TOP JAVA CHALLENGES: CRACKING THE CODING INTERVIEW

Based on 30 real questions

Moises Gamio

Top Java Challenges: Cracking the Coding Interview

by Moises Gamio

If you find any error in the text or the code or if you have any suggestions, please let me know by emailing at support@codersite.dev. Once you have read and used this book, please leave a review on the site that you purchased it. By doing so, you can help me improve the next editions of this book. Thanks, and I hope you enjoy using the text in your job interview!

Cover designer: Maria Elena Gamio

ISBN: 9798650252368

1. Reverse a Text

Given a string of characters, reverse the order of the characters in an efficient manner.

Solution

We choose an ***array*** – holds values of a single type - as our data structure because the algorithm receives a small amount of data, which is predictable and is read it randomly (its numerical index accesses each element).

Firstly, convert the text to be reversed, to a character array. Then, calculate the length of the string.

Secondly, swap the position of array elements using a loop. Don't use additional memory, which means avoiding unnecessary objects or variables (space complexity). Swapping does it in-place by transposing values using a temporary variable. Then, swap the first element with the last, the second element with the penultimate, and so on. Moreover, we only need to iterate until half of the array.

Finally, it returns the new character array as a String. Listing 1.1 shows the algorithm.

Listing 1.1 – Reverse a Text

```
public class StringUtils {
  public static String reverse(String text) {
    char[] chars =text.toCharArray();
    final int arrayLength =chars.length;
    char temp;
    for (int idx =0; idx < arrayLength/2; idx++) {
      temp =chars[idx];
      chars[idx] =chars[arrayLength - 1 - idx];
      chars[arrayLength - 1 - idx] =temp;
    }
    return String.valueOf(chars);
  }
}
```

Example:

When idx = 0:
chars = {a, b, c, 2, 1, 3, 2}

Under the *loop for* sentence, we need to add the following conditional sentence:

```
   if ((idx+1) % 2 != 0) {
     …
```

```
@Test
public void reverseOdssText() {
  assertEquals("ub32tca192", StringUtils.reverse ("2b12cta39u"));
}
```

What is the performance of the algorithm?

Start to analyze the most important sentences:

`char[] chars =text.toCharArray();` -> runs in only 1 execution: O(1)
`final int arrayLength =chars.length;` -> runs in only 1 execution: O(1)
`for (int idx=0; idx<arrayLength/2; idx++){` -> runs in O(N)
`return String.valueOf(chars);` -> runs in only 1 execution: O(1)

Total time: O(1) + O(1) + O(N) + O(1) = O(N)

In this scenario, A constant time O(1) is insignificance compared with a linear time O(N)

See appendix A find out why Big O Notation ignores constants

For instance:

When `chars.length` is 100, then,
Total time:
O(1)+O(1)+O(100/2)+O(1) = 1+1+50+1 = 53 ≈ 50 → (N/2) → N

When `chars.length` is 100000, then,
Total time:
O(1)+O(1)+O(100000/2)+O(1) = 1+1+50000+1 = 50003 ≈ 50000 → (N/2) → N

Therefore, we can say that our Reverse Text algorithm runs in O(N) time.

15. Given an integer N, returns its Factorial

The **factorial** is the **product** of all positive integers less than or equal to the non-negative integer. In real life, the factorial is the number of ways you can arrange *n* objects.

Solution

We make use of recursion. A recursive function is one that is defined in terms of itself. We always include a base case to finish the recursive calls.

Each time a function calls itself, its arguments are stored on the stack before the new arguments take effect. Each call creates ***new local variables***. Thus, each call has its copy of arguments and local variables. That is one reason that we don't need to use recursion in the Production environment when we pass a big integer, they can overflow the stack and crash any application. Time complexity is O(N). Listing 21.1 shows the algorithm.

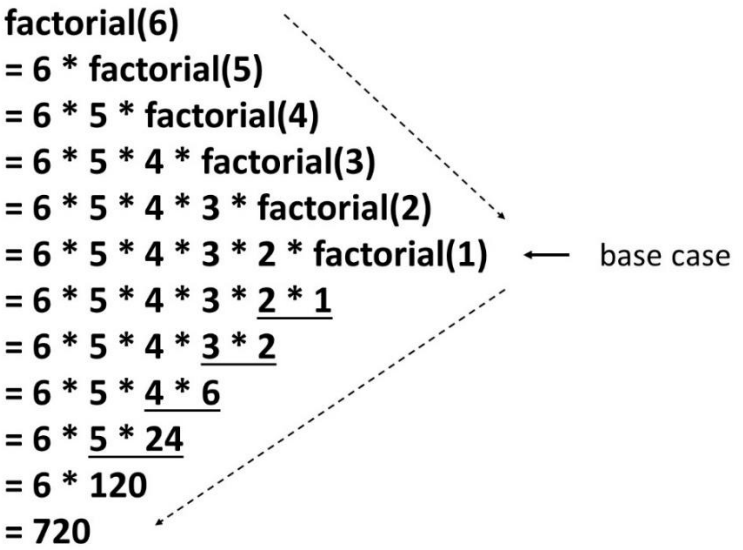

Figure 21.1 N Factorial

16. Bubble Sort

Bubble Sort uses a not sorted array, which contains at least two adjacent elements that are out of order. The algorithm repeatedly passes through the array, swapping elements that are out of order, and continues until it cannot find any more swaps.

Solution

The algorithm uses a Boolean variable to keep track of whether it has found a swap in its most recent pass through the array; as long as the variable is True, the algorithm loops through the Array, looking for adjacent pairs of elements that are out of order and swap them. The time complexity, in the worst case it requires $O(n^2)$ comparisons. Listing 14.1 shows the algorithm.

Listing 14.1 – Bubble Sort

```
public class Sorting {

  public int[] bubbleSort(int[] numbers) {
    if (numbers == null)
      return numbers;

    boolean numbersSwapped;
    do {
      numbersSwapped = false;
      for (int i = 0; i < numbers.length - 1; i++) {
        if (numbers[i] > numbers[i + 1]) {
          int aux = numbers[i + 1];
          numbers[i + 1] = numbers[i];
          numbers[i] = aux;
          numbersSwapped = true;
        }
      }
    } while (numbersSwapped);

    return numbers;
  }
}
```

Example:
First pass-through:
{**6, 4**, 9, 5} -> {**4, 6**, 9, 5} swap because of 6 > 4
{4, **6, 9**, 5} -> {4, **6, 9**, 5}
{4, 6, **9, 5**} -> {4, 6, **5, 9**} swap because of 9 > 5

19. Binary Search

Given a sorted array of N elements, write a function to search a given element X in the Array.

Solution

Search the sorted array by repeatedly dividing the search interval in half. If the element X is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Repeatedly check until the value is found or the interval is empty. Figure 17.1 shows the iteration when we search for 21. Time complexity is O (log n). If we pass an array of 4 billion elements, it takes at most 32 comparisons. Listing 17.1 shows the algorithm.

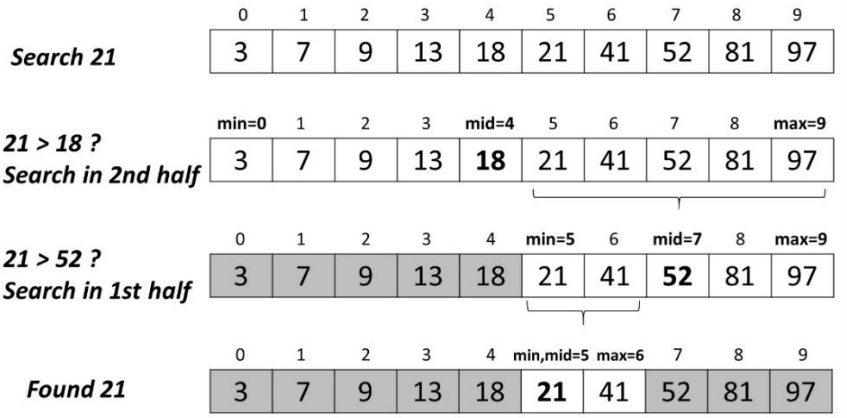

Figure 17.1 Binary Search example

Listing 17.1 Binary Search

```
public class BinarySearch {

  public static <T extends Comparable<T>>
    boolean search(T target, T[] array) {
```

20. Rotate the matrix by 90 degrees

Given a square matrix, turn it by 90 degrees in a clockwise direction.

Solution

We build two for loops, an outer one deals with one layer of the matrix per iteration, and an inner one deals with the rotation of the elements of the layers. We rotate the elements in n/2 cycles. In every square cycle, we swap the elements with the corresponding cell in the matrix by using a temporary variable. Listing 18.1 shows the algorithm.

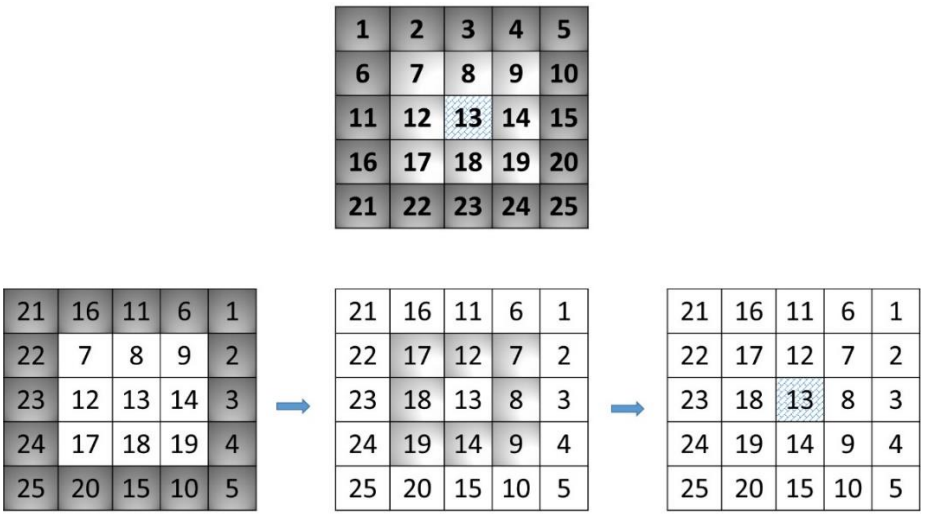

Figure 18.1 Rotate matrix by 90 degrees

Listing 18.1 – Rotate the Matrix by 90 degrees.

```
public class MatrixUtils {
  public static void rotate(int[][] matrix) {
    int n = matrix.length;
    if (n <=1)
      return;


    /* layers */
    for (int i = 0; i < n / 2; i++) {
      /* elements */
      for (int j = i; j < n - i - 1; j++) {
        //Swap elements in the clockwise direction
```

```
      }
    }
  }
}
```

Tests

```
public class MatrixUtilsTest {

  @Test
  public void rotate4x4() {
    int[][] matrix = new int[][]{
        {9, 10, 11, 12},
        {16, 17, 18, 19},
        {23, 24, 25, 26},
        {30, 31, 32, 33}};
    MatrixUtils.rotate(matrix);
    assertArrayEquals(new int[]{30, 23, 16, 9}, matrix[0]);
    assertArrayEquals(new int[]{33, 26, 19, 12}, matrix[3]);
  }

  @Test
  public void rotate5x5() {
    int[][] matrix = new int[][]{
      {1, 2, 3, 4, 5},
      {6, 7, 8, 9, 10},
      {11, 12, 13, 14, 15},
      {16, 17, 18, 19, 20},
      {21, 22, 23, 24, 25}};
    MatrixUtils.rotate(matrix);
    assertArrayEquals(new int[]{21, 16, 11, 6, 1}, matrix[0]);
    assertArrayEquals(new int[]{22, 17, 12, 7, 2}, matrix[1]);
  }
}
```

22. Queue via Stacks

Build a queue data structure using only two internal stacks.

Stacks are based on the LIFO principle, i.e., the element *inserted* at *last* is the *first* element to come *out* of the list. **Queues** are based on the FIFO principle, i.e., the element *inserted* at *first* is the *first* element to come *out* of the list.

Solution

Stacks and Queues are abstract at their definition. That means, for example, in the case of Queues, that we can implement its behavior using two stacks.

Then we create two stacks of `java.util.Stack`, inbox, and outbox. The *add* method pushes new elements onto the inbox. And the *peek* method will do the following:

- If the outbox is empty, refill it by popping each element from the inbox and pushing it onto the outbox.
- Pop and return the top element from the outbox.

Listing 19.1 shows the algorithm.

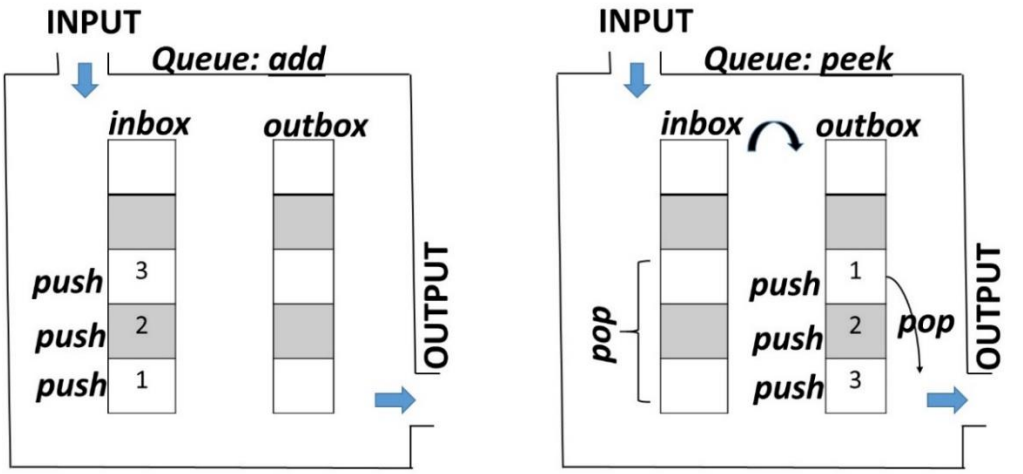

Figure 19.1 Queue via Stacks

26. Binary Search Tree

Tree structures are non-linear data structures. They allow us to implement algorithms much faster than when using linear data structures. A binary tree can have at the most two children: a left node and a right node. Every node contains two elements: a key used to identify the data stored by the node, and a value that is the data collected in the node.

The most common type of binary tree is the Binary Search Tree, which has two main characteristics:

- The value of the left node must be lesser than the value of its parent.
- The value of the right node must be greater than or equal to the value of its parent.

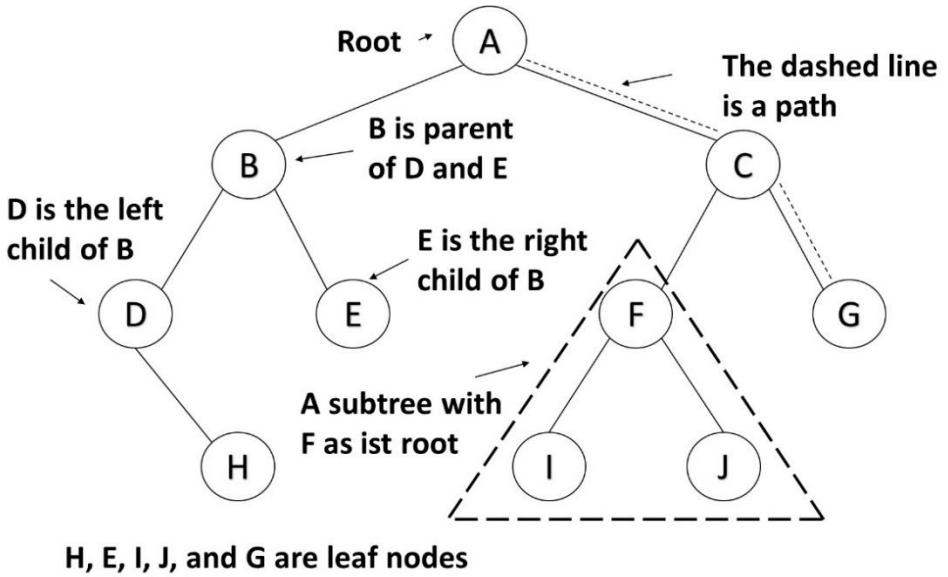

Figure 26.1 Binary Search Tree - terminology

You can search a tree quickly, as you can an ordered array, and you can also insert and delete items quickly, as you can with a linked list.

The Global Trade Item Number (GTIN) can be used by a company to identify all of its trade items uniquely. The GTIN identifies types of products that are produced by different manufacturers.

Use Case: A Webshop wants to retrieve information about GTINs efficiently by using a binary search algorithm.

27. Depth-First Search (DFS)

Implement the depth-first search algorithm to traverse a graph data structure.

Solution

A **Graph** is a non-linear **data structure** consisting of nodes (vertices) and edges. Its shape depends on the physical or abstract problem we are trying to solve. For example, if nodes represent cities, the routes which connect cities may be defined by *no-directed* edges. But if nodes represent tasks to complete a project, then their edges must be *directed* to indicate which task must be completed before another.

Terminology

A Graph can models ***Hyperloop*** transport to be installed in Germany.

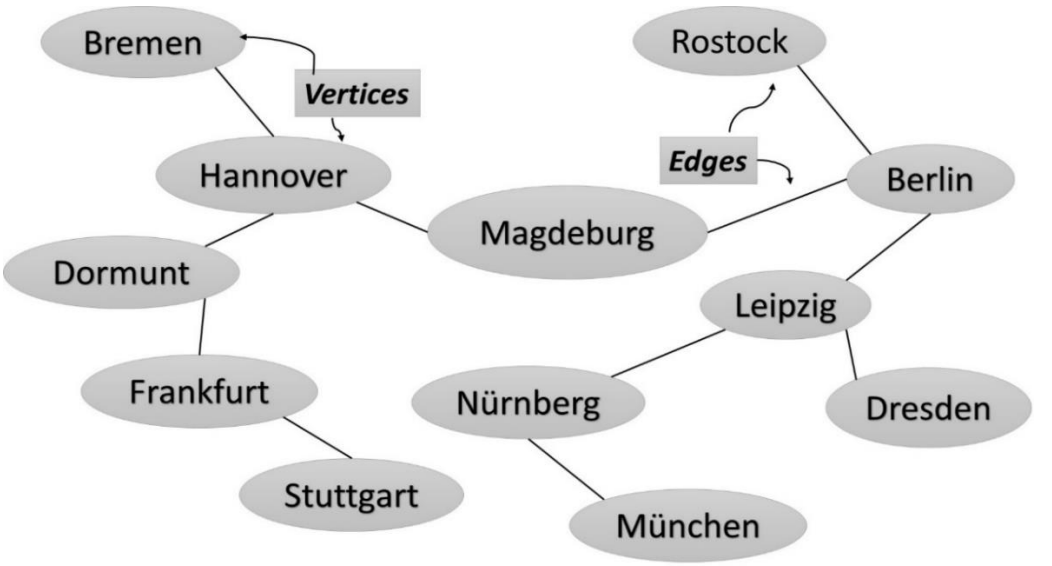

Figure 27.1 Graph - terminology

A Graph shows only the relationships between the *vertices* and the *edges*. The most important here is which edges are connected to which vertex. We can also say that Graph models connections between objects.

29. Optimize online purchases

Given a budget *B* and a 2-D array, which includes [product-id][price][value], write an algorithm to optimize a basket with the most valuable products whose costs are less or equal than *B*.

Solution

Imagine that we have a budget of 4 US$ and we want to buy the most valuable snacks from the table 29.1.

Id	Name	Price US$	Amount gr.	Amount x US$
1	Snack Funny Pencil	0,48	36	75g
2	Snackin Chicken Protein	0,89	10	11g
3	Snacks Waffle Pretzels	0,98	226	230g
4	Snacks Tahoe Pretzels	0,98	226	230g
5	Tako Chips Snack	1,29	60	47g
6	Shrimp Snacks	1,29	71	55g
7	Rasa Jagung Bakar	1,35	50	37g
8	Snack Balls	1,65	12	7g
9	Sabor Cheese Snacks	1,69	20	12g
10	Osem Bissli Falafel	4,86	70	14g

Table 29.1 List of snacks

But who decides if a product is more valuable than another one? Well, this depends on every business. It could be an estimation based on quantitative or qualitative analysis. For our solution, we choose a quantitative approach based on which product gives us *more grams per every dollar invested.*

To implement our algorithm, we use the Red-Green Refactor technique, which is the basis of test-drive-development (TDD). In every assumption, we will write a test and see if it fails. Then, we write the code that implements only that test and sees if it succeeded, then we can refactor the code to make it better. Then we continue with another assumption and repeat the previous steps until the algorithm is successfully implemented for all tests.

To generalize the concept of “the most valuable product,” we assign a value to every

Tests

```
public class BasketOptimizedTest {

  BasketOptimized basketOptimized;

  @Before
  public void setup() {
    basketOptimized = new BasketOptimized();
  }

  @Test
  public void given_productsOrderedByValue_return_mostValueables() {

    double[][] myProducts = new double[][]{
        {1, 0.98, 230},
        {2, 0.98, 230},
        {3, 0.48, 75},
        {4, 1.29, 55},
        {5, 1.29, 47},
        {6, 4.86, 14},
        {7, 1.69, 12}
    };
    double[][] mostValueableProducts
      = basketOptimized.fill(myProducts, 4);
    assertEquals(590d,
        Arrays.stream(mostValueableProducts).mapToDouble(arr ->
        arr[2]).sum(), 0);
  }

  @Test
  public void given_productsNotOrderedByValue_return_mostValuables() {

    double[][] myProducts = new double[][]{
        {1, 0.98, 230},
        {2, 1.29, 47},
        {3, 1.69, 12},
        {4, 1.29, 55},
        {5, 0.98, 230},
        {6, 4.86, 14},
        {7, 0.48, 75}
    };

    double[][] mostValueableProducts
      = basketOptimized.fill(myProducts, 4);
    assertEquals(590d,
        Arrays.stream(mostValueableProducts).mapToDouble(arr ->
        arr[2]).sum(), 0);
  }
```

30. Tic tac toe

Write a tic-tac-toe program where the size of the board should be configurable between 3x3 and 9x9. It should be for three players instead of two, and its symbols must be configurable. One of the players is an AI. All three players play all together against each other. The play starts in random. The input of the AI is automatic. The input from the console must be provided in format X, Y. After every move, the new status of the Board is printed. The winner is who completes a whole row, column, or diagonal.

General Rules: https://en.wikipedia.org/wiki/Tic-tac-toe

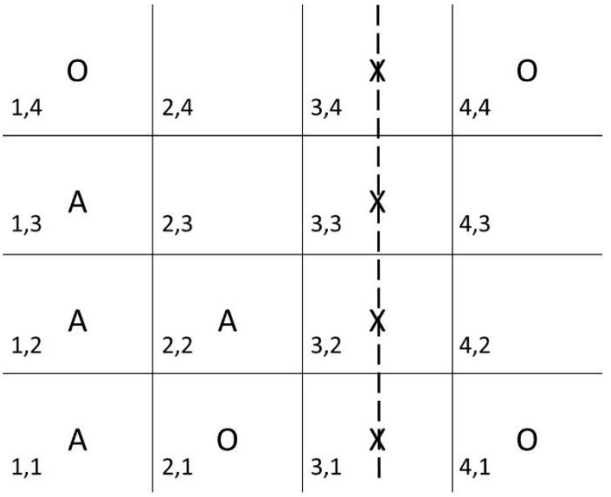

Figure 30.1 Tic-tac-toe game

Solution

What we learn from Object-Oriented Design and SOLID principles is that we need to delegate responsibilities to different components. For this game, we identify the following classes:

Board – set Size, get Winner, draw?
Player – (Human, IA)
Utils – to load configuration files.
App – the main class that assembly and control our different components.

Test case #1: Define the size of the Board

Based on the size of the Board, we need to initialize a bi-dimensional array to store the symbols after every move. Listing 30.1 shows one assumption about the setSize method.

Listing 30.1 – Board Class, setSize Test case

```
public class BoardTest {

  private Board board;

  @Before
  public void setUp() {
    board = new Board();
  }

  @Test
  public void whenSizeThenSetupBoardSize() throws Exception {
    board.setSize(10);
    assertEquals(10, board.getBoard().length);
  }
}
```

Listing 30.2 shows an initial implementation of Board Class and the setSize method.

Listing 30.2 – Board Class, setSize method

```
public class Board {

  private final static String EMPTY_ = " ";
  private String[][] board;

  public void setSize(int size) {
    this.board = new String[size][size];
    for (int x = 0; x < size; x++) {
      for (int y = 0; y < size; y++) {
        board[x][y] = EMPTY_;
      }
    }
  }
  public String[][] getBoard() {
    return board;
  }
}
```

TDD allows us to design, build, and test the smallest methods first and assemble them later. And the most important is that we can refactor it without breaking the rest of the test cases.

Big O Notation

Big O Notation is a mathematical function, which helps us to analyze how complex an algorithm is in time and space. It matters when we build an application for millions of users. We implement different algorithms to solve one problem and measure how efficient is one respect to the other ones.

Time and Space complexity

Time complexity is related to how many steps take the algorithm.
Space complexity is related to how efficient the algorithm is using the memory and disk.
Both terms depend on the input size, the number of items in the input. We can analyze the complexity based on three cases:

- Best case or Big Omega **Ω(n)**: Usually, the algorithm executes in one step independently of the input size.
- Average case or Big Theta **Θ(n)**: When the input size is random
- Worst-case or Big O Notation **O(n)**: Gives us an upper bound on the runtime for any input. It gives us a kind of guarantee that the algorithm will never take any longer with new input size.

Order of growth

The order of growth is related to how the runtime of an algorithm increases when the size of the input increases without limit and tells us how efficient the algorithm is. We can compare the relative performance of alternative algorithms.

Common order-of-growth classifications:

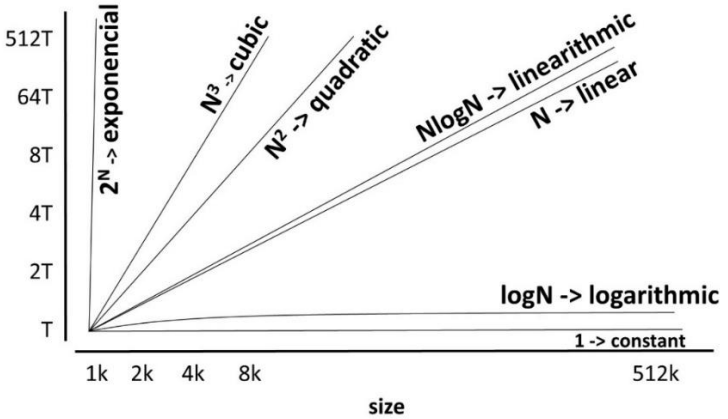

Figure A.1 Big O Notation - order of growth

Big O Notations examples:

O(1) - Constant

It does not matter if the input contains 1000 or 1 million items. The code always executes in one step.

```
public class BigONotation {

  public void constant(List<String> list, String item) {
    list.add(item);
  }
}
```

```
@Test
public void test_constantTime() {
  List<String> list =
     new ArrayList<>(Arrays.asList("one", "two", "three"));
  bigONotation.constant(list, "four");
}
```

O(N) – linear

We say our algorithm runs in O(N) time if the number of steps depends on the number of items included in the input

```
public int sum(int[] numbers) {
  int sum =0;
  for (int i =0; i<numbers.length; i++) {
    sum+=numbers[i];
  }
  return sum;
}
```

```
@Test
public void test_linearTime() {
  final int[] numbers = {1, 2, 4, 6, 1, 6};
  assertTrue(bigONotation.sum(numbers) == 20);
}
```