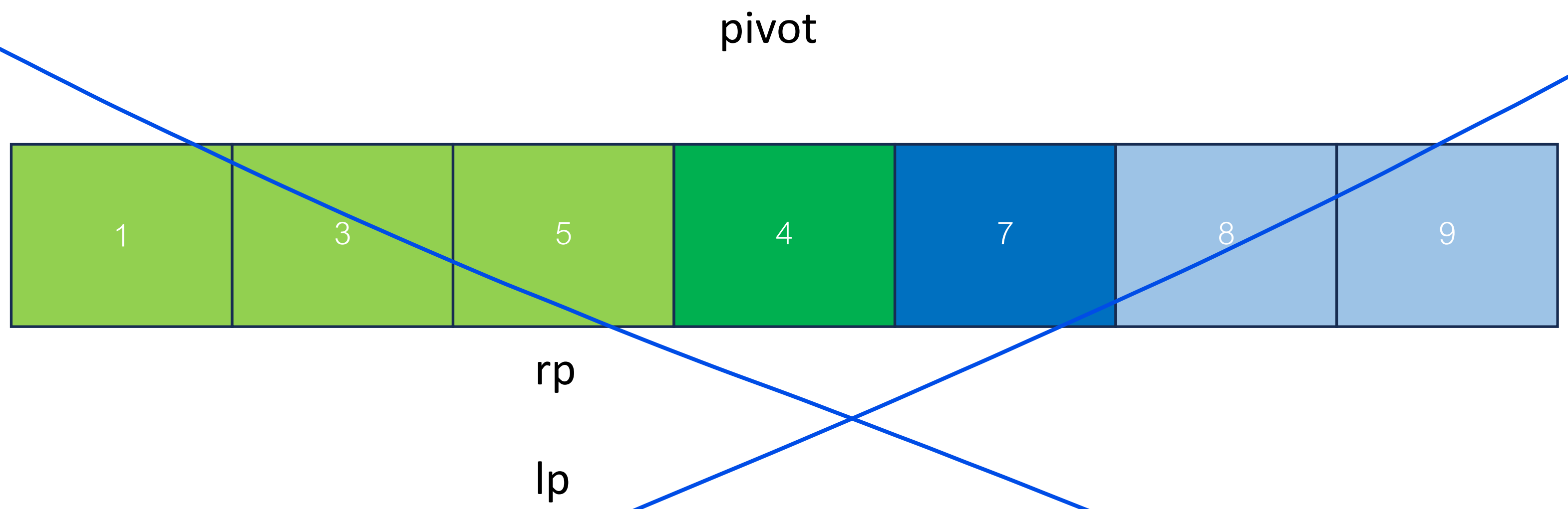
The Steps of Quick Sort



The Steps of Quick Sort



The Steps of Quick Sort



The Steps of Quick Sort

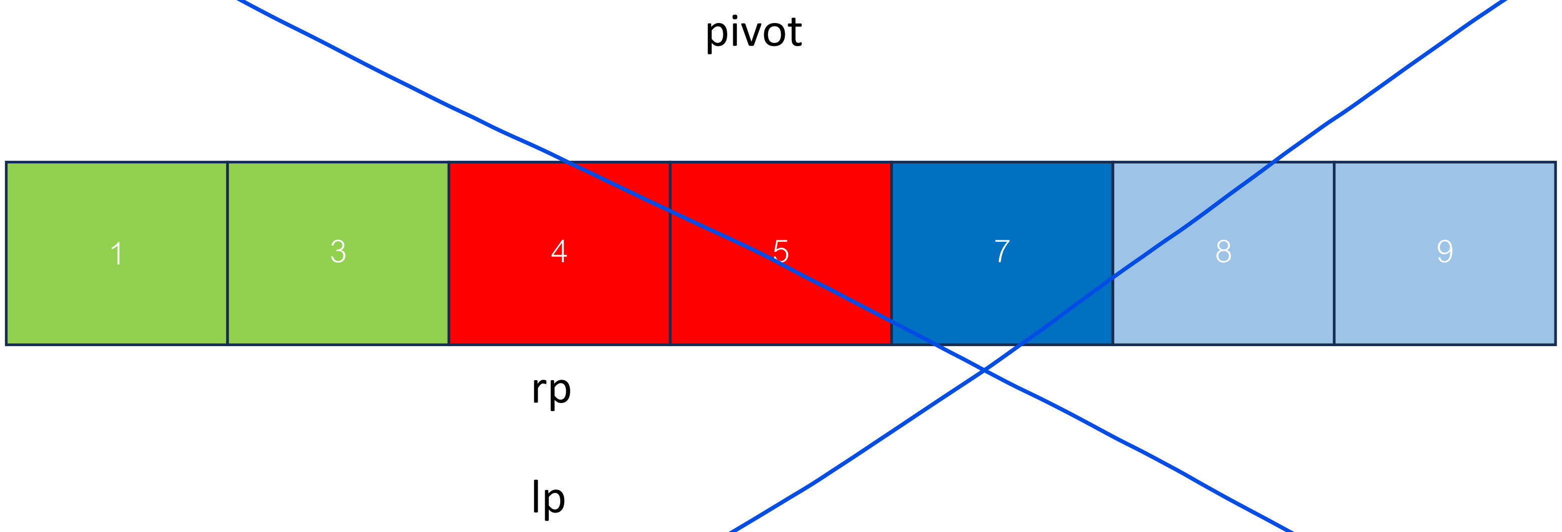
pivot

1	3	5	4	7	8	9

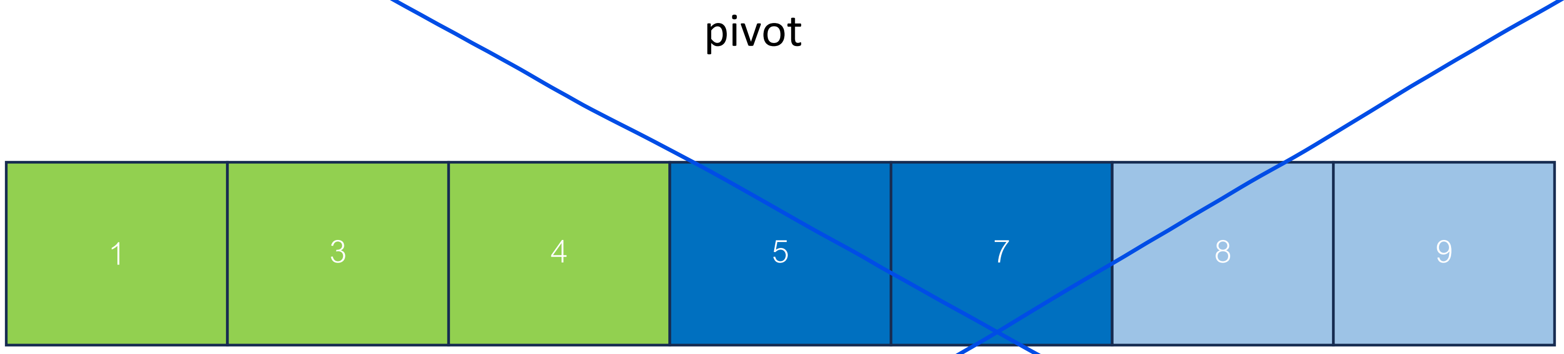
rp

lp

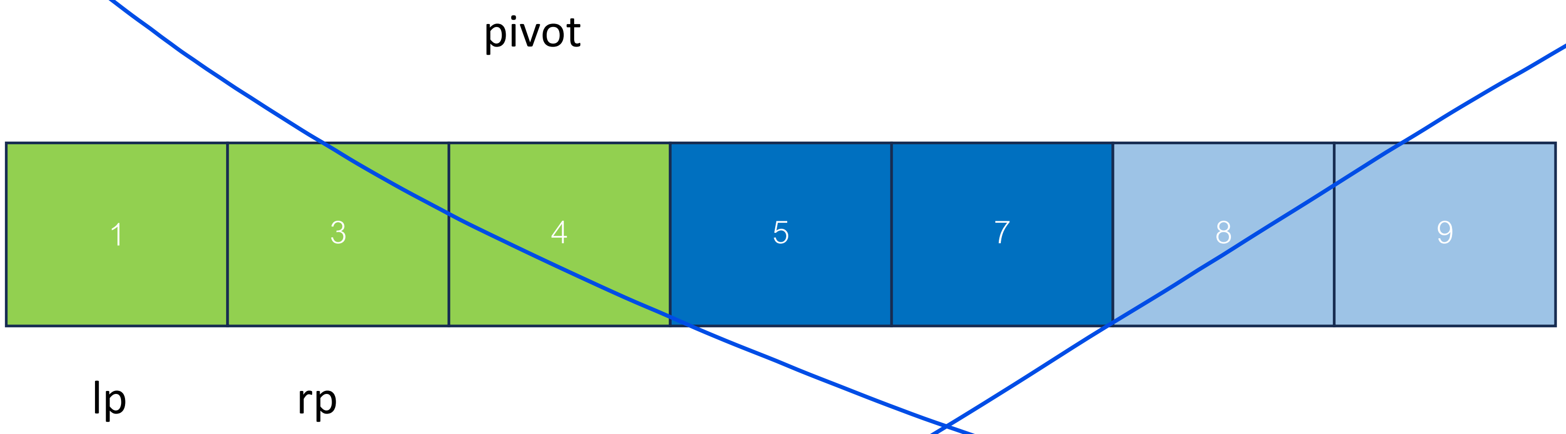
The Steps of Quick Sort



The Steps of Quick Sort

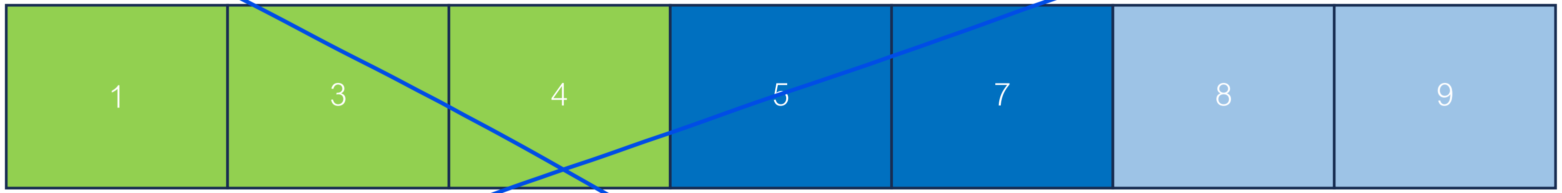


The Steps of Quick Sort



The Steps of Quick Sort

pivot



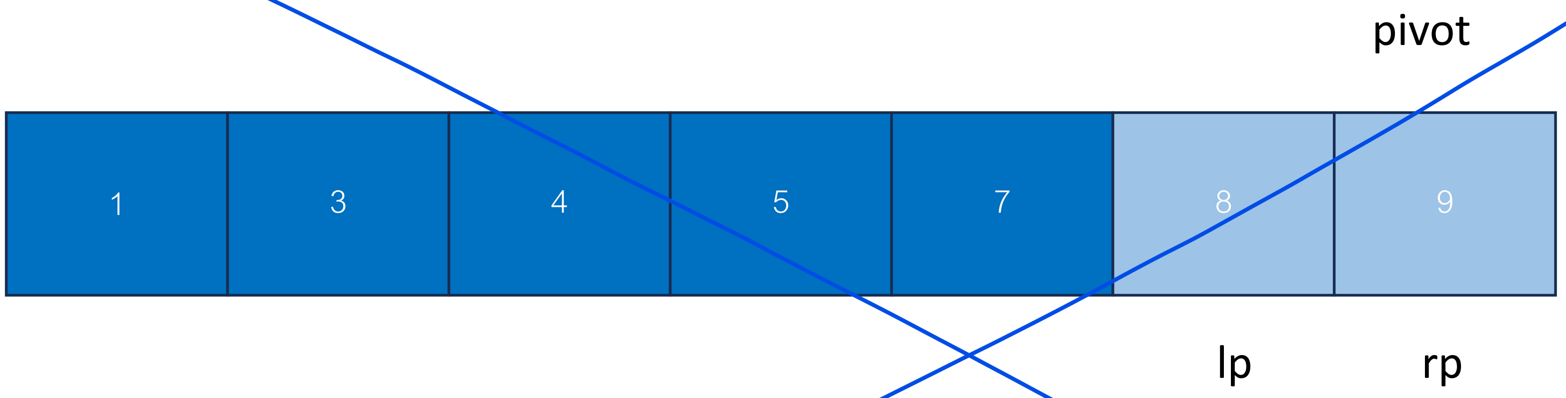
rp

lp

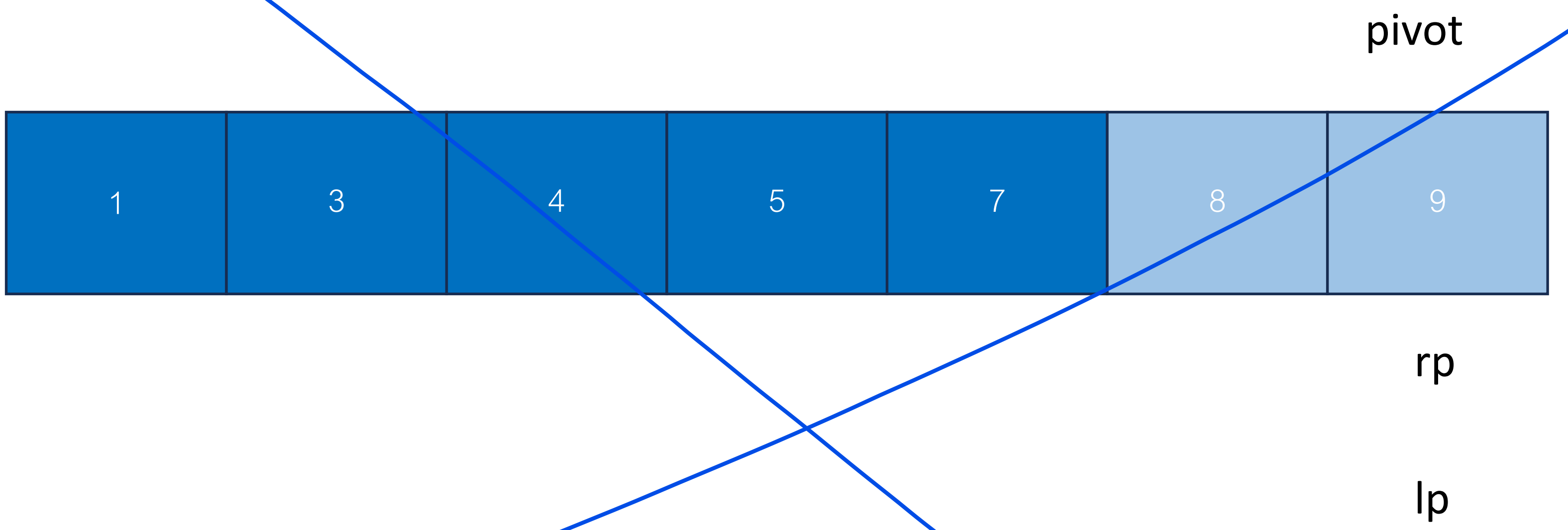
lp = rp, stop

The Steps of Quick Sort

The Steps of Quick Sort



The Steps of Quick Sort



The Steps of Quick Sort

Quick Sort

- Java code???
- [Quick-sort with Hungarian (Küküllőmenti legényes) folk dance (youtube.com)]

- Remarks
  - The are many ways to choose pivot and partitioning.

Definitions

- Recursion
  - Process of solving a problem by reducing it to smaller versions of itself
- Recursive definition
  - Definition in which a problem is expressed in terms of a smaller version of itself
  - Has one or more base cases

Definitions

- Recursive algorithm
  - Algorithm that finds the solution to a given problem by reducing the problem to smaller versions of itself
  - Has one or more base cases
  - Implemented using recursive methods

- Recursive method(general case)
  - Method that calls itself

- Base case
  - Case in recursive definition in which the solution is obtained directly
  - Stops the recursion

Definitions

- General solution
  - Breaks problem into smaller versions of itself
- General case
  - Case in recursive definition in which a smaller version of itself is called
  - Must eventually be reduced to a base case

Definitions

- Recursion in algorithms:
  - Natural approach to some (not all) problems
  - A recursive algorithm uses itself to solve one or more smaller identical problems

- Recursion in Java:
  - Recursive methods implement recursive algorithms
  - A recursive method includes a call to itself

Recursive Methods Must Eventually Terminate

- A base case does not execute a recursive call stops the recursion
- Each successive call to itself must be a "smaller version of itself”
  - an argument that describes a smaller problem
  - a base case is eventually reached

A recursive method must have at least one base, or stopping, case.

Example in Recursion – Factorial

- N! = (N-1)! * N   [for N > 1]
- 1! = 1
- 3!

  = 3 * 2!

  = 3 * 2 * 1!

  = 3 * 2 * 1
- Recursive design:
  - Decomposition: (N-1)!
  - Composition: * N
  - Base case: 1!

Example in Recursion – Factorial Method

```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n - 1) * n; // composition
  else // base case
    fact = 1;

  return fact;
}
```

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = factorial(1) * 2;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = factorial(1) * 2;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 1)
{
  int fact;
  if (n > 1)
    fact = factorial(n - 1) * n;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = factorial(1) * 2;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 1)
{
  int fact;
  if (n > 1)
    fact = factorial(n - 1) * n;
  else
    fact = 1;
  return 1;
}
```

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = 1 * 2;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 1)
{
  int fact;
  if (n > 1)
    fact = factorial(n - 1) * n;
  else
    fact = 1;
  return 1;
}
```

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = 1 * 2;
  else
    fact = 1;
  return 2;
}
```

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = 2 * 3;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = 1 * 2;
  else
    fact = 1;
  return 2;
}
```

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = 2 * 3;
  else
    fact = 1;
  return 6;
}
```

Execution Trace (decomposition)

```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n - 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```

$$\text{factorial(4)} \to \text{factorial(3)}\quad 4$$

Execution Trace (decomposition)

```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n - 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```

```
          factorial(4)
factorial(3)        4
factorial(2)      3
```

Execution Trace (decomposition)

```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n – 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```

factorial(4)

factorial(3)   4

factorial(2)   3

factorial(1)   2

Execution Trace (composition)

```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n - 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```

factorial(4)

*

factorial(3) 4

*

factorial(2) 3

*

factorial(1)->1 2

Execution Trace (composition)

```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n - 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```

factorial(4)

*

factorial(3)   4

*

factorial(2)->2   3

Execution Trace (composition)

```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n - 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```

factorial(4)

*

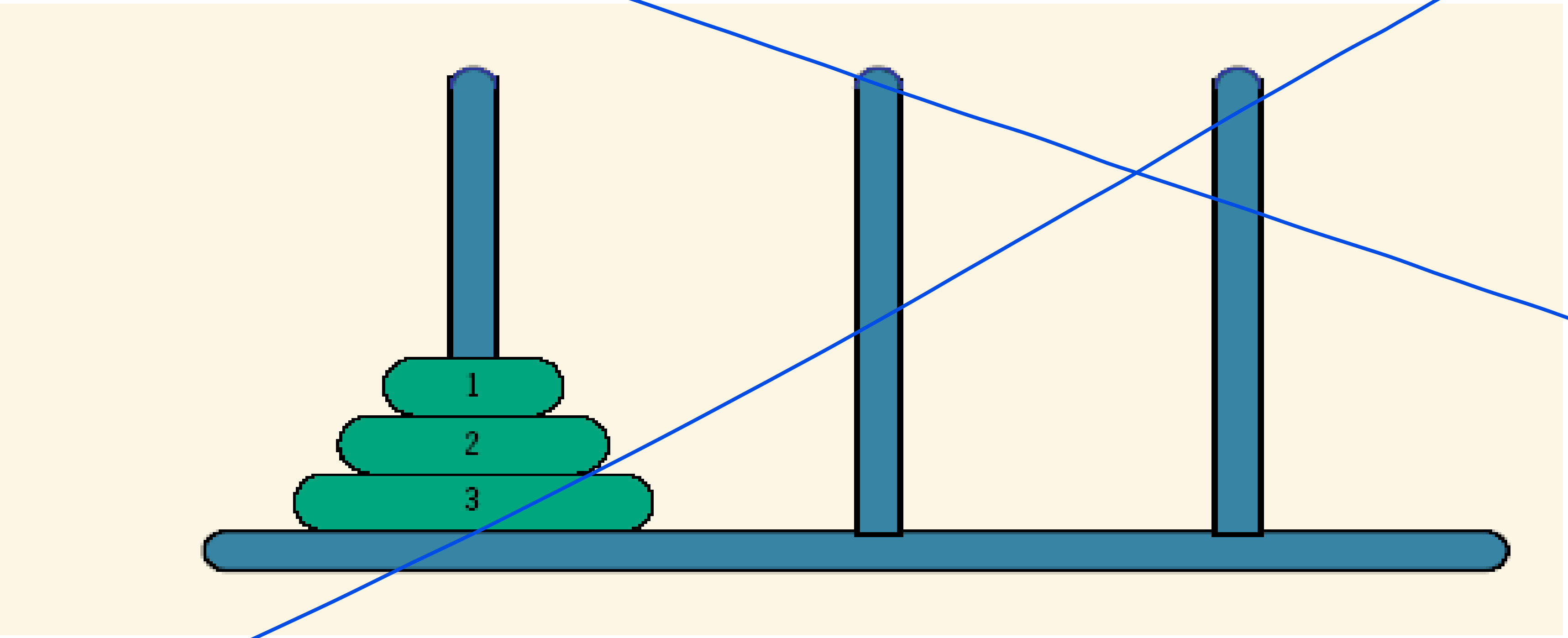
factorial(3)->6   4

Execution Trace (composition)

```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n - 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```

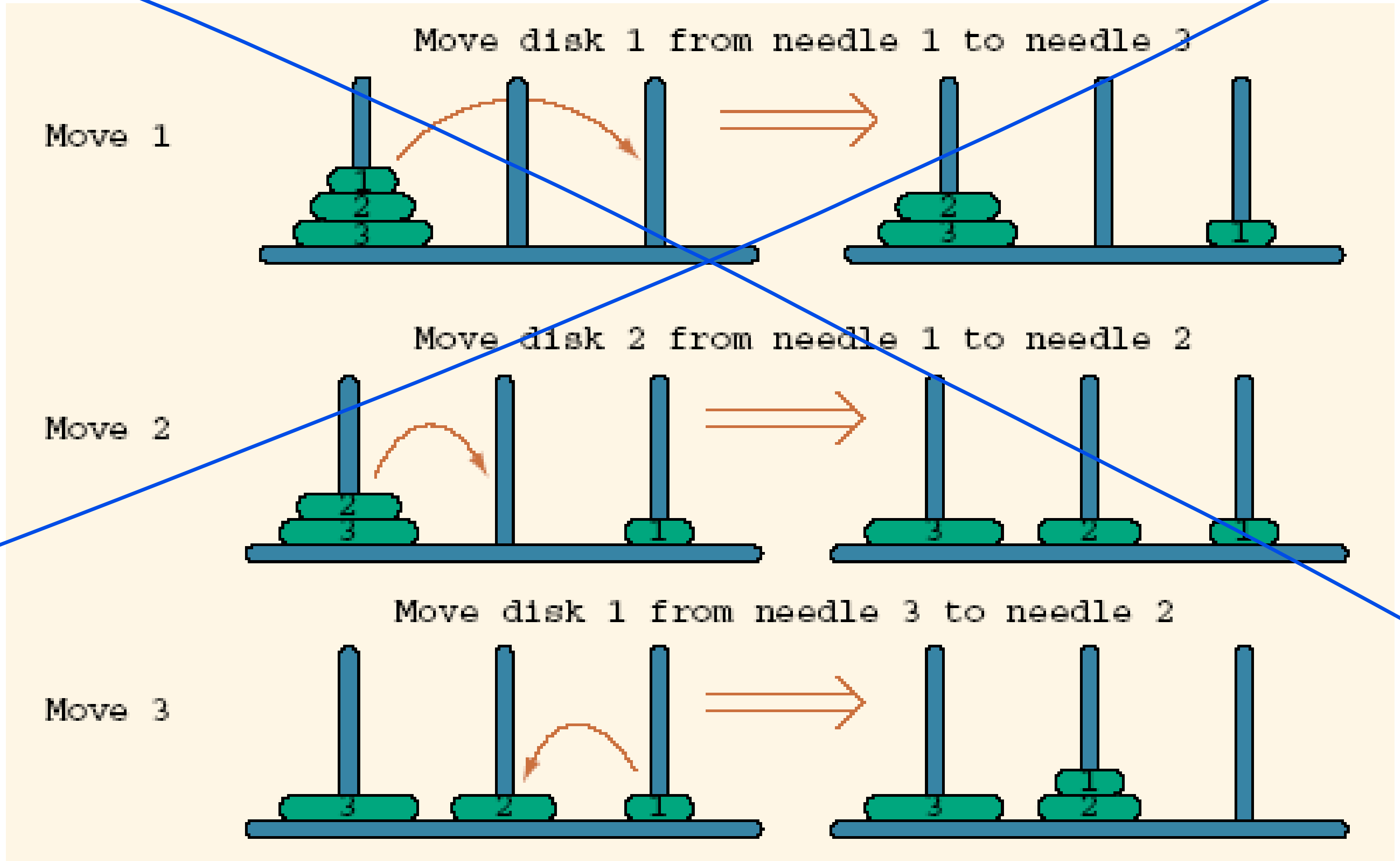
`factorial(4)->24`

Revisit - Towers of Hanoi Problem with Three Disks



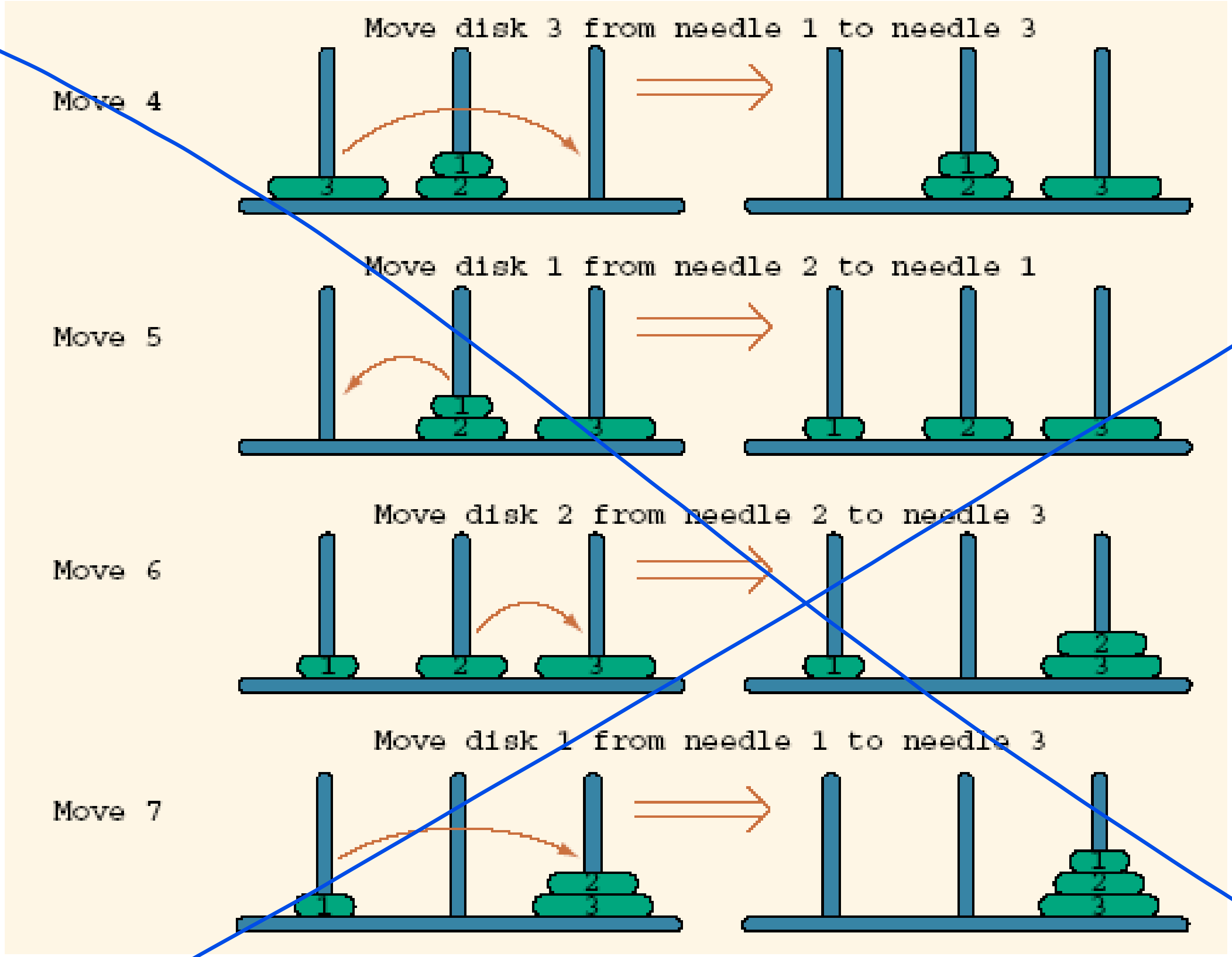
Tower of Hanoi Problem with three disks

Towers of Hanoi: Three Disk Solution



Tower of Hanoi Problem with three disks – Solution 1

Towers of Hanoi: Three Disk Solution (Cont'd)



Tower of Hanoi Problem with three disks – Solution 2

Example in Recursion – Tower of Hanoi

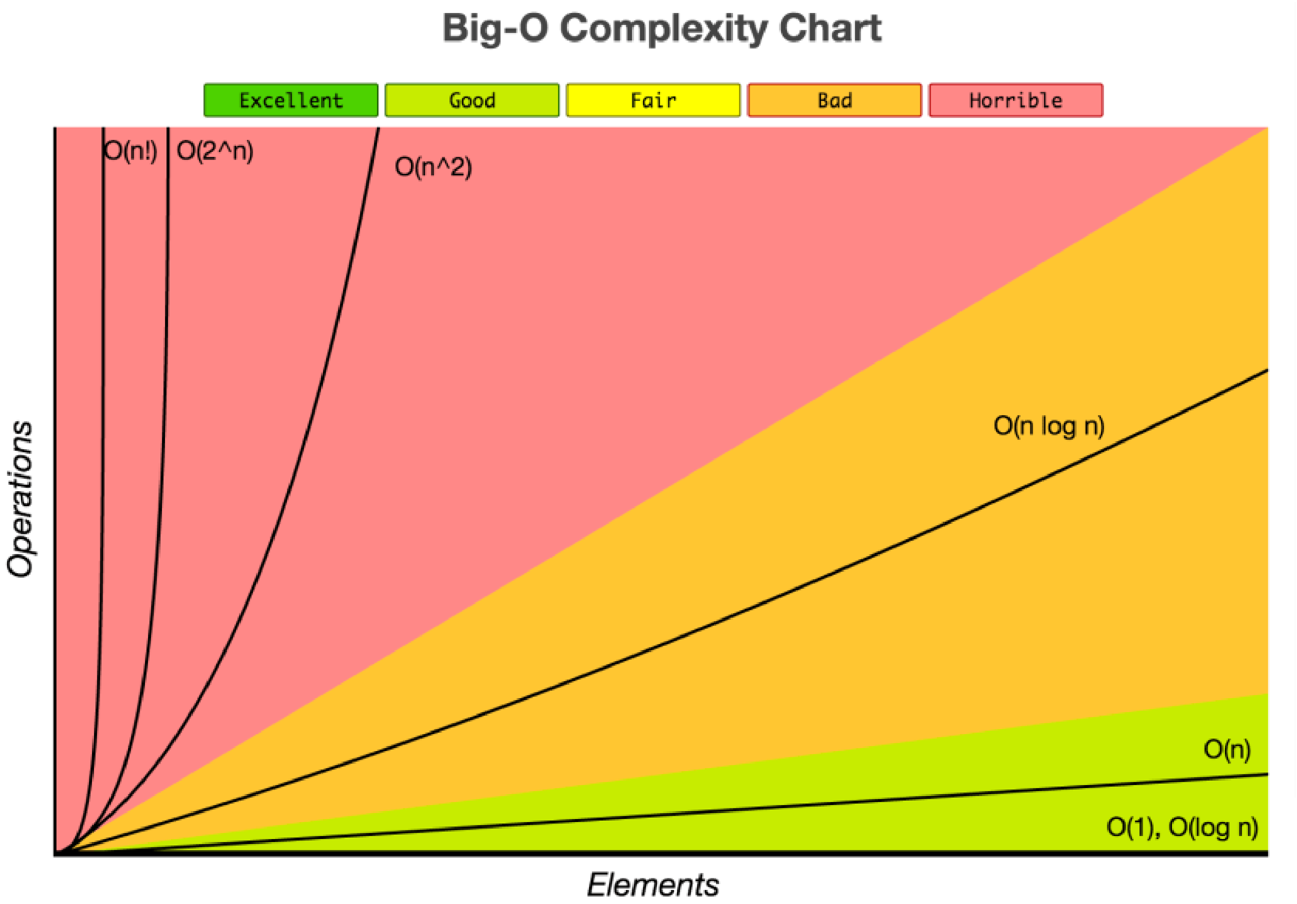
```
public static void moveDisks(int count, int needle1, int needle3, int needle2) {
   if (count > 0)   {
       moveDisks(count - 1, needle1, needle2, needle3);
       System.out.println("Move disk " + count + " from needle "
                        + needle1 + " to needle "
                        + needle3 + ". ");
       moveDisks(count - 1, needle2, needle3, needle1);
    }
}
```

Big O

- To analyze the **cost of the algorithm**
- Constant: O(1)
- Linear time: O(n)
- Logarithmic time: O(n log n)
- Quadratic time: O(n^2)
- Exponential time: 2 ^(n)
- Factorial time: O(n!)

Big O

- When we compute the time complexity T(n) of an algorithm, we rarely get an exact result, just an estimate.
- In computer science we are typically only interested in how fast T(n) is growing as a function of the input size n.



$\boxed{O(1) < O(\log n) < O(n) < O(n\log n)}$

Big O(1) - constant

```
function isOdd (num) {
  if (num % 2 === 0) {
    return true
  }
  else {
    return false
  }
}
```

Big O(n) - linear

```
function findValue (array, value) {
for (int i = 0 ; i < array.length ; i++) {
    if (array[i] === value)
      return true;
  }
}
```

Big O(n log n)-linearithmic

```
function mergeSort (arr) {
  if (arr.length === 1) {
    return arr  }

  const middle =
Math.floor(arr.length / 2)
  const left = arr.slice(0, middle)
  const right = arr.slice(middle)

  return merge(   mergeSort(left),
    mergeSort(right)  )}
```

```
function merge (left, right) {
  int results[] = new int[1000];
  let indexLeft = 0  let indexRight = 0
 while (indexLeft < left.length &&
indexRight < right.length) {    if
(left[indexLeft] < right[indexRight]) {
      result.push(left[indexLeft])
      indexLeft++    } else {
      result.push(right[indexRight])
      indexRight++
    } }
  return
result.concat(left.slice(indexLeft)).concat(
right.slice(indexRight))}
```

Big O (n^2)

```
function isduplicate (array) {
  int i = 0
  while (i < array.length) {
    let j = i + 1
    while (j < array.length) {
      if (array[i] === array[j]) {
        return true
      }     j++    }    i++  }
  return false
}
```

Big O (2^n)

```
function fib (n) {
  if (n <= 1) {    return n  }
  else    return fib(n - 2) + fib(n - 1)
}
```

Big O(n!)

```
function facRuntime (n) {
for (int i = 0 ; i < n ; i++)
    facRuntime (n - 1);
}
```

recursive

Big-O Complexities

- https://www.bigocheatsheet.com/

Solution

```java
public class IterativeCountdown {
    public static void main(String[] args) {
        for (int i = 20; i >= 1; i--) {
            System.out.println(i);
        }
    }
}
```

```java
public class RecursiveCountdown {
    public static void main(String[] args) {
        countdown(20);
    }

    public static void countdown(int n) {
        if (n >= 1) {
            System.out.println(n);
            countdown(n - 1);
        }
    }
}
```

Big O

```java
public class O1Example {
    public static void main(String[] args) {
        int a = 5;
        int b = 10;
        int result = add(a, b);
        System.out.println("The sum is: " + result);
    }

    public static int add(int x, int y) {
        return x + y;
    }
}
```

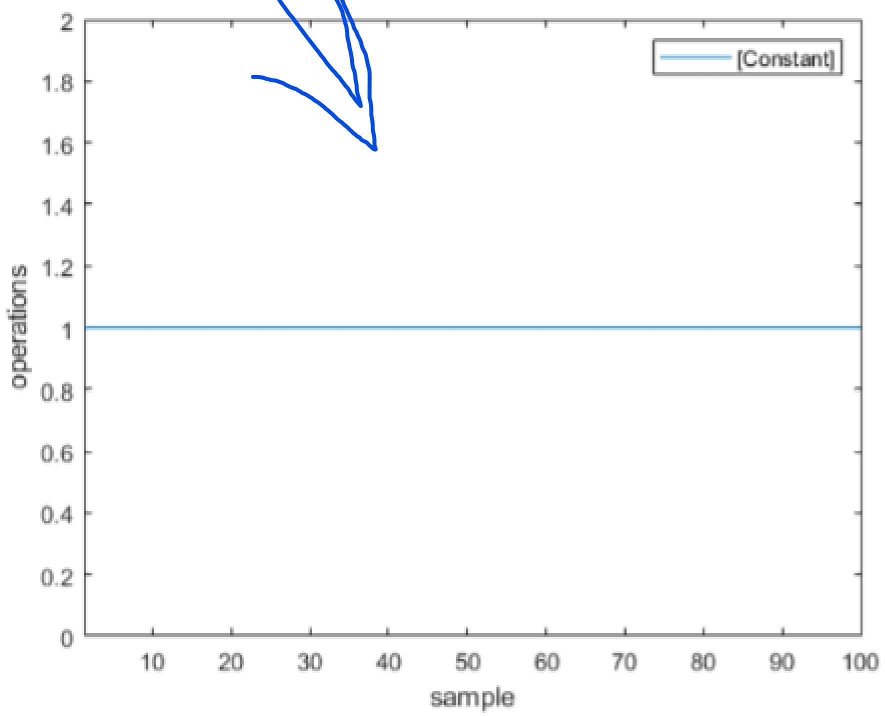


Figure 1.7: Complexity = constant $O(1)$

```java
public class ONExample {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};
        int target = 3;
        boolean found = contains(numbers, target);

        if (found) {
            System.out.println("The target " + target + " was found in the array.");
        } else {
            System.out.println("The target " + target + " was not found in the array.");
        }
    }

    public static boolean contains(int[] arr, int target) {
        for (int num : arr) {
            if (num == target) {
                return true;
            }
        }
        return false;
    }
}
```

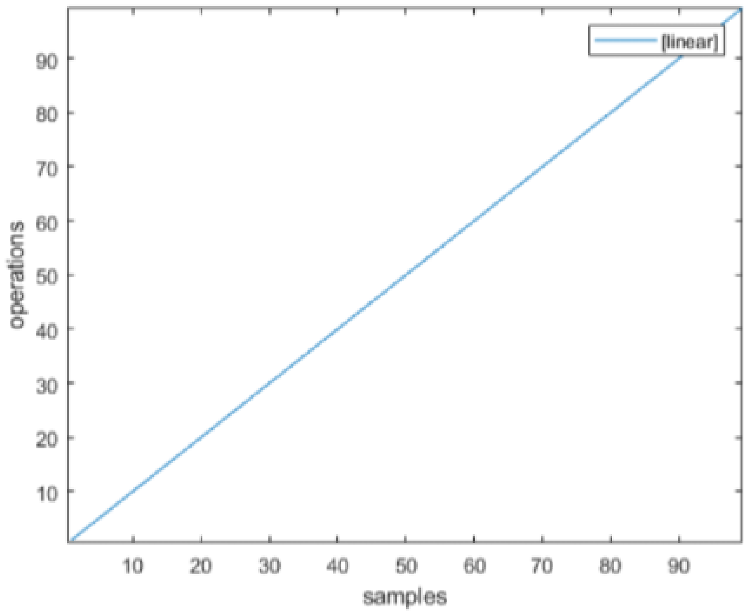


Figure 1.8: Complexity = linear $O(n)$

Big O

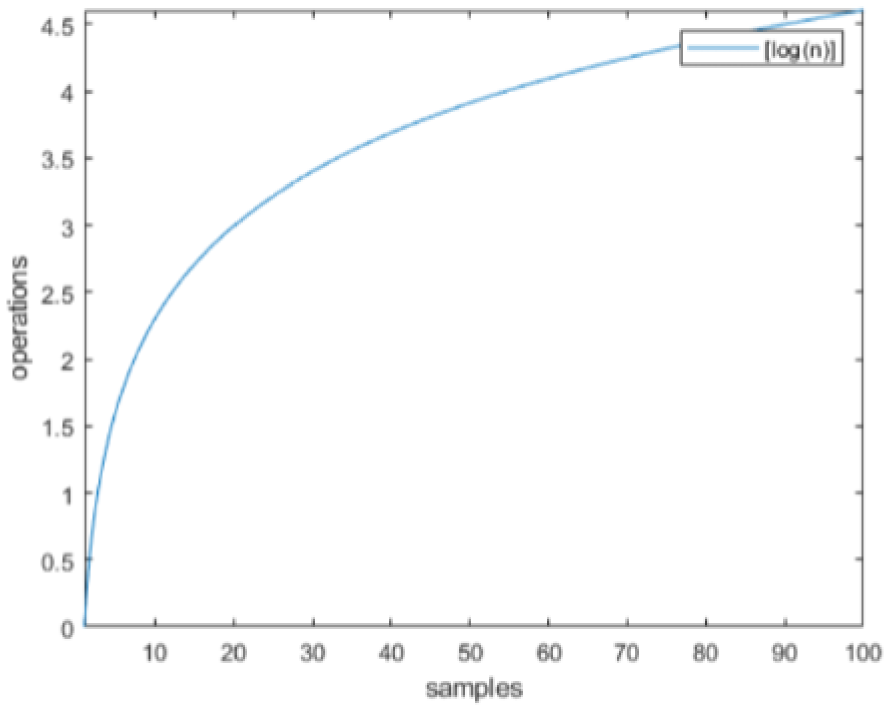


Figure 1.9: Complexity = $O(logn)$

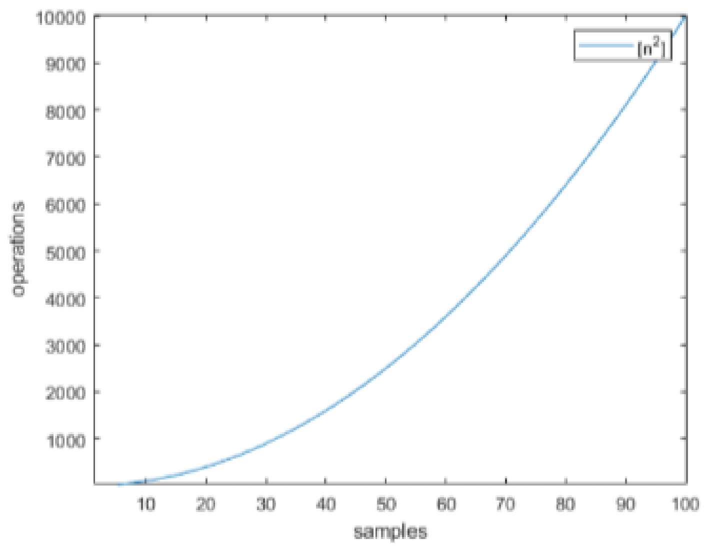


Figure 1.10: Complexity = $O(n^2)$

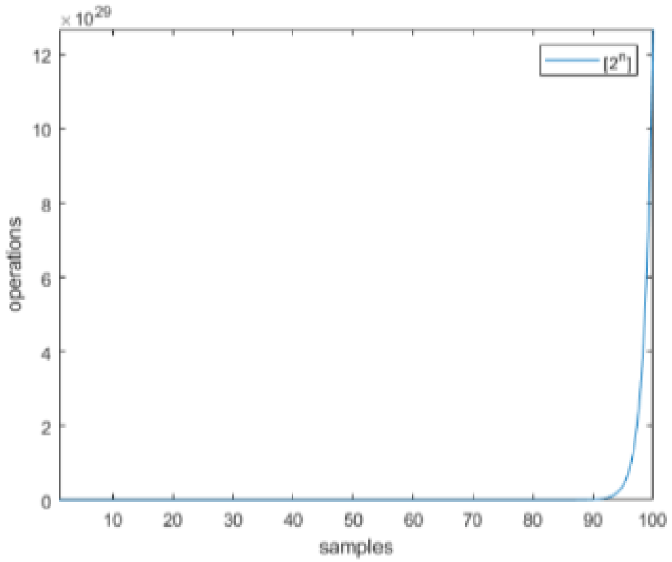


Figure 1.11: Complexity = $O(2^n)$

```java
 * @author Pree Thiengburanathum
 * pree.t@cmu.ac.th
 * BigODemo.java
 */
public class BigODemo {
    public static void main (String args[]) {
    int n = 10;
        // O(n)
        System.out.println("O(n)");
        for (int i = 0; i < n; i++) {
            System.out.println("counting " + i +"//");
        }

        // O(nlogn)
        System.out.println("O(nlogn)");
        for (int i = 1; i <= n; i++){
            for(int j = 1; j < n; j = j * 2) {
                System.out.println("counting" + i + " and " + j+"//");
            }
        }
        // O(n^2)
        System.out.println("O(n^2)");
        for (int i = 1; i <= n; i++) {
            for(int j = 1; j <= n; j++) {
                System.out.println("counting " + i + " and " +
                    j+"//");
            }
        }

        // O(n^2)
        System.out.println("O(2^n)");
        for (int i = 1; i <= Math.pow(2, n); i++){
            System.out.println("counting " + i+"//");
        }
    }// end main
```

Java String

- In Java, a String is a class that represents a sequence of characters.
- one of the most commonly used classes in the Java programming language and is part of the java.lang package
- No need to import it explicitly in your code.

String examples

```java
String str1 = "Hello";
String str2 = "World";

// Concatenation
String result = str1 + ", " + str2;

// Length of the string
int length = result.length();

// Substring extraction
String substring = result.substring(0, 5); // "Hello"

// Searching for a character or substring
boolean contains = result.contains("World");

// Replacing a substring
String replaced = result.replace("World", "Java");
```

Iterate to char in String

```java
String text = "Hello, World!";
for (int i = 0; i < text.length(); i++) {
    char character = text.charAt(i);
    // Now, 'character' holds the character at position 'i'
    System.out.println(character);
}
```

Palindrome

- A palindrome - a word, phrase, number, or other sequence of characters that reads the same forward and backward, ignoring spaces, punctuation, and capitalization (in the case of letters).
- In essence, a palindrome remains unchanged when its characters are reversed.

Examples

- "racecar"
- "level"
- "deified"
- "A man, a plan, a canal, Panama!"
- "12321"

```
c static boolean isPalindrome(String str) {
tr = str.replaceAll("[\\s+.,!?:;]", "").toLowerCase(); // Removes special char
nt left = 0;
nt right = str.length() - 1;

hile (left < right) {
    if (str.charAt(left) != str.charAt(right)) {
        return false; // Not a palindrome
    }
    left++;
    right--;

eturn true; // Is a palindrome
```

Recursive practice 1

- Class exercise : recursion
  - Printing problem
  - Write a simple Java program that print number from 20 to 1 for both iterative and recursive version.

- Output
- 20
- 19
- 18
- ..
- ..
- 1