TOP

Java Challenges

Cracking the Coding Interview

BASED ON REAL QUESTIONS

Moises Gamio

TOP JAVA CHALLENGES

CRACKING THE CODING INTERVIEW

Based on real questions

Moises Gamio

Founder, codersite.dev

Top Java Challenges

Cracking the Coding Interview

by Moises Gamio

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! If you find any error in the text, code, or suggestions, please let me know by emailing me at codersitedev@gmail.com.

Cover designer: Maria Elena Gamio

ISBN: 9798650252368

For Lorena

Contenido

PREFACE **vii**

Introduction **1**

Algorithms and Data structures *1*

Abstract Data Types *1*

How to develop a Usable Algorithm *1*

Clean Code *1*

Test-Driven Java Development *2*

Arrays and Strings **3**

1.0 Fundamentals *3*

1.1 Reverse a Text *5*

1.2 Validate If a String Has All Unique Characters *8*

1.3 Validate If a String Is a Palindrome *9*

1.4 Compare Application Version Numbers *10*

1.5 Remove Duplicates from a List *12*

1.6 Rotate a Matrix by 90 Degrees *13*

1.7 Items in Containers *15*

1.8 Shopping Options *18*

Linked Lists **21**

2.0 Fundamentals *21*

2.1 Implement a Linked List *22*

2.2 Reverse a Linked List *24*

Math and Logic Puzzles **26**

3.0 Fundamentals *26*

3.1 Sum of All Even Numbers from 1 to a Given Integer *27*

3.2 Validate a Leap Year *28*

3.3 Given an Integer N, Returns the Smallest Number of Digits *29*

3.4 Fizz-Buzz *31*

3.5 Verify If a Number Is a Valid Power of a Base *32*

3.6 Validate If a Number Is Prime or Not *33*

3.7 Distance Between Two Points *34*

3.8 Write an Immutable Class to convert Currencies *36*

3.9 Number of Products of Two Consecutive Integers *38*

3.10 Assemble Parts in Minimum Time *39*

Recursion **41**

4.0 Fundamentals *41*

4.1 Calculate Factorial of a Given Integer N *41*

4.2 Calculate Fibonacci Series *42*

Sorting and Searching **44**

5.0 Fundamentals 44
5.1 Bubble Sort 44
5.2 Insertion Sort 47
5.3 Quick Sort 48
5.4 Binary Search 50

Stacks and Queues **53**
6.0 Fundamentals 53
6.1 Delimiter Matching 54
6.2 Queue via Stacks 56
6.3 Reverse a String using a Stack 57

Hash Table **59**
7.0 Fundamentals 59
7.1 Design a Hash Table 60
7.2 Find the Most Frequent Elements in an Array 63
7.3 Nuts and Bolts 64

Trees **66**
8.0 Fundamentals 66
8.1 Binary Search Tree 67
8.2 Tree Traversal 70

Graphs **74**
9.0 Fundamentals 74
9.1 Depth-First Search (DFS) 75
9.2 Breadth-First Search (BFS) 81

Coding Challenges **86**
10.0 Fundamentals 86
10.1 Optimize Online Purchases 86
10.2 Tic Tac Toe 94

Big O Notation **114**
Time and Space Complexity 114
Order of Growth of Common Algorithms 114
How to Find the Time Complexity of an Algorithm 117
Why Big O Notation ignores Constants? 118

PREFACE

I am a software engineer who faced real interviews as a candidate for startups and big companies. Throughout the years, I have sourced factual questions that have been tried, tested, and commented on step by step and are now part of this book! I hope you find them practical and valuable in your career search.

Yes, your CV matters. The job market is tough, and big firms and tech startups receive many applicants every day, so the need to filter the best one becomes a daunting process. The interview by phone and the code challenge to be solved at home will lead you to the last and most decisive stage of the recruiting process, where the interviewer evaluates your problem-solving skills and knowledge of efficient data structures and algorithms.

Usually, the interviewer poses a complex problem to solve in a limited time. You need to provide rationales about design choices made in your proposed solution in terms of resource and time efficiency.

Therefore, knowing the most appropriate data structures and algorithms to solve common problems frequently used in the recruiting process can be decisive. Whether your knowledge of efficient data structures and algorithms is scarce or buried in your memory since you learned about them for the first time, this book can help you. It includes the most common questions, and their respective solutions, that you can find in a real interview. Recall, the more prepared you are, the more points you accumulate objectively concerning other candidates.

This book includes typical questions based on real interviews that you must know and practice. All solved and explained in detail.

Typical questions include String manipulation, arrays, variables swapping, linked list, refactoring, recursion, sorting, searching, stacks, queues, trees, graphs, optimization, and games. In these questions, you can see how to choose the proper data structure for their optimal implementation.

Also included are questions where you can design an algorithm based on the test development-driven approach, which is exceptionally essential for today's companies.

The more prepared and confident you are, the better the chances of negotiating your next salary.

Moises Gamio

Software Engineer, Senior Java Developer

Introduction

Algorithms and Data structures

Data structures refer to how data is organized and stored and can impacts how fast your program runs.

An algorithm is a set of instructions well defined for accomplishing a task or solve a problem.

It is possible to have two different algorithms that solve the same problem, but it matters which one is better in terms of performance.

Abstract Data Types

Abstract Data Type (ADT) is a data type, which specifies a set of data and a set of operations that can perform with that data. It's called abstract because only its behavior is defined but not its implementation.

How to develop a Usable Algorithm

- Model the problem, identifying the elements of the problem.
- Design a sequence of steps algorithm should do, defining the data structures and data types.
- Is the algorithm fast enough and fit in memory? Here you must refactor the algorithm, possibly changing the data structures and the sequence of steps. Big O Notation is your helper in this task.
- Iterate until you are satisfied.
- We are ending possibly in a new abstract data type.

Clean Code

Clean code can be read and enhanced by a developer other than its original author.

If you want to be a better programmer, you must follow these recommendations.

Clean code has Intention-Revealing names

Names reveal intent. Someone who reads your code must understand the purpose of your

variable, function, or Class.

Real situation:	It must be refactored to this:

`int sId; //supplier Id`
`int artDelPrice;`
	`int supplierId;`
	`int articleDeliveredPrice;`

Even with external dependencies:

```
private Z_E2F_RS_Result e2fResult; //ingredients recordset
```

It must be refactored to this:

```
private Z_E2F_RS_Result ingredients;
```

Imagine that we don't have the `//ingredients` comment in `e2fResult` variable. Then, further in any part of our code, when we try to process this variable, we have the following sentence:

```
e2f = e2fResult[i];
```

And we don't know what does `e2f` means! Well, someone suggests asking the person responsible for this code. But that guy is not at the office. Well, send it an email, and he is on holiday!

But if, instead, we adopt names that *reveal intent* from the beginning, we could avoid these catastrophic scenarios.

```
ingredient = ingredients[i];
```

Throughout this book, you will see how important these practices are to write efficient and readable algorithms.

Test-Driven Java Development

Test-Driven Development (TDD) is a process, which is based on a procedure called red-green-refactor. This procedure consists of a few steps that are repeated over and over again: Write a test, run all tests, write an implementation code, run all tests, refactor code, run all tests.

While writing tests, we are in the red state. When the test implementation is ready, all tests should pass, and then we will be in the green state.

In Java, we use JUnit libraries to implement our test cases. Throughout this book, we will omit, in some cases, the definition of a Java Test Class and focus only on the methods with *@Test* annotations.

1.0 Fundamentals

Arrays

An array is an object that stores a fixed number of elements of the same data type. It uses a contiguous memory location to store the elements. Its numerical index accesses each element.

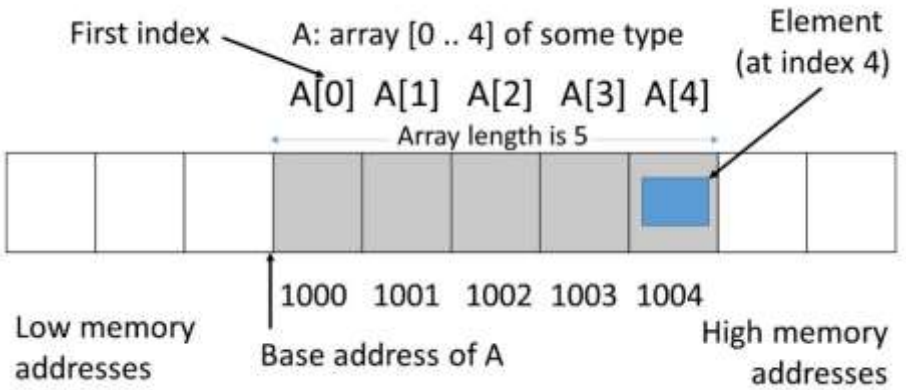


Figure 1.0.1 Array

If you ask the Array, give me the element at index 4, the computer locates that element's cell in a single step.

That happens because the computer finds the memory address where the Array begins - 1000 in the figure above - and adds 4, so the element will be located at memory address 1004.

Arrays are a linear data structure because the elements are arranged sequentially and accessed randomly.

One of the limitations of the Array is that adding or deleting data takes a lot of time.

In a one-dimensional array, you retrieve a single value when accessing each index.

```
//declares an array of integers

int[] arrayOfInts;

//allocates memory for 10 integers

arrayOfInts = new int[10];
```

```
//initialize first element

arrayOfInts[0] = 100;

//initialize fourth element

arrayOfInts[3] = 200;

//get contents of fourth element

int temp = arrayOfInts[3];
```

In a multidimensional array, you retrieve an array when accessing each index. A two-dimensional array is known as a matrix.

Arrays allow us to have one variable store multiple values.

For example, we define an array of products 2-D, whose elements per row represents: *productId, price,* and *grams*.

```
double[][] arrayOfProducts = new double[4][3];
arrayOfProducts[2][0] = 2;
arrayOfProducts[2][1] = 1.29;
arrayOfProducts[2][2] = 240;
```

The following figure shows the representation of this Array of arrays:

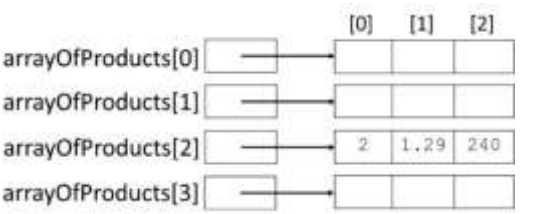


Figure 1.0.2 Multidimensional Array

Once an array is created, it cannot change its size. For dynamic arrays, we can use a List.

An ArrayList is a resizable array implementation of the List interface.

```
List<String> listOfCities = new ArrayList<>();
listOfCities.add("New York");
listOfCities.add("Berlin");
listOfCities.add("Paris");
listOfCities.add("Berlin");

//Traversing list through the for-each loop

for (String city : listOfCities)
  System.out.println(city);
```

Strings

A sequence of character data is called a string and is implemented by the String class.

There are two ways to create a String object:

1. By string literal

```
String myString = "literal of ";
```

The java compiler creates and places a new string instance in the string constant pool. This avoids to create two instances with the same value.

2. By new keyword

```
String s = new String("hello world");
```

The java compiler creates a new string object in the heap memory. The variable s refers to the object.

```
myString + "chars."
```

The previous concatenation creates a third String object in memory. That is because Strings are immutable, they cannot change once it is created. Use a mutable StringBuilder class if you want to manipulate the contents of the string on the fly.

```
StringBuilder stringOnTheFly = new StringBuilder();
stringOnTheFly.append(myString).append("chars.");
```

String, *StringBuffer* and *StringBuilder* implements the *CharSequence* interface that is used to represent a sequence of characters.

1.1 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}
chars[idx] = a
chars[arrayLength-1-idx] = 2

When idx = 1:
chars = {2, b, c, 2, 1, 3, a}
chars[idx] = b
chars[arrayLength-1-idx] = 3

When idx = 2:
chars = {2, 3, c, 2, 1, b, a}
chars[idx] = c
chars[arrayLength-1-idx] = 1

When idx = 3:
chars = {2, 3, 1, 2, c, b, a}
idx is not less than arrayLength/2
end
```

Tests

```
@Test
public void reverseText_useCases() {
  assertEquals("abc2132", StringUtils.reverse("2312cba"));
  assertEquals("ba", StringUtils.reverse("ab"));
  assertEquals("c a1", StringUtils.reverse("1a c"));
}
```

During the interview, it is common to receive additional questions about your code. For instance, what happens if we pass a *null* argument.

First, we need to define our test case

```
@Test(expected = RuntimeException.class)
public void reverseText_exceptionThrownCase() {
  assertEquals("cda1", StringUtils.reverse(null));
}
```

To avoid a *NullPointerException*, we need to add the following precondition:

```
if (text == null)
      throw new RuntimeException("text is not initialized");
```

Moreover, the interviewer wants our algorithm to reverse only those characters that occupy an odd position inside the Array.

Again, we define our assumption using a test case.

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

The % operator is used to detect these locations. Under the loop *for* sentence, we need to add the following conditional sentence:

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

What is the performance of this 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).

In order to know why Big O Notation ignores constants, let's see the following example:

When `chars.length` is 100, then:

Total run time:

O(1) + O(1) + O(100/2) + O(1) = 1 + 1 + 50 + 1 = 53

But 53 ≈ 50 → (N/2)

If 50 executions depend on N/2, and N/2 depend on N

Then, we can say that 50 executions depend on N as well.

When `chars.length` is 100000, then:

Total run time:

O(1) + O(1) + O(100000/2) + O(1) = 1 + 1 + 50000 + 1 = 50003

But 50003 ≈ 50000 → (N/2)

If 50000 executions depend on N/2, and N/2 depend on N

Then, we can say that 50000 executions depend on N as well.

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

1.2 Validate If a String Has All Unique Characters

Given a string of characters, validate if all of them are unique.

Solution

We assume the charset is ASCII, but you should always ask interviewers if you are unsure. Originally based on the English alphabet, ASCII encodes 128 specified characters into seven-bit integers. Ninety-five of the encoded characters are printable: these include the digits *0* to *9*, lowercase letters *a* to *z*, uppercase letters *A* to *Z*, and punctuation symbols.

Firstly, create a Boolean array to store the occurrence of every character.

Secondly, iterate the string of characters, use the java *charAt* method to return the numerical representation for every character. Check this value in the Boolean Array. If it exists, then the string does not have unique values. Otherwise, store this value in the Boolean Array as its first occurrence.

Finally, if all characters had only one occurrence in the Boolean Array, the string has all unique characters. Listing 1.2 shows the algorithm.

`Listing 1.2 – Validate if a String Has All Unique Characters`

```
public class StringUtils {
  public static boolean areUniqueChars(String str) {
    if (str.length() > 128)
      return false;

    boolean[] booleans = new boolean[128];
    for (int idx = 0; idx < str.length(); idx++) {
      int value = str.charAt(idx);
      if (booleans[value]) { //is found?
        return false;
      }
      booleans[value] = true;
    }
    return true;
  }
}
```

Tests

```
public class AreUniqueCharsTest {
  @Test
  public void is_not_UniqueChars() {
    assertFalse(StringUtils.areUniqueChars("29s2"));
    assertFalse(StringUtils.areUniqueChars("1903aio9p"));
  }
  @Test
  public void is_UniqueChars() {
    assertTrue(StringUtils.areUniqueChars("29s13"));
    assertTrue(StringUtils.areUniqueChars("2813450769"));
  }
}
```

1.3 Validate If a String Is a Palindrome

A palindrome is a string that reads the same forward and backward, for example, *level, wow,* and *madam.*

Solution

We can loop through each character and check it against another one on the opposite side. If one of these checks fails, then the text is not Palindrome. Listing 1.3 shows the algorithm.

`Listing 1.3 – Validate If a String Is a Palindrome`

```
public class StringUtils {
  public static boolean isPalindrome(String text) {
    final int length = text.length();
    for (int idx = 0; idx < length / 2; idx++) {
      if (text.charAt(idx) != text.charAt(length - 1 - idx))
        return false;
    }
    return true;
  }
}
```

Tests

```
public class IsPalindromeTest {
  @Test
  public void is_not_palindrome() {
    assertFalse(StringUtils.isPalindrome("2f1"));
    assertFalse(StringUtils.isPalindrome("-101"));
  }
  @Test
  public void is_palindrome() {
    assertTrue(StringUtils.isPalindrome("2f1f2"));
    assertTrue(StringUtils.isPalindrome("-101-"));
    assertTrue(StringUtils.isPalindrome("9"));
    assertTrue(StringUtils.isPalindrome("99"));
    assertTrue(StringUtils.isPalindrome("madam"));
  }
}
```

1.4 Compare Application Version Numbers

Semantic versioning is a formal convention for specifying compatibility using a three-part version number: *major* version, *minor* version, and *patch*. Minor changes and bug fixes increment the *patch* number, which does not change the software's application programming interface (API). Given version1 and version2, returns:

* -1 if version1 < version2

* 1 if version1 > version2

* 0 if version1 == version2

Solution

Firstly, we must split the version number into its components. Since the input is in the form of strings, there is a need to divide them, given the dot delimiter that separates them to convert them to int arrays.

Secondly, we must iterate the arrays while it has not achieved at least one of the lengths of

the arrays. Compare each component from the left to the right side. If one version contains one more component than the other one, then check that a value of 0 in that additional component is not representative.

Finally, we return 0 when versions were not different during the iteration.

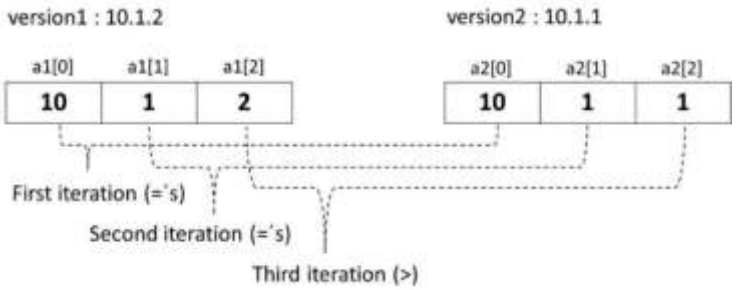


Figure 1.4 Compare Application Version Numbers

Listing 1.4 – Compare Application Version Numbers

```
public class VersionNumber {
  public static int compare(String v1, String v2) {
    int[] a1 = Arrays.stream(v1.split("\\."))
            .map(String::trim)
            .mapToInt(Integer::parseInt).toArray();
    int[] a2 = Arrays.stream(v2.split("\\."))
            .map(String::trim)
            .mapToInt(Integer::parseInt).toArray();
    int idx = 0;
    while (idx < a1.length || idx < a2.length) {
      if (idx < a1.length && idx < a2.length) {
        if (a1[idx] < a2[idx]) {
          return -1;
        } else if (a1[idx] > a2[idx]) {
          return 1;
        }
      } else if (idx < a1.length) {
        if (a1[idx] != 0) {
          return 1;
        }
      } else if (idx < a2.length) {
        if (a2[idx] != 0) {
          return -1;
        }
      }
      idx++;
    }
    return 0;
  }
}
```

Tests

```
@Test
public void versionNumber_usesCases() {
  assertEquals(0, VersionNumber.compare("15", "15.0"));
  assertEquals(0, VersionNumber.compare("10.1", "10.1.0"));
  assertEquals(-1, VersionNumber.compare("10.1", "10.1.1"));
  assertEquals(1, VersionNumber.compare("10.1.2", "10.1.1"));
}
```

1.5 Remove Duplicates from a List

Write a program to check if a list has any duplicates, and if it does, it removes them.

Solution

Maybe your first idea is to iterate through the list. You compare every item with the other ones. If a duplicate is detected, then it is removed. Or maybe you traverse the list and store the first occurrence of each item in a new list and ignore all the next occurrences of that item. Those solutions are called brute force algorithms because they use straightforward methods of solving a problem, but sometimes what the interviewer expects is to reuse the libraries included in the JDK, which improves efficiency.

We use the *Set* interface from the Collections library. By definition, *Set* does not allow duplicates. Then, use the *sort* method to order its items.

A *Comparable* interface sorts lists of custom objects in natural ordering. List of Objects that already implement *Comparable* (e.g., String) can be sorted automatically by *Collections.sort*.

LinkedHashSet (Collection list) is used to initialize a *HashSet* with the list items, removing the duplicates. Listing 1.5 shows this implementation.

Listing 1.5 – Remove Duplicates from a List.

```
public class ListUtils {
  public static <E extends Comparable<E>>
  List<E> removeDuplicatesAndOrder(List<E> list) {
    Set<E> set = new LinkedHashSet<>(list);
    ArrayList<E> arrayList = new ArrayList<>(set);
    Collections.sort(arrayList);
    return arrayList;
  }
}
```

Tests

```
@Test
public void givenStringsThenRemovedDuplicates() {
  List<String> input = new ArrayList<>();
  input.add("c");
  input.add("b");
  input.add("b");
  input.add("d");
  input.add("c");
  input.add("a");
  List<String> result = ListUtils.removeDuplicatesAndOrder(input);
  assertEquals("[a, b, c, d]", result.toString());
}

@Test
public void givenIntegersThenRemovedDuplicates() {
  List<Integer> input = new ArrayList<>();
  input.add(Integer.valueOf(3));
  input.add(Integer.valueOf(3));
  input.add(Integer.valueOf(4));
  input.add(Integer.valueOf(1));
  input.add(Integer.valueOf(7));
  input.add(Integer.valueOf(1));
  input.add(Integer.valueOf(2));
  input.add(Integer.valueOf(1));
  List<Integer> result = ListUtils.removeDuplicatesAndOrder(input);
  assertEquals("[1, 2, 3, 4, 7]", result.toString());
}
```

1.6 Rotate a 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. We swap the elements with the corresponding cell in the matrix in every square cycle by using a temporary variable. Listing 1.6 shows the algorithm.

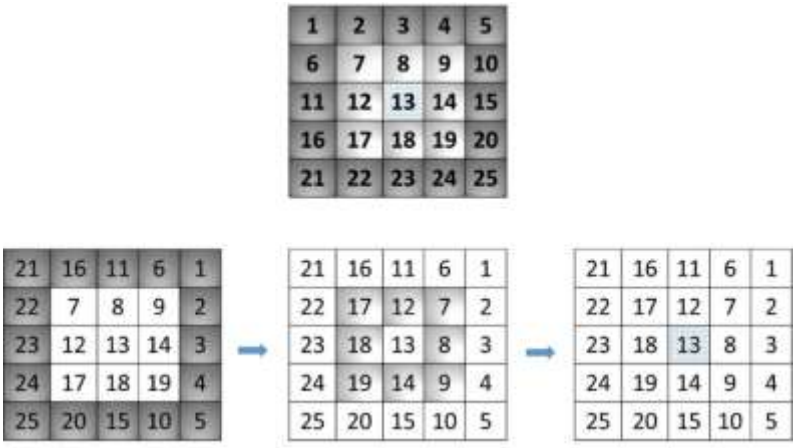


Figure 1.6 Rotate a Matrix by 90 Degrees

Listing 1.6 – Rotate a 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 clockwise direction
        //temp = top-left
        int temp = matrix[i][j];
        //top-left <- bottom-left
        matrix[i][j] = matrix[n - 1 - j][i];
        //bottom-left <- bottom-right
        matrix[n - 1 - j][i] = matrix[n - 1 - i][n - 1 - j];
        //bottom-right <- top-right
        matrix[n - 1 - i][n - 1 - j] = matrix[j][n - 1 - i];
        //top-right <- top-left
        matrix[j][n - 1 - i] = temp;
      }
    }
  }
}
```

Tests

```
@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]);
}
```

1.7 Items in Containers

Amazon would like to know how much inventory exists in their closed inventory compartments. Given a string *s* consisting of items as "*" and closed compartments as an open and close "|", an array of starting indices *startIndices* and an array of ending indices *endIndices*, determine the number of items in closed compartments within the substring between the two indices, inclusive.

- An item is represented as an asterisk ('*' = ascii decimal 42)
- A compartment is represented as a pair of pipes that may or may not have items between them ('|' = ascii decimal 124).

Example

s = '|**|*|*'

startIndices = [1,1]

endIndices = [5,6]

The string has a total of 2 closed compartments, one with 2 items and one with 1 item. For the first pair of indices, *(1,5),* the substring is '|**|*'. There are 2 items in a compartment.

For the second pair of indices, *(1,6),* the substring is '|**|*|' and there are *2 + 1 = 3* items

in compartments.

Both of the answers are returned in an array. *[2, 3]*.

Function Description

Write a function that returns an integer array that contains the results for each of the *startIndices[i]* and *endIndices[i]* pairs.

The function must have three parameters:

- *s*: A string to evaluate
- *startIndices*: An integer array, the starting indices.
- *endIndices*: An integer array, the ending indices.

Constraints

- $1 \leq m, n \leq 10^5$
- $1 \leq startIndices[i] \leq endIndices[i] \leq n$
- Each character of *s* is either '*' or '|'

Solution

To determine the number of items in closed compartments, we need to build the substrings from the two indices *startIndices* and *endIndices*.

We evaluate every character from the substring. All strings start with a '|' character. We define a numOfAsterisk variable to count items inside a compartment.

We define a *wasFirstPipeFound* variable to initialize our *numOfAsterisk* variable the first time a '|' character is found, and we accumulate all items since subsequent '|' characters.

```
Listing 1.7 – Items in Containers.
```

```
import java.util.ArrayList;
import java.util.List;
public class Container {
  public static List<Integer> numberOfItems(String s,
    List<Integer> startIndices, List<Integer> endIndices) {

    if (startIndices.size()<1 || startIndices.size()>100000)
      throw new RuntimeException("wrong size in startIndices");

    if (endIndices.size()<1 || endIndices.size()>100000)
      throw new RuntimeException("wrong size in endIndices");
```

```
    List<Integer> numberOfItemsInClosedCompartments = new ArrayList<>();
    for (int idx=0; idx<startIndices.size(); idx++) {
      int start = startIndices.get(idx);
      int end = endIndices.get(idx);

      if (start<1 || start>100000)
        throw new RuntimeException("wrong value at startIndices");

      if (end<1 || end>100000)
        throw new RuntimeException("wrong value at endIndices");

      int numOfAsterisk = 0;
      boolean wasFirstPipeFound  = false;
      int numOfAsteriskAccumulated = 0;
      for (char c : s.substring(start-1, end).toCharArray()) {
        if (c == '|') {
          if (wasFirstPipeFound == true) {
            numOfAsteriskAccumulated +=numOfAsterisk;
            numOfAsterisk = 0;
          } else {
            wasFirstPipeFound = true;
            numOfAsterisk = 0;
          }
        } else if (c == '*') {
          numOfAsterisk +=1;
        } else {
          throw new RuntimeException("wrong character");
        }
      }
      numberOfItemsInClosedCompartments.add(numOfAsteriskAccumulated);
    }
  return numberOfItemsInClosedCompartments;
  }
}
```

Test

```
@Test
public void test_numberOfItems() {
  assertEquals(new ArrayList<Integer>(Arrays.asList(2,3)),
    Container.numberOfItems("|**|*|*",
            new ArrayList<Integer>(Arrays.asList(1,1)),
            new ArrayList<Integer>(Arrays.asList(5,6))
  ));
}
```

1.8 Shopping Options

An Amazon customer wants to buy a pair of jeans, a pair of shoes, a skirt, and a top but has a limited budget in dollars. Given different pricing options for each product, determine how many options our customer has to buy 1 of each product. You cannot spend more money than the budgeted amount.

Example

```
priceOfJeans = [2,3]
priceOfShoes = [4]
priceOfSkirts = [2,3]
priceOfTops = [1,2]
budgeted = 10
```

The customer must buy shoes for 4 dollars since there is only one option. This leaves 6 dollars to spend on the other 3 items. Combinations of prices paid for jeans, skirts, and tops respectively that add up to 6 dollars or less are *[2,2,2], [2,2,1], [3,2,1], [2,3,1]*. There are 4 ways the customer can purchase all 4 items.

Function description

Create a function that returns an integer which represents the number of options present to buy the four items.

The function must have 5 parameters:

int[] priceOfJeans: An integer array, which contains the prices of the pairs of jeans available.

int[] priceOfShoes: An integer array, which contains the prices of the pairs of shoes available.

int[] priceOfSkirts: An integer array, which contains the prices of the skirts available.

int[] priceOfTops: An integer array, which contains the prices of the tops available.

int dollars: the total number of dollars available to shop with.

Constraints

- $1 \leq$ length(priceOfJeans, priceOfShoes, priceOfSkirts, priceOfTops) $\leq 10^3$
- $1 \leq$ dollars, prices $\leq 10^9$

Solution

To find how many ways the customer can purchase all four items, we can iterate the four arrays, combine all its products, and validate that customer cannot spend more money than the budgeted amount. The for-each construct helps our code be elegant and readable and there is no use of the index.

```
int numberOfOptions = 0;
for (int priceOfJean : priceOfJeans) {
  for (int priceOfShoe : priceOfShoes) {
    for (int priceOfSkirt : priceOfSkirts) {
      for (int priceOfTop : priceOfTops) {
        if (priceOfJean + priceOfShoe + priceOfSkirt + priceOfTop <= dollars)
          numberOfOptions +=1;
      }
    }
  }
}
```

This algorithm works fine when array size and values are small. But based on the constraints, imagine you are processing a *priceOfShoe* value of 100000 at the location *priceOfShoes[101],* and at that moment, the sum of *priceOfJean* + *priceOfShoe* is greater than 10 dollars. Therefore, it does not make sense to continue processing the following possible 1000000000 items of *pricesOfSkits[]* and other 1000000000 items of *priceOfTops[]*.

To skip this particular iteration, we use the *"continue"* statement and proceed with the next iteration in the loop. The next *priceOfShoe* at the location *priceOfShoes[102],* for example. Listing 1.8 shows an optimized solution.

Listing 1.8 – Shopping Options.

```
public class ShoppingOptions {
  public static int getNumberOfOptions(int[] priceOfJeans,
    int[] priceOfShoes, int[] priceOfSkirts, int[] priceOfTops, int dollars) {
    if (dollars < 1 || dollars > 1000000000)
      throw new RuntimeException("wrong value for budget");

    validate(priceOfJeans, "jeans");
    validate(priceOfShoes, "shoes");
    validate(priceOfSkirts, "skirts");
    validate(priceOfTops, "tops");
    int numberOfOptions = 0;
    for (int priceOfJean : priceOfJeans) {
      if (priceOfJean >= dollars)
        continue;
      for (int priceOfShoe : priceOfShoes) {
        if (priceOfJean + priceOfShoe >= dollars)
          continue;
        for (int priceOfSkirt : priceOfSkirts) {
          if (priceOfJean + priceOfShoe + priceOfSkirt >= dollars)
            continue;
          for (int priceOfTop : priceOfTops) {
            if (priceOfJean + priceOfShoe + priceOfSkirt + priceOfTop <= dollars)
              numberOfOptions +=1;
          }
```

```
        }
      }
    }
    return numberOfOptions;
  }

  private static void validate(int[] array, String arrayName) {
    if (array.length < 1 || array.length > 1000)
      throw new RuntimeException("wrong size in array " + arrayName);

    for (int price : array) {
      if (price < 1 || price > 1000000000)
        throw new RuntimeException("wrong value in array " +  arrayName);
    }
  }
}
```

Tests

```
@Test
public void test_shoppingOptions() {
  int[] priceOfJeans = {2, 3};
  int[] priceOfShoes = {4};
  int[] priceOfSkirts = {2, 3};
  int[] priceOfTops = {1, 2};
  assertEquals(4, ShoppingOptions.getNumberOfOptions(priceOfJeans, priceOfShoes,
          priceOfSkirts, priceOfTops, 10));
}

@Test
public void test_shoppingOptionsBigPrices() {
  int[] priceOfJeans = {2, 10000, 3};
  int[] priceOfShoes = {2000002, 4};
  int[] priceOfSkirts = {2, 3000000, 3};
  int[] priceOfTops = {1, 2};
  assertEquals(4, ShoppingOptions.getNumberOfOptions(priceOfJeans, priceOfShoes,
          priceOfSkirts, priceOfTops, 10));
}
```

A function or method should be small, making it easier to read and understand. We have moved all validations to a private method.

What happens when our possible choosing prices are located at the end of the arrays?. One possible solution could be to sort the arrays before iterating and find the right combinations of prices.

Linked Lists

2.0 Fundamentals

A linked list is a linear data structure that represents a sequence of nodes. Unlike arrays, linked lists store items at a not contiguous location in the computer's memory. It connects items using pointers.

Connected data that dispersed is throughout memory are known as nodes. In a linked list, a node embeds data items. Because there are many similar nodes in a list, using a separate class called Node makes sense, distinct from the linked list itself.

Each Node object contains a reference (usually called next or link) to the next Node in the list. This reference is a pointer to the next Node's memory address. A Head is a special node that is used to denote the beginning of a linked list. A linked list representation is shown in the following figure.

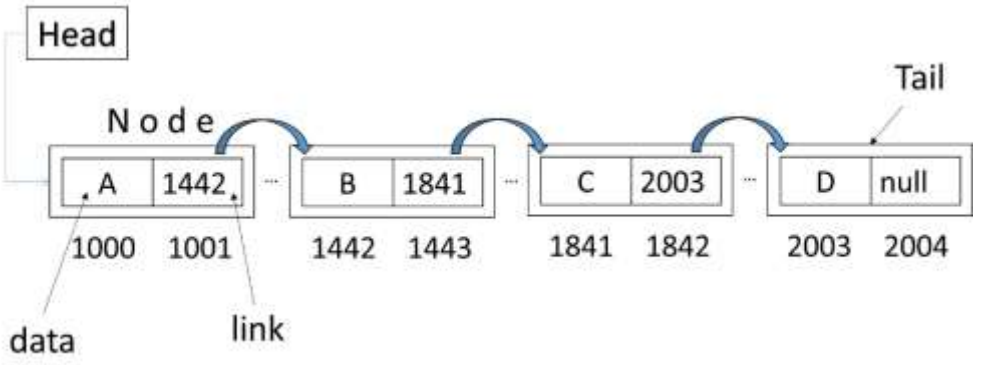


Figure 1.0.1 Linked list

Each Node consists of two memory cells. The first cell holds the actual data, while the second cell serves as a link indicating where the next Node begins in memory. The final Node's link contains null since the linked list ends there.

In the figure above, we say that "B" follows "A," not that "B" is in the second position.

A linked list's data can be spread throughout the computer's memory, which is a potential advantage over the Array. An array, by contrast, needs to find an entire block of contiguous cells to store its data, which can get increasingly difficult as the array size grows. For this reason, Linked Lists utilize memory more effectively.

When each Node only points to the next Node, we have a singly linked list. We have a doubly-linked list when each Node points to the next Node and the previous Node.

If the tail points to the head, then we have a circular singly linked list

The following code represents the Node of a doubly-linked list:

```
private final class Node {
  private int data;
  private Node next;
  private Node prev;
}
```

Unlike an array, a linked list doesn't provide constant time to access the *nth* element. We have to iterate n-1 elements to obtain the *nth* element. But we can insert, remove, and update nodes in constant time from the beginning of a linked list.

2.1 Implement a Linked List

Implement a Linked List that includes methods such as create, add, and traverse.

Solution

Create a linked list class

We can represent a LinkedList as a class with its Node as a separate class. The LinkedList class will have a reference to the Node type.

```
Listing 2.1.1 – Linked List Class
```

```
//Generic linked list
public class LinkedList<T> {
  Node head;

  private class Node {
    final T data;
    Node next;
    //next is by default initialized as null
    Node(T data) {
      this.data = data;
      this.next = null;
    }
  }
}
```

Adding a node

We can add a new node in three ways:

- At the front of the linked list.
- Before/After a given node.
- At the end of the linked list.

We add a new node at the end of the linked list for our purpose.

Steps to add a new Node:

- Create a new node with the given data.
- If the head node is null (Empty Linked List), make the new node the head.
- If the head node is not null, find the last node. Make the last node => next as the new node.

Listing 2.1.2 – Adding a node

```
  public void add(T data) {
    Node node = new Node(data);
    if (head == null) {
      head = node;
    } else {
      Node last = head;
      while (last.next != null) {
        last = last.next;
      }
      last.next = node;
    }
  }
```

Traverse a linked list

Traverse means printing all data included in a linked list by traversing the list from the head node to the last node.

Listing 2.1.3 – Traversing a linked list

```
  @Override
  public String toString() {
    StringJoiner stringJoiner = new StringJoiner(" -> ", "[", "]");
    Node currentNode = head;
    while (currentNode != null) {
      stringJoiner.add(currentNode.data.toString());
      currentNode = currentNode.next;
    }
    return stringJoiner.toString();
  }
```

Tests

```
  @Test
  public void addNodes() {
    LinkedList<String> linkedList = new LinkedList<>();
    linkedList.add("s1");
    linkedList.add("s2");
```

```
    linkedList.add("s3");
    linkedList.add("s4");
    assertEquals("[s1 -> s2 -> s3 -> s4]", linkedList.toString());
  }
```

2.2 Reverse a Linked List

Given a pointer to the head node of a linked list, write a program to reverse the linked list.

Solution

To reverse a linked list, we implement an iterative method:

1. Initialize three-pointers: *prev* as NULL, *current* as head, and *next* as NULL.
2. Iterate through the linked list. Inside the loop, do the following:

// Before changing *next* of *current*, store *next* node

```
next = current->next
```

// Now change *next* of *current,* here is where actual reversing happens

```
current->next = prev
```

// Move *prev* and *current* one step forward

```
prev = current
```

```
current = next
```

```
Listing 2.2 – Reverse a Linked List
```

```
  public void reverse() {
    if (head == null)
      return;

    Node prev = null;
    Node current = head;
    Node next = null;
    while (current != null) {
      next = current.next;
      current.next = prev;
      prev = current;
      current = next;
    }
    head = prev;
  }
```

Let's see what happens at the reverse method in the first iteration of a LinkedList:

[s1 -> s2 -> s3]

next = current.next	next becomes s2, which is the next of s1
current.next = prev	now the next of s1 points to null
prev = current	prev becomes s1
current = next	the next iteration will process the new current = s2

Tests

```
@Test
public void reverseLinkedList() {
  LinkedList<String> linkedList = new LinkedList<>();
  linkedList.add("s1");
  linkedList.add("s2");
  linkedList.add("s3");
  linkedList.add("s4");
  linkedList.reverse();
  assertEquals("[s4 -> s3 -> s2 -> s1]",
          linkedList.toString());
}

@Test
public void reverseIntegersLinkedList() {
  LinkedList<Integer> linkedList = new LinkedList<>();
  linkedList.add(new Integer(1));
  linkedList.add(new Integer(2));
  linkedList.add(new Integer(3));
  linkedList.add(new Integer(5));
  linkedList.reverse();
  assertEquals("[5 -> 3 -> 2 -> 1]",
          linkedList.toString());
}
```

Math and Logic Puzzles

3.0 Fundamentals

Puzzles are usually asked to see how you go about solving a tricky problem. You should know basic math concepts to excel in coding interviews.

Useful methods included in the *Math* Class:

abs(x): Returns the absolute value of x.

Example: abs(-1) = 1;

round(x): Returns the value of x rounded to its nearest integer.

Example: round(4.4) = 4

ceil(x): Returns the value of x rounded up to its nearest integer.

Example: ceil(4,4) = 5.0

floor(x): Returns the value of x rounded down to its nearest integer.

Example: floor(4.4) = 4.0

max(x): Returns the number with the highest value.

Example: max(6,7) = 7

pow(x,y): Returns the value of x to the power of y.

Example: pow(2,5) = 32.0

random(): Returns a random number between 0 and 1.

Example: random() => 0.45544344999209374

Other useful formulas:

Sum of first n positive integers:

$n(n+1)/2$

Distance between two points P(x1,y1) and Q(x2,y2):

$d(P, Q) = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$

Logarithm as an inverse function to exponentiation

$\log_b(b^x) = x$

3.1 Sum of All Even Numbers from 1 to a Given Integer

Given an Integer *N*, calculates the sum of all even numbers from 1 to *N*.

Solution

It looks like an easy program, but believe me, most experienced programmers during the interview forget to use the operator "%," which returns the division remainder. If it is 0, the number is even; otherwise, it is odd. Listing 3.1 shows the algorithm.

Listing 3.1 – Sum of All Even Numbers from 1 to N

```
public class NumberUtils {
  public static int sumOfEvenNumbers(int N) {
    int sum = 0;
    for (int number = 1; number <= N; number++)
      if ((number % 2) == 0)
        sum = sum + number;

    return sum;
  }
}
```

Our code runs properly, but as Software Engineers, we need to care about code efficiency. The following algorithm runs faster than the previous one because it only loops the N/2 times.

```
public class NumberUtils {
  public static int sumOfEvenNumbers(int N) {
    int sum = 0;
    int number = 2;
    while (number <= N) {
      sum += number;
      //increase number by 2, which in definition
      //is the next even number
      number+=2;
    }
    return sum;
  }
}
```

Tests

```
  @Test
  public void sumOfEvenNumbers_test() {
    assertEquals(42, NumberUtils.sumOfEvenNumbers(12));
    assertEquals(110, NumberUtils.sumOfEvenNumbers(21));
  }
```

3.2 Validate a Leap Year

A leap year is a calendar year containing an additional day. This extra day occurs in February. The following rules define a leap year:

- A year will be a leap year if it is divisible by 4, but not by 100

  or

- A year will be a leap if it is divisible by 400.

Solution

We use the % operator to verify if a year is divisible by 4, 100, or 400. Listing 3.2 shows the algorithm.

Listing 3.2 – Validate a Leap Year

```
public class DateUtils {
  public static boolean isLeapYear(int year){
    return (year % 400 == 0) ||
           (year % 4 == 0 && year % 100 != 0);
  }
}
```

This algorithm had a real application in the Y2K problem, where COBOL programs in a Bank used to store four-digit years with only the final two digits, so you could not distinguish 2000 from 1900. In addition to calculating a leap year:

For years represented from 50 until 99, add 19 at begin, resulting in 1950 until 1999.

For years represented from 00 until 49, add 20 at begin, resulting in 2000 until 2049.

The Bank assumed that before 2049 all systems would be migrated to modern programming languages (digital transformation), but that is another story.

Tests

```
  @Test
  public void isLeapYear() {
    assertTrue(DateUtils.isLeapYear(400));
    assertTrue(DateUtils.isLeapYear(2000));
    assertTrue(DateUtils.isLeapYear(2020));
  }
  @Test
  public void is_notLeapYear() {
    assertFalse(DateUtils.isLeapYear(401));
    assertFalse(DateUtils.isLeapYear(2018));
  }
```

3.3 Given an Integer N, Returns the Smallest Number of Digits

Write a method that, given an original number *N* of *d* digits, returns the smallest number with the same number of digits. For example, given N=4751, the method should return 1000. Given N=1, the method should return 0.

Solution

Maybe the first idea that comes to our minds could be to iterate from the given number and decrease one by one, and in every iteration to check if every new number contains one digit less than the previous number. If the answer is True, then the previous one is the smallest number with the same number of digits as the original number. Listing 3.3 shows the first algorithm for positive numbers.

Listing 3.3 – Given an Integer N, Returns the Smallest Number of Digits

```
public class NumberUtils {
  public static int smallest(int N) {
    int smallestNumber = 0;
    if (N <= 1)
      return smallestNumber;

    int numberOfDigitsOriginalN = String.valueOf(N).length();
    while (N > 0) {
      N--;
      if (String.valueOf(N).length() ==
              (numberOfDigitsOriginalN - 1)) {
        return ++N;
      }
    }
    return smallestNumber;
  }
}
```

But if we realize, the solution follows a particular pattern:

N = 4751 -> smallest number = 1000

N = 189 -> smallest number = 100

N = 37 -> smallest number = 10

The smallest number is a power of 10, where the exponent is: *number of digits – 1*

Second solution:

```
public static int smallest(int N) {
  int smallestNumber = 0;
  if (N <= 1 || String.valueOf(N).length() == 1) {
    return smallestNumber;
  }
  int numberOfDigits = String.valueOf(N).length();
  return (int) Math.pow(10, numberOfDigits - 1);
}
```

If we want to include negative numbers, we must consider the smallest number with the same number of digits and the same sign. Here the solution:

```
public class NumberUtils {
  public static int smallest(int N) {
    int numberOfDigits = (int) String.valueOf(Math.abs(N)).length();
    if (N >= 0) {
      if (numberOfDigits == 1) {
        return 0;
      } else {
        return (int) Math.pow(10, numberOfDigits - 1);
      }
    } else
      return 1 - (int) Math.pow(10, numberOfDigits);
  }
}
```

The main idea in Analysis of Algorithms is always to improve the algorithm performance by reducing the number of steps and comparisons. The simpler and more intuitive an algorithm is, the more useful and efficient it will be.

Tests

```
@Test
public void test_right_smallest_values() {
  assertTrue(NumberUtils.smallest(4751) == 1000);
  assertTrue(NumberUtils.smallest(189) == 100);
  assertTrue(NumberUtils.smallest(37) == 10);
  assertTrue(NumberUtils.smallest(1) == 0);
  assertTrue(NumberUtils.smallest(0) == 0);
  assertTrue(NumberUtils.smallest(-1) == -9);
  assertTrue(NumberUtils.smallest(-38) == -99);
}

@Test
public void test_wrong_smallest_values() {
  assertFalse(NumberUtils.smallest(8) == 1);
  assertFalse(NumberUtils.smallest(2891) == 2000);
}
```

3.4 Fizz-Buzz

Write a program that will display all the numbers between 1 and 100.

- For each number divisible by three, the program will display the word "Fizz."
- For each number divisible by five, the program will display the word "Buzz."
- For each number divisible by three and five, the program will display the word "Fizz-Buzz."

The output will look like this:

1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, Fizz-Buzz, 16, 19, …

Solution

It looks like a simple algorithm but is "hard" for some programmers because they try to follow the following reasoning:

```
if (theNumber is divisible by 3) then
     print "Fizz"
else if (theNumber is divisible by 5) then
     print "Buzz"
else /* theNumber is not divisible by 3 or 5 */
     print theNumber
end if
```

But where do we print "Fizz-Buzz" in this algorithm? The interviewer expects that you think for yourself and made good use of conditional without duplication. Realizing that a number divisible by 3 and 5 is also divisible by 3*5 is the key to a FizzBuzz solution. Listing 3.4 shows the algorithm.

Listing 3.4 – Fizz-Buzz

```
public class NumberUtils {
  public static void fizzBuzz(int N) {
    final String BUZZ = "Buzz";
    final String FIZZ = "Fizz";
    for (int i = 1; i <= N; i++) {
      if (i % 15 == 0) {
        System.out.print(FIZZ + "-" + BUZZ + ", ");
      } else if (i % 3 == 0) {
        System.out.print(FIZZ + ", ");
      } else if (i % 5 == 0) {
        System.out.print(BUZZ + ", ");
      } else {
        System.out.print(i + ", ");
      }
    }
  }
}
```

You should never use *System.out.print* in a Production environment. I/O routines consume a lot of resources in time and memory.

Tests

```
public class FizzBuzzTest {
  @Test
  public void testFizzBuzz() {
    NumberUtils.fizzBuzz(100);
  }
}
```

3.5 Verify If a Number Is a Valid Power of a Base

In mathematics, a function is an expression that maps an independent variable (domain) to a dependent variable (range). Functional programming (FP) emphasizes functions that produce results that depend *only on their inputs and not on the program state*—i.e., pure mathematical functions.

One of Java's approaches to FP is the definition of functional interfaces. A functional interface in Java has only one abstract method. We can use functional components available in the Java 8 library like the *Java.util.function.BiFunction<T, U, R>,* which accepts two arguments and produces one result.

```
BiFunction<Integer, Integer, Integer> sum = (x1, x2) -> x1 + x2;
Integer result = sum.apply(2, 3);
System.out.println(result); // 5
```

Exponentiation involves two numbers: the base *b* and the exponent or power *n*. Exponentiation corresponds to repeated multiplication of the base *n* times. For instance, in the following expression: $3^5 = 243$, we say that 243 is the *5th* power of 3. Therefore, 243 is the correct power of 3.

Solution

We do the inverse operation to verify if a number is a valid power nth of another number (base).

Divide the given number by the *base* and evaluate if it provides a 0 remainder, then we iterate this operation until we find a value of 1. Otherwise, if the rest is not 0, then the given number is not a valid *power nth* of the *base*. Listing 3.5 shows a function, which returns true when a given number is the right power of a base.

```
Listing 3.5 – Verify If a Number Is a Valid Power of a Base
```

```
import java.util.function.BiFunction;
public class IsNumberAValidPowerOfBase implements
  BiFunction<Integer, Integer, Boolean> {

  @Override
  public Boolean apply(Integer number, Integer base) {
    return isNumberAValidPowerOfBase(number, base);
  }

  static boolean isNumberAValidPowerOfBase(int number, int base) {
    if (number == 0)
      return true;

    while (number != 1) {
      if ((number % base) != 0)
        return false;

      number = number / base;
    }
    return true;
  }
}
```

Tests

```
@Test
public void testOfWrongReturnValues() {
  assertFalse(isNumberAValidPowerOfBase.apply(6, 2));
  assertFalse(isNumberAValidPowerOfBase.apply(16, 5));
  assertFalse(isNumberAValidPowerOfBase.apply(14, 7));
}

@Test
public void testOfValidReturnValues() {
  assertTrue(isNumberAValidPowerOfBase.apply(243, 3));
  assertTrue(isNumberAValidPowerOfBase.apply(16, 4));
  assertTrue(isNumberAValidPowerOfBase.apply(125, 5));
}
```

3.6 Validate If a Number Is Prime or Not

A prime number is a whole number greater than one, whose only factors are one and itself. A factor is a whole number that can be divided evenly into another number. For instance: 2, 3, 5, 7, 11 are prime numbers.

Solution

A simple solution is to iterate decreasingly through all numbers from the half of the given

number *n*, and for every number, check if it divides *n*. If we find any number that divides, then we return false. Otherwise, it returns true. Listing 3.6 shows the algorithm.

Listing 3.6 – Validate If a Number Is Prime or Not

```
public class MathUtils {
  public static boolean isPrimeNumber(int number) {
    if (number < 2)
      return false;

    if (number == 2)
      return true;

    for (int div = (number / 2) + 1; div > 1; div--) {
      if (number % div == 0) {
        return false;
      }
    }
    return true;
  }
}
```

Tests

```
  @Test
  public void notPrimeNumbers() {
    assertFalse(MathUtils.isPrimeNumber(-1));
    assertFalse(MathUtils.isPrimeNumber(625));
    assertFalse(MathUtils.isPrimeNumber(4));
    assertFalse(MathUtils.isPrimeNumber(100));
  }

  @Test
  public void primeNumbers() {
    assertTrue(MathUtils.isPrimeNumber(2));
    assertTrue(MathUtils.isPrimeNumber(3));
    assertTrue(MathUtils.isPrimeNumber(5));
    assertTrue(MathUtils.isPrimeNumber(7));
    assertTrue(MathUtils.isPrimeNumber(73));
  }
```

3.7 Distance Between Two Points

Given two points, calculate the distance between them.

Solution

Each point has coordinates *(x,y)*, so we can calculate the distance with the hypotenuse.

$d(A, B) = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$

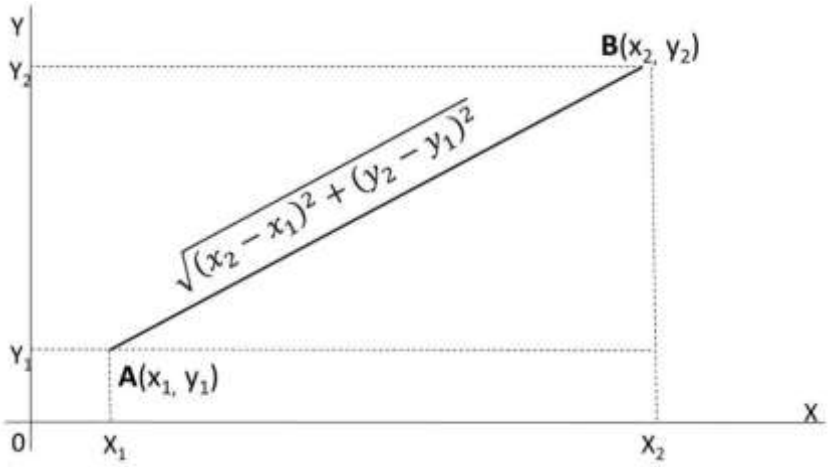


Figure 3.7 Distance Between two Points

Listing 3.7 - Distance Between Two Points

```
public class Point {
  final double x;
  final double y;

  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public double distance(Point point) {
    if (this == null || point == null)
      throw new RuntimeException("Points are not initialized");

    double dx = this.x - point.x;
    double dy = this.y - point.y;

    return Math.sqrt(dx * dx + dy * dy);
  }
}
```

Tests

```
@Test
public void given_twoPoints_return_distance() {
  Point point1 = new Point(2, 3);
  Point point2 = new Point(5, 7);
  assertEquals(5, point1.distance(point2), 0);
}
```

3.8 Write an Immutable Class to convert Currencies

Design a Money Class, which can convert Euros to Dollars and vice versa. As examples, write two instances with the following values: 67.89 EUR and 98.76 USD

Solution

An immutable class is a class whose instances cannot be modified. Its information is fixed for the lifetime of the object without changes.

To make a class immutable, we follow these rules:

- Don't include a mutators method that could modify the object's state.
- Don't allow to extend the Class.
- Make all class members final and private
- Ensure exclusive access to any mutable components. Don't make references to those objects. Make defensive copies.

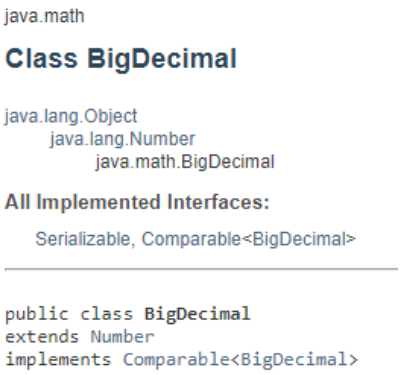
Immutable objects are thread-safe; they require no synchronization. We use a *BigDecimal* data type for our Class because it provides operations on numbers for arithmetic, rounding and can handle large floating-point numbers with great precision. Listing 3.8 shows an immutable Class.

Listing 3.8 - Money Class

```
import java.math.BigDecimal;
public final class Money {
  private static final String DOLAR = "USD";
  private static final String EURO = "EUR";
  private static int ROUNDING_MODE = BigDecimal.ROUND_HALF_EVEN;
  private static int DECIMALS = 2;
  private BigDecimal amount;
  private String currency;

  public Money() {
  }

  public static Money valueOf(
          BigDecimal amount,
          String currency) {
    return new Money(amount, currency);
  }
```

```
  //Caller cannot see this private constructor
  private Money(
          BigDecimal amount,
          String currency) {
    this.amount = amount;
    this.currency = currency;
  }

  //Currency converter
  public Money multiply(BigDecimal factor) {
    return Money.valueOf(
            rounded(this.amount.multiply(factor)),
            this.currency.equals(DOLAR) ? EURO : DOLAR);
  }

  //round to 2 decimals
  private BigDecimal rounded(BigDecimal amount) {
    return amount.setScale(DECIMALS, ROUNDING_MODE);
  }

  public BigDecimal getAmount() {
    return amount;
  }

  public String getCurrency() {
    return currency;
  }
}
```

String class produces immutable objects, so we can trust any client who passes it in the arguments. But this is not the case for *BigDecimal*, which can be extended and manipulated for some untrusted clients, e.g., to expand its *toString()* method.


java.math

Class BigDecimal

java.lang.Object
  java.lang.Number
    java.math.BigDecimal

All Implemented Interfaces:

Serializable, Comparable<BigDecimal>

```
public class BigDecimal
extends Number
implements Comparable<BigDecimal>
```

To protect our Class from untrusted clients, we can create copies of these arguments. The following code shows the *multiply* method modified.

```
  //Currency converter, more secure
  public Money multiplysecure(BigDecimal factor) {
    if (factor.getClass() == BigDecimal.class)
      factor = new BigDecimal(factor.toString());
    else {
      //TODO throw exception?
    }
    return Money.valueOf(
            rounded(this.amount.multiply(factor)),
            this.currency.equals(DOLAR) ? EURO : DOLAR);
  }
```

Tests:

```
@Test
public void convert_EURO_to_DOLLAR() {
  final Money moneyInEuros = Money.valueOf(new BigDecimal("67.89"), "EUR");
  final Money moneyInDollar =
            moneyInEuros.multiply(new BigDecimal("1.454706142288997"));
  assertEquals(new BigDecimal("98.76"), moneyInDollar.getAmount());
}

@Test
public void convert_DOLLAR_to_EURO() {
  final Money moneyInDollar = Money.valueOf(new BigDecimal("98.76"), "USD");
  final Money moneyInEuros =
            moneyInDollar.multiplysecure(new BigDecimal("0.6874240583232078"));
  assertEquals(new BigDecimal("67.89"), moneyInEuros.getAmount());
}
```

3.9 Number of Products of Two Consecutive Integers

Given two integers *X* and *Y*, returns the number of integers from the range [*X* .. *Y*], which can be expressed as the product of two consecutive integers, e.g., *N*(N+1)* for some integer *N*.

Solution

We need to find the total number of products in this range [*X* .. *Y*].

Example:

Given X=6 and Y=20

The function should return 3

These integers are **6**=2*3, **12**=3*4, and **20**=4*5.

We could iterate from 1 until the upper limit=Y, checking if N*(N+1) <=Y. But if the lower limit is 1000, the first small products with two consecutive integers, such as 1*2, 2*3, 3*4, do not fit in the range [1000, 1130]. It is better to start the iteration from the square root of the lower limit X. Listing 3.9 shows the algorithm.

Listing 3.9 – Number of Products of Two Consecutive Integers

```
public class NumberUtils {
  public static int validProducts(int X, int Y) {
    if (X < 1)
      throw new IllegalArgumentException("invalid input: " + X);

    int N = (int) Math.sqrt(Integer.valueOf(X).doubleValue());
    int numberOfValidProducts = 0;
    while (N * (N + 1) <= Y) {
      int product = N * (N + 1);
      if (product >= X && product <= Y) {
        numberOfValidProducts++;
      }
      N++;
    }
    return numberOfValidProducts;
  }
}
```

Tests

```
  @Test
  public void test_right_products() {
    assertTrue(NumberUtils.validProducts(6, 20) == 3);
    assertTrue(NumberUtils.validProducts(1000, 1130) == 2);
  }

  @Test
  public void test_wrong_products() {
    assertFalse(NumberUtils.validProducts(21, 29) == 1);
  }
```

3.10 Assemble Parts in Minimum Time

Write a method to calculate the minimum possible time to put the N parts together and build the final product. The input consists of two arguments: *numOfParts*, an integer representing the number of the parts, and *parts*, a list of integers representing the size of the parts.

Example: numOfParts=4, parts=[8,4,6,12], Output: 58

Explanation:

Step1: Assemble the parts of sizes 4 and 6 (time required is **10**). Then, the size of the remaining parts after merging: [8,10,12].

Step 2: Assemble the parts of sizes 8 and 10 (time required is **18**). Then, the size of the remaining parts after merging: [18,12].

Step 3: Assemble the parts of sizes 18 and 12 (time required is **30**).

The total time required to assemble the parts is 10+18+30=58.

Solution

We order the parts, then calculates the time between the two consecutive parts of small sizes until we arrive at the last part. Listing 3.10 shows the algorithm.

```
Listing 3.10 – Assemble Parts in Minimum Time
```

```
public class AssembleParts {
  public static int minimumTime(int numOfParts, List<Integer> list) {
    int[] arrayOfSizes = list.stream().mapToInt(i -> i).toArray();
    Arrays.sort(arrayOfSizes);

    int accumulatedTime = 0;
    for (int idx = 0; idx < arrayOfSizes.length - 1; idx++) {
      accumulatedTime += (arrayOfSizes[idx] + arrayOfSizes[idx + 1]);
      //once assembled, we carry the current time to the next element
      //so in the next iteration, the first of the next two parts
      //it will already include the total time required
      //to assemble the two previous parts
      arrayOfSizes[idx + 1] = arrayOfSizes[idx] + arrayOfSizes[idx + 1];
    }
    return accumulatedTime;
  }
}
```

Tests

```
@Test
public void test_right_values() {
  assertTrue(AssembleParts.minimumTime(4,
          new ArrayList<Integer>(Arrays.asList(8, 4, 6, 12))) == 58);
  assertTrue(AssembleParts.minimumTime(5,
          new ArrayList<Integer>(Arrays.asList(3, 7, 2, 10, 5))) == 59);
}
```

Recursion

4.0 Fundamentals

A method or function that calls itself is called recursion. A recursive function 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 sometimes we don't need to use recursion in the Production environment; for example, when we pass a big integer, they can overflow the Stack and crash any application. But in other cases, we can efficiently solve problems.

4.1 Calculate Factorial of a Given Integer N

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 define the base case: returns 1 when N <= 1 to stop the recursion
- We use a recursive formula: N * factorial(N - 1)

```
Listing 4.1 – Calculate Factorial of a Given Integer N
```

```
public class FactorialRecursive {
  public static int factorial(int N) {
    //base case
    if (N <= 1)
      return 1;
    else
      //recursive call
      return (N * factorial(N - 1));
  }
}
```

The following figure shows in the first half how a succession of recursive calls executes until factorial(1) - the base case - returns 1, which stops the recursion. The second half shows the values calculated and returned from each recursive call to its caller.

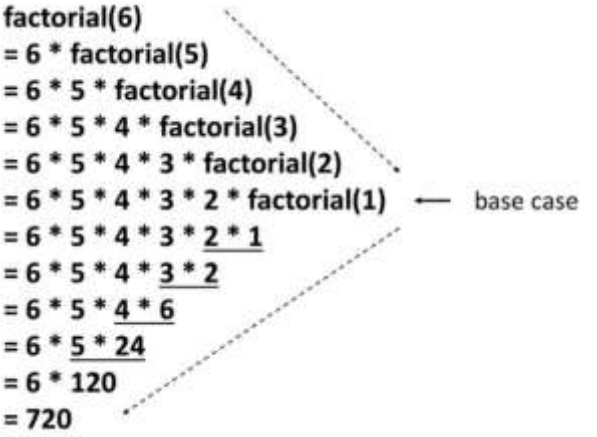


Figure 4.1 N Factorial

The following Listing shows the iterative version (and more efficient).

```
public class FactorialIterative {
  public static int factorial(int N) {
    int result = 1;
    for (int i = 2; i <= N; i++)
      result *= i;
    return result;
  }
}
```

Tests

```
@Test
public void test_right_results() {
  assertTrue(FactorialRecursive.factorial(1) == 1);
  assertTrue(FactorialRecursive.factorial(0) == 1);
  assertTrue(FactorialRecursive.factorial(3) == 6);
  assertTrue(FactorialRecursive.factorial(6) == 720);
}

@Test
public void test_wrong_results() {
  assertFalse(FactorialRecursive.factorial(3) == 5);
  assertFalse(FactorialRecursive.factorial(4) == 10);
}
```

4.2 Calculate Fibonacci Series

A Fibonacci Series is a series of numbers in which each subsequent Fibonacci number is the sum of the previous two numbers.

0, 1, 1, 2, 3, 5, 8, 13, 21, …

The first two numbers of the Fibonacci series are 0 and 1.

Examples:

Fibonacci of 7 is: 13

Fibonacci of 10 is: 55

Defines a method to calculate the Fibonacci Series using Recursion.

Solution

We define two base cases:

fibonacci(0) = 0 and fibonacci(1) = 1

We calculate the *ith* Fibonacci number recursively:

fibonacci(N) = fibonacci(N-1) + fibonacci(N-2)

Listing 4.2 – Calculate Fibonacci Series

```
public class MathUtils {
  public static int fibonacci(int N) {
    if (N == 0)
      return 0;

    if (N == 1)
      return 1;

    return fibonacci(N-1) + fibonacci(N-2);
  }
}
```

Tests

```
public class FibonacciTest {
  @Test
  public void test_fibonacciSeries() {
    assertTrue(MathUtils.fibonacci(3) == 2);
    assertTrue(MathUtils.fibonacci(7) == 13);
    assertTrue(MathUtils.fibonacci(10) == 55);
  }
}
```

Sorting and Searching

5.0 Fundamentals

Searching refers to finding an item in a collection that meets some specified criterion.

Sorting refers to rearranging all the items in a collection into increasing or decreasing order.

Sorting algorithms are essential to improve the efficiency of other algorithms that require input data previously sorted.

Applications of sorting:

- Searching - Binary search algorithm needs sorted lists to run in an O(logN) performance.
- Element uniqueness - An algorithm would sort the numbers in a list and check adjacent pairs to detect duplicate items.
- Frequency distribution - An algorithm would sort a list of items and count from left to right.
- Merge lists - An algorithm would compare elements from both sorted lists and add the smaller ones to every iteration's new merged list.

5.1 Bubble Sort

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

Solution

The algorithm uses a Boolean variable to track 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 5.1 shows the algorithm.

```
Listing 5.1 – Bubble Sort
```

```
public class Sorting {
  public int[] bubbleSort(int[] numbers) {
    if (numbers == null)
      throw new RuntimeException("array not initialized");

    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
NumbersSwapped=true

Second pass-through:
{4, 6, 5, 9} -> {4, 6, 5, 9}
{4, 6, 5, 9} -> {4, 5, 6, 9} swap because of 6 > 5
{4, 5, 6, 9} -> {4, 5, 6, 9}
NumbersSwapped=true

Third pass-through:
{4, 5, 6, 9} -> {4, 5, 6, 9}
{4, 5, 6, 9} -> {4, 5, 6, 9}
{4, 5, 6, 9} -> {4, 5, 6, 9}
NumbersSwapped=false
```

Efficiency of bubble sort

In a worst-case scenario, where the Array comes in descending order, we need a swap for each comparison. In our algorithm, a comparison happens when we compare adjacent pairs of elements to determine which one is greater.

Given an array: {9, 6, 5, 4}

In the first pass through, we have to make three comparisons -> {6, 5, 4, 9}

In our second passthrough, we have to make only two comparisons because we didn't need to compare the final two numbers -> {5, 4, 6, 9}

In our third pass through, we made just one comparison -> {4, 5, 6, 9}

In summarize:

3 + 2 + 1 = 6 comparisons, or (N-1) + (N-2) + (N-3) … + 1 comparisons

Moreover, in this scenario, we need a swap for each comparison. Therefore, we have six comparisons and six swaps = 12, which is approximately 4^2. As the number of items N increases, the number of steps exponentially grows, as shown in the following table.

N	**# steps**	**N^2**
4	12	16
5	20	25
10	90	100

Therefore, in Big O Notation, we could say that the Bubble Sort algorithm has O(N2) efficiency.

Tests

```
@Test
public void sortingArrays() {
  final int[] numbers = {6, 4, 9, 5};
  final int[] expected = {4, 5, 6, 9};
  int[] numbersSorted = sorting.bubbleSort(numbers);
  assertArrayEquals(expected, numbersSorted);
}

@Test
public void sortManyElementArray() {
  final int[] array = {7, 9, 1, 4, 9, 12, 4, 13, 9};
  final int[] expected = {1, 4, 4, 7, 9, 9, 9, 12, 13};
  sorting.bubbleSort(array);
  assertArrayEquals(expected, array);
}
```

5.2 Insertion Sort

Insertion sort is a sorting algorithm that builds the final sorted array one element at a time. It's similar to the way we sort playing cards in our hands.

Solution

- Iterates over all the elements and start at index i=1.
- Compare the current element (key) with all its preceding elements.
- If the key element is smaller than its predecessors, swap them. Move elements that are greater than the key, one position ahead of their current position.

This algorithm takes a quadratic running time $O(n^2)$. Listing 5.2 shows the algorithm.

Listing 5.2 – Insertion Sort

```
public class Sorting {
  public int[] insertSort(int[] numbers) {
    if (numbers == null)
      throw new RuntimeException("numbers not initialized");

    for (int i = 1; i < numbers.length; i++) {
      int key = numbers[i];
      int j = i - 1;
      while (j >= 0 && numbers[j] > key) {
        numbers[j + 1] = numbers[j];
        j = j - 1;
      }
      numbers[j + 1] = key;
    }
    return numbers;
  }
}
```

Example:

numbers = {13, 12, 14, 6, 7}

When i = 1. Since 13 is greater than 12, move 13 and insert 12 before 13

12, 13, 14, 6, 7

When i = 2., 14 will remain at its position as all previous elements are smaller than 14

12, 13, 14, 6, 7

When i = 3., 6 will move to the beginning, and all other elements from 12 to 14 will move

one position ahead of their current position.

6, 12, 13, 14, 7

When i = 4., 7 will move to the position after 6, and elements from 12 to 14 will move one position ahead of their current position.

6, 7, 12, 13, 14

Tests

```
@Test
public void test_insertionSort() {
  final int[] numbers = {13, 12, 14, 6, 7};
  final int[] expected = {6, 7, 12, 13, 14};
  sorting.insertSort(numbers);
  assertArrayEquals(expected, numbers);
}

@Test
public void sortingArray() {
  final int[] numbers = {7, 9, 1, 4, 9, 12, 4, 13, -2, 9};
  final int[] expected = {-2, 1, 4, 4, 7, 9, 9, 9, 12, 13};
  sorting.insertSort(numbers);
  assertArrayEquals(expected, numbers);
}
```

5.3 Quick Sort

QuickSort is a divide and conquer algorithm. Given an array, it picks an element as a pivot and partitions the given array around the chosen pivot.

Solution

Given an array, we execute the following **in-place** steps:

1. We pick from the Array, the last element as our pivot.
2. We execute a partition operation, where we put all smaller elements before the pivot and put all greater elements after the pivot. After this reorder of elements, the pivot is in its final and correct position.
3. Apply the above steps recursively to every sub-array of elements.

Listing 5.3 – Quick Sort

```
public class Sorting {
  public int[] quickSort(int[] numbers, int lo, int hi) {
    if (lo < hi) {
      int partition_border = partition(numbers, lo, hi);
      //sort elements recursively
      quickSort(numbers, lo, partition_border - 1);
      quickSort(numbers, partition_border + 1, hi);
    }
    return numbers;
  }

  private int partition(int[] numbers, int lo, int hi) {
    //element to be placed at right position
    int pivot = numbers[hi];
    int i = lo - 1; //index of smaller element
    for (int j = lo; j < hi; j++) {
      //swap when element smaller than the pivot
      if (numbers[j] < pivot) {
        i++;
        int aux = numbers[i];
        numbers[i] = numbers[j];
        numbers[j] = aux;
      }
    }
    numbers[hi] = numbers[i + 1];
    numbers[i + 1] = pivot;
    return i + 1;
  }
}
```

Tests

```
@Test
public void test_quickSort() {
  final int[] numbers = {13, 12, 14, 6, 7};
  final int[] expected = {6, 7, 12, 13, 14};
  sorting.quickSort(numbers, 0, numbers.length - 1);
  assertArrayEquals(expected, numbers);
}

@Test
public void sortingArray() {
  final int[] numbers = {7, 9, 1, 4, 9, 12, 4, 13, -2, 9};
  final int[] expected = {-2, 1, 4, 4, 7, 9, 9, 9, 12, 13};
  sorting.quickSort(numbers, 0, numbers.length - 1);
  assertArrayEquals(expected, numbers);
}
```

5.4 Binary Search

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

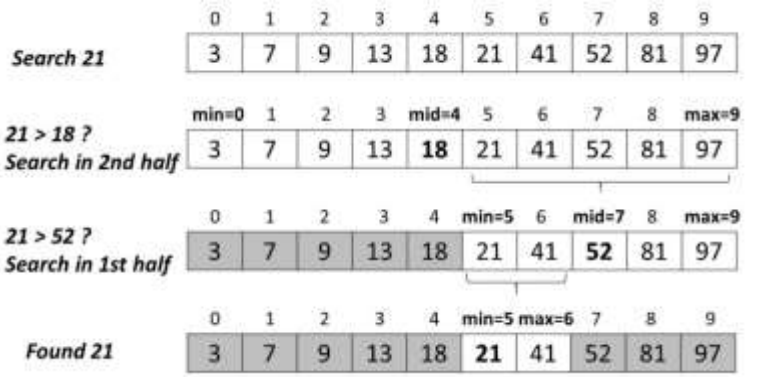


Figure 5.4 Binary Search example

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 5.4 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 5.4 Binary Search

```
public class BinarySearch {
  public static <T extends Comparable<T>> boolean search(T target, T[] array) {
    if (array == null || array.length <= 0)
      return false;

    int min = 0;
    int max = array.length - 1;
    while (min <= max) {
      int mid = (min + max) / 2;
      if (target.compareTo(array[mid]) < 0) {
        max = mid - 1;
      } else if (target.compareTo(array[mid]) > 0) {
        min = mid + 1;
      } else {
        return true;
      }
    }
    return false;
  }
}
```

Tests

```
@Test
public void binarySearch_target_notFound() {
  assertFalse(BinarySearch.search("fin", new String[]{"ada", "fda"}));
  assertFalse(BinarySearch.search("eda",
          new String[]{"ada", "bda", "cda", "dda"}));
}

@Test
public void binarySearch_target_Found() {
  assertTrue(BinarySearch.search("cal",
          new String[]{"ada", "cal", "fda"}));
  assertTrue(BinarySearch.search(21,
          new Integer[]{1, 2, 3, 4, 5, 21}));
  assertTrue(BinarySearch.search(21,
          new Integer[]{3, 7, 9, 13, 18, 21, 41, 52, 81, 97}));
}
```

5.5 Merge Two Sorted Lists

Given two sorted lists, merge them in a new sorted list.

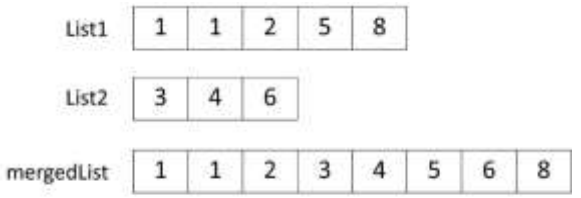


Figure 5.5 Merge Two Sorted Lists

Solution

We can join the two lists into a new list and apply a sort algorithm such as bubble sort, insertion, or quicksort. What we are going to do is implement a new algorithm maintaining the same NlogN performance.

- We define a new List to add all elements from the other two lists in a sorted way.
- We define two indexes that point to every element in every list
- We iterate both lists while still exist elements in both lists
- We compare elements from both lists and add the smaller one to the new list in every iteration. Before passing to the next iteration, we increment in one the index of the list, which contains the smaller element.
- If there is a list that still contains elements, we add them directly to the new list.

Listing 5.5 Merge Two Sorted Lists.

```
public class SortedList {
  public static List<Integer> merge_sorted(
    List<Integer> sList1, List<Integer> sList2) {
    List<Integer> mergedSortedList = new ArrayList<>();
    int idx1 = 0;
    int idx2 = 0;

    while (idx1 < sList1.size() && idx2 < sList2.size()) {
      if (sList1.get(idx1) <= sList2.get(idx2)) {
        mergedSortedList.add(sList1.get(idx1));
        idx1++;
      } else {
        mergedSortedList.add(sList2.get(idx2));
        idx2++;
      }
    }

    if (idx1 < sList1.size())
      for (int idx = idx1; idx < sList1.size(); idx++)
        mergedSortedList.add(sList1.get(idx));

    if (idx2 < sList2.size())
      for (int idx = idx2; idx < sList2.size(); idx++)
        mergedSortedList.add(sList2.get(idx));

    return mergedSortedList;
  }
}
```

Tests

```
@Test
public void mergeSortedLists() {
  List<Integer> sList1 = Arrays.asList(1,1,2,5,8);
  List<Integer> sList2 = Arrays.asList(3,4,6);
  assertEquals("[1, 1, 2, 3, 4, 5, 6, 8]",
          SortedList.merge_sorted(sList1,sList2).toString());
}

@Test
public void mergeSortedLists2() {
  List<Integer> sList1 = Arrays.asList(2,4,5);
  List<Integer> sList2 = Arrays.asList(1,3,6);
  assertEquals("[1, 2, 3, 4, 5, 6]",
          SortedList.merge_sorted(sList1,sList2).toString());
}
```

Stacks and Queues

6.0 Fundamentals

Stack

A Stack is an abstract data type, which includes a collection of objects that follow the last-in, first-out (LIFO) principle, i.e., the element inserted at last is the first element to come out of the list. A real-world example of a stack is the Stack of trays at a cafeteria.

Stacks have the following constraints:

- A *push* operation allows inserting data at the end of a stack.
- A *pop* operation allows deleting data from the end of a stack.
- It can read only the last element of a stack, which is called the top. The *peek* operation returns the value stored at the top of a stack without removing it from the stack.

You can implement a Stack using an Array or a Singly Linked List.

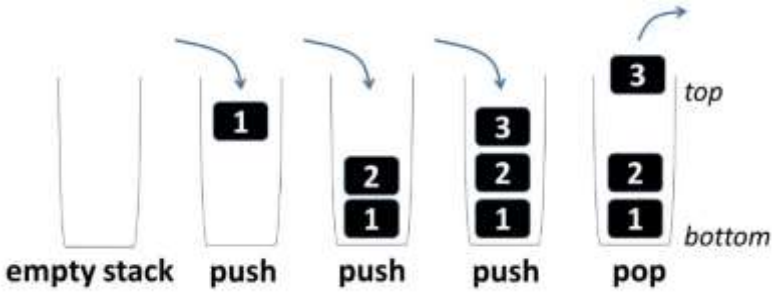


Figure 6.0.1 Stack

Stacks are helpful to handle temporary data as part of various algorithms, for example:

- Undo mechanism on text editor is based on Stack, i.e., cancel recent edit operations. The editor keeps text changes in a Stack.
- Web browsers store URLs of visited websites on a stack. When a user visits a new website, its URL is "pushed" on the Stack. When the user presses the "back" button, the previous URL is "pop" from the Stack.

Queue

A Queue is an abstract data type, which includes a collection of objects that follow the first-in, first-out (FIFO) principle, i.e., the element inserted at first is the first element to come out of the list.

Queues have the following constraints:

- An *enqueue* operation allows inserting data at the end of a Queue.
- A *dequeue* operation allows deleting data from the front of a Queue.
- It can read only the element at the front of a Queue, called a front.

Figure 6.0.2 Queue

Queues are helpful to handle waiting times, for example:

- Call centers, theaters, and other similar services process customer requests using the FIFO principle.

6.1 Delimiter Matching

Check if the parentheses, braces, and brackets in an expression are balanced. In doing so, we must ensure that:

- Each opening symbol on the left delimiter matches a closing symbol on the right delimiter.
- Left delimiters that occur later should be closed before those occurring earlier.

Solution

We use a *stack* data structure to ensure two delimiting symbols match up correctly (right pair). Why Stack? Because insertion and deletion of items take place at one end called the top of the Stack. The JDK includes the *Java.util.Stack* data structure. The *push* method adds an item to the top of this Stack. The *pop* method removes the item at the top of this Stack.

- Iterate every character from the given expression. If it is an opening symbol, push that symbol onto the Stack. If it is a closing symbol, pop an element from the Stack (the last opening symbol added) and check that they are the right pair.

- If we reach the end and the **Stack is empty**, then the expression is balanced; otherwise, it is not balanced.

Listing 6.1 – Delimiter Matching

```
  private static boolean isBalanced(String expression) {
    if (expression == null || expression.trim().length() < 2)
      return false;

    Stack<Character> stack = new Stack<>();
    for (char c : expression.toCharArray()) {
      if (c == '{' || c == '(' || c == '[') {
        stack.push(c);
      } else if (c == '}') {
        if (stack.empty() || stack.pop() != '{') {
          return false;
        }
      } else if (c == ')') {
        if (stack.empty() || stack.pop() != '(') {
          return false;
        }
      } else if (c == ']') {
        if (stack.empty() || stack.pop() != '[') {
          return false;
        }
      }
    }
    return stack.empty() ? true : false;
  }
```

Tests

```
  @Test
  public void incorrect_expressions() {
    assertFalse(delimiterMatching.apply(null));
    assertFalse(delimiterMatching.apply(""));
    assertFalse(delimiterMatching.apply("("));
    assertFalse(delimiterMatching.apply("(()a]"));
  }

  @Test
  public void correct_expressions() {
    assertTrue(delimiterMatching.apply("()"));
    assertTrue(delimiterMatching.apply("([])"));
    assertTrue(delimiterMatching.apply("{{([])}}"));
    assertTrue(delimiterMatching.apply("{{a([b])}c}dd"));
    assertTrue(delimiterMatching.apply("(w*(x+y)/z-(p/(r-q)))"));
  }
```

6.2 Queue via Stacks

Build a queue data structure using only two internal stacks.

Solution

Stacks and Queues are abstract in their definition. That means, for example, in the case of Queues, 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*.

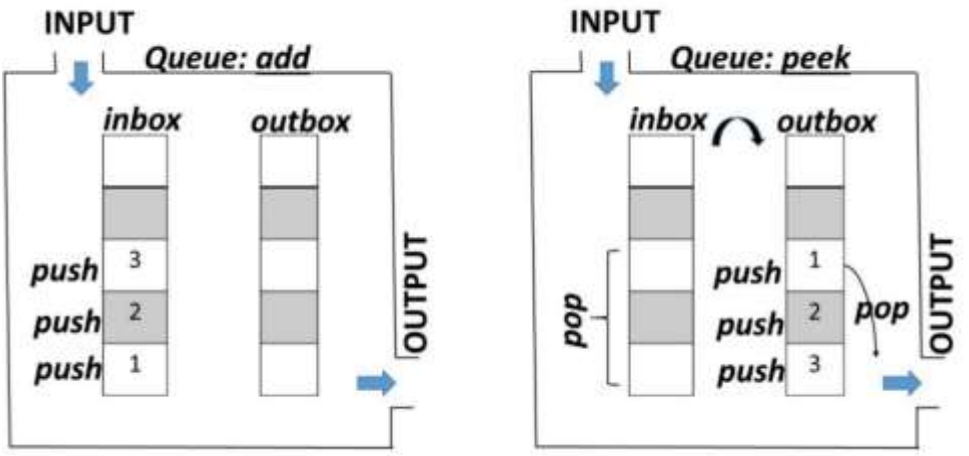


Figure 6.2 Queue via Stacks

Listing 6.2 – Queue via Stacks

```
import java.util.Stack;
public class QueueViaStacks<T> {

  Stack<T> inbox;
  Stack<T> outbox;

  public QueueViaStacks() {
    inbox = new Stack<>();
    outbox = new Stack<>();
  }
```

```
  public void add(T value) {
    //This stack always has the newest elements on top
    inbox.push(value);
  }

  public T peek() {
    if (outbox.isEmpty()) {
      while (!inbox.isEmpty()) {
        //Filled in inverse order
        outbox.push(inbox.pop());
      }
    }
    return outbox.pop();
  }
}
```

Tests

```
public class QueueViaStacksTest {

  QueueViaStacks<Integer> queueViaStacks;

  @Before
  public void Before() {
    queueViaStacks = new QueueViaStacks<>();
  }

  @Test
  public void pop_firstElement() {
    queueViaStacks.add(4);
    queueViaStacks.add(2);
    queueViaStacks.add(9);
    assertEquals(new Integer(4), queueViaStacks.peek());
  }
}
```

6.3 Reverse a String using a Stack

Given a string of characters, reverse the order of the characters using a Stack.

Solution

- Convert the text to an array of characters.
- Create an empty stack.
- Push all characters into the stack, one by one.

- Pull all characters from the stack, and put them into the array of characters.
- Finally, return the array of characters as a String

Listing 6.3 – Reverse a String using a Stack

```
import java.util.Stack;
public class StringUtils {
  public static String reverse(String text) {
    char[] charsArray = text.toCharArray();
    Stack<Character> stack = new Stack<Character>();

    for (int i=0; i<charsArray.length; i++)
      stack.push(charsArray[i]);

    for (int i=0; i<charsArray.length; i++)
      charsArray[i] = stack.pop();

    return String.valueOf(charsArray);
  }
}
```

Tests

```
@Test
public void reverseText_success() {
  assertEquals("efac", StringUtils.reverse("cafe"));
  assertEquals("2312cba", StringUtils.reverse("abc2132"));
}
```

Hash Table

7.0 Fundamentals

A hash table is a data structure that is very fast for insertion and searching. In Big O Notation takes a constant time: O(1).

A hash table is built using an array of paired values. Each pair is comprised of a key and a value, which are paired to indicate some significant association, for example:

```
students = { "1573297" => "Jan",
             "8174321" => "Brian",
             "8119214" => "James" }
```

In this example, the 1573297 is the key, and Jan is the value. They indicate that Passport Number 1573297 identifies Jan.

For a seven-digit numeric range, we need an array's size of 10,000,000. If we decide to store only a few thousand passport numbers, *the array will be almost empty*.

We need to squeeze our input key range into the array index range. To locate `1573297` in a fast way, we use a hash function (*hashing*) to convert `1573297` (*key*) to an index number within the size of our Array. Its value is stored at the index of the key after the key has been hashed.

For example, the following figure shows our hash table after using a hash function.

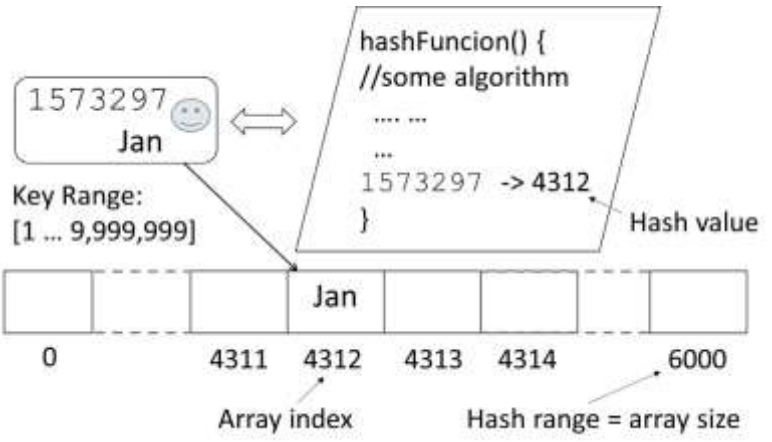


Figure 7.0.1 Hash table

Now, our hash table looks like this:

```
students = { 4312 => "Jan",
             5102 => "Brian",
             5303 => "James" }
```

An algorithm hashes the key, turns it into an array index number, and jumps directly to the index with that number to get the associated value.

But sometimes, there is a risk that different keys map to the same hashed array index. That is called a collision; we obtain a key hash to an already filled position.

There are two solutions to deal with collisions: Chaining, where each entry in the hash array points to its own linked list. The other one is open addressing, where only one array stores all items; when we want to insert a new hashed key, and the slot is already filled, then the algorithm looks for another empty slot to insert the new item. This kind of search is called the probe sequence.

Most of the programming languages already implement a strategy for dealing with collisions and choosing a hash function.

7.1 Design a Hash Table

Design a hash table, which implements add and get operations, key-value pairs should be generic, and use chaining technique for solving index collisions.

Solution

Imagine that we want to identify warehouses as keys composed of 3 characters.

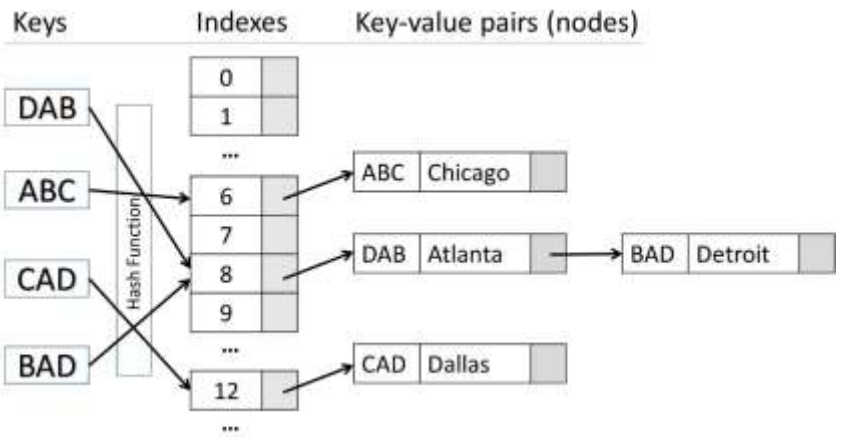


Figure 7.1 Design a Hash Table

```
warehouses = {"DAB", "Atlanta",
              "ABC", "Chicago",
              "CAD", "Dallas",
              "BAD", "Detroit"}
```

We realize that exists a collision for the keys "DAB" and "BAD".

To implement chaining, we need to create a hash table *entry* that behaves as a linked list. And we define an array that holds all entries.

Listing 7.1 – Design a Hash Table

```
public class HashTable<K,V> {

  private static final int SIZE = 121;
  private Entry[] entries = new Entry[SIZE];

  private static class Entry<K,V> {
    K key;
    V value;
    Entry<K,V> next;

    Entry(K key, V value) {
      this.key = key;
      this.value = value;
      this.next = null;
    }
  }
}
```

We need to define our hash function. We will do it easily for testing purposes. We map letters to numbers. (A = 1, B = 2, C = 3, D = 4, E = 5). Then take the product of the digits.

Example:

Key: ”DAB”

Mapping: D = 4, A = 1, B = 2

Product:

4 * 1 * 2 = 8

First, we define our assumption.

```
@Test
public void hashTest() {
  assertEquals(8, hashTable.hashTheKey("DAB"));
  assertEquals(15, hashTable.hashTheKey("ACE"));
}
```

Here, the implementation of the hash function.

```
public int hashTheKey(K key) {
  int product = 1;
  for (char c : ((String)key).toCharArray()) {
    if (c == 'A') {
      product *=1;
    } else if (c == 'B') {
      product *=2;
    } else if (c == 'C') {
      product *=3;
    } else if (c == 'D') {
      product *=4;
    } else if (c == 'E') {
      product *=5;
    }
  }
  return product;
}
```

The *put* method creates a new hashed index by calling the *hashTheKey* method and creates a new entry in the linked list at the hashed index. If the entry was already filled, then create the entry at the end of the linked list.

Before implementing *put* and *get* methods, we define our assumption.

```
@Test
public void getTest() {
  hashTable.put("DAB", "Atlanta");
  hashTable.put("ABC", "Chicago");
  hashTable.put("CAD", "Dallas");
  hashTable.put("BAD", "Detroit");
  assertEquals("Atlanta", hashTable.get("DAB"));
}
```

And here, the implementation.

```
public void put(K key, V value) {
  int hash = hashTheKey(key);
  Entry newEntry = new Entry(key, value);
  if (entries[hash] == null) {
    entries[hash] = newEntry;
  } else { //collision, chaining,
    //insert as the last node at the entry linked list
    Entry currentEntry = entries[hash];
    while (currentEntry.next != null) {
      currentEntry = currentEntry.next;
    }
    currentEntry.next = newEntry;
  }
}
```

The *get* method hash the key again by calling the *hashTheKey* method, iterates the linked list of entries looking for the key, and returns the value.

And here, the implementation.

```
public V get(K key) {
  int hash = hashTheKey(key);
  if (entries[hash] != null) {
    Entry currentEntry = entries[hash];
    while (currentEntry != null) {
      if (currentEntry.key.equals(key)) {
        return (V)currentEntry.value;
      }
        currentEntry = currentEntry.next;
    }


  }
  return null;
}
```

7.2 Find the Most Frequent Elements in an Array

Given an array, find the most frequent elements in the array.

Solution

Firstly, create a class that implements the Map interface and which permits null values.

Secondly, iterate the input array and store elements and their frequency as key-value pairs. Then, traverse the Map and inverse the order with the maximum frequency on top.

Finally, traverse the previous Map, and build a list of the elements with maximum frequencies. Listing 7.2 shows the algorithm.

```
Listing 7.2 – Return the most frequent elements of an array
```

```
import static java.util.stream.Collectors.*;
public class ArrayUtils {

  public static int[] mostFrecuent(int[] array) {

    Map<Integer, Integer> mapFrecuencyByElement = new HashMap<>();
    for (int element : array) {
      Integer frecuency = mapFrecuencyByElement.get(element);
      mapFrecuencyByElement.put(element, frecuency == null ? 1 : frecuency + 1);
    }
```

```
  Map<Integer, Integer> mapOrderedByTopFrecuency = mapFrecuencyByElement
            .entrySet()
            .stream()
            .sorted(Collections.reverseOrder(Map.Entry.comparingByValue()))
            .collect(
                    toMap(Map.Entry::getKey, Map.Entry::getValue, (e1, e2) -> e2,
                            LinkedHashMap::new)
            );

    List<Integer> result = new ArrayList<>();
    int idx = 0; //to iterate the result list
    for (Map.Entry<Integer, Integer> entry : mapOrderedByTopFrecuency.entrySet()) {
      if (idx > 0) {
        //continue only if exists same frecuencies
        if (entry.getValue().equals(result.get(idx - 1))) {
          result.add(entry.getKey());
        } else {
          break;
        }
      } else {
        result.add(entry.getKey());
      }
      idx++;
    }

    int[] mostFrecuent = result.stream().mapToInt(i -> i).toArray();
    return mostFrecuent;
  }
}
```

Tests

```
@Test
public void test_right_values() {
    assertArrayEquals(new int[]{2, 3},
            ArrayUtils.mostFrecuent(new int[]{3, 2, 0, 3, 1, 2}));
    assertArrayEquals(new int[]{5},
            ArrayUtils.mostFrecuent(new int[]{3, 5, 0, 5, 5, 1, 2}));
    assertArrayEquals(new int[]{7},
            ArrayUtils.mostFrecuent(new int[]{7}));
}
```

7.3 Nuts and Bolts

Given n nuts and n bolts of different sizes, and consider a one-to-one mapping between them. Write a method that finds all matches between the nuts and bolts efficiently.

nuts = {'$', '%', '&', 'x', '@'};

bolts = {'%', '@', 'x', '$', '&'};

Solution

- The method receives two characters array, nuts[] and bolts[].
- Create a HashMap and put each nut character as a key.
- Iterate the bolts array and check if the HashMap already contains each bolt as a key.
- If a bolt at i location in the bolts array is already included in the HashMap, replace the nuts[i] with the content from the bolts[i] to indicate we found a match.

Listing 7.3 – Nuts and Bolts

```
public class NutsAndBolts {
  public static void match(char[] nuts, char[] bolts) {
    Map<Character, Integer> map = new HashMap<>();
    for (int i=0; i<nuts.length; i++) {
      map.put(nuts[i],i);
    }

    for (int i=0; i<bolts.length; i++) {
      if (map.containsKey(bolts[i])) {
        nuts[i] = bolts[i];
      }
    }
  }
}
```

Tests

```
@Test
public void matchNultsAndBoltsTest() {
  char nuts[] = {'$', '%', '&', 'x', '@'};
  char bolts[] = {'%', '@', 'x', '$', '&'};
  NutsAndBolts.match(nuts, bolts);
  assertArrayEquals(nuts, bolts);
}
```

Trees

8.0 Fundamentals

Tree

A tree is a data structure that consists of nodes connected by edges.

Tree structures are non-linear data structures. They allow us to implement algorithms much faster than when using linear data structures.

Binary Tree

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 collected in the node. The following figure shows the binary tree terminology

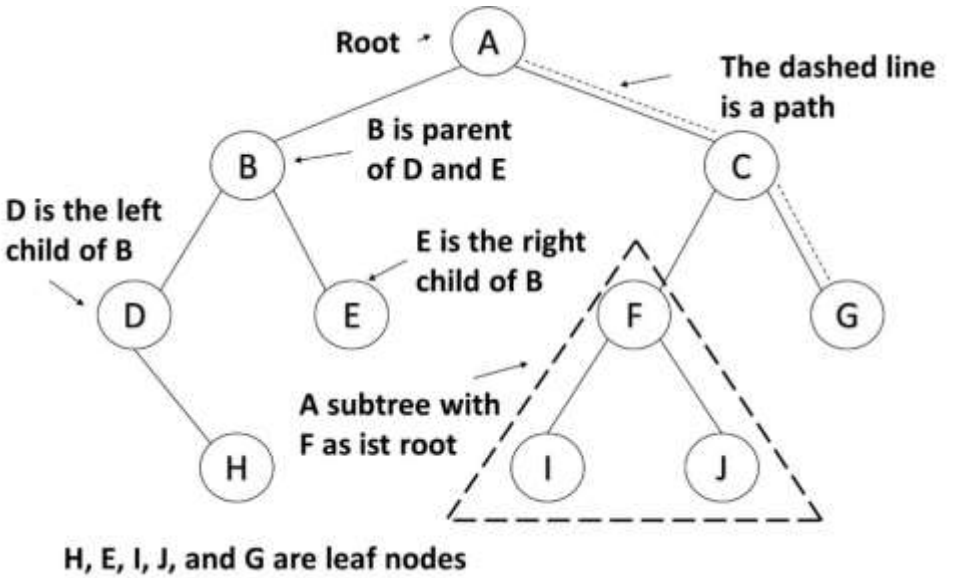


Figure 8.0 Binary Search Tree - terminology

Binary Search Tree

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.

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

It takes a maximum of log2(N) attempts to find a value. As the collection of nodes gets large, the binary search tree becomes faster over a linear search which takes up to (N) comparisons.

8.1 Binary Search Tree

A company uses the Global Trade Item Number (GTIN) to uniquely identify all of its trade items. The GTIN identifies the types of products that different manufacturers produce.

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

Solution

We define a Product Class which will be the data contained in a Node.

Listing 8.1.1 shows how we create a Product Class.

```
Listing 8.1.1 – Product Class
```

```
public class Product {
  Integer productId;
  String name;
  Double price;
  String manufacturerName;
  //setters and getters are omitted
}
```

Listing 8.1.2 shows how we create a NodeP Class to store a list of Products. Moreover, this Class allows us to have two NodeP attributes to hold the left and right nodes.

```
Listing 8.1.2 – NodeP Class
```

```
public class NodeP {
  private String gtin;
  private List<Product> data;
  private NodeP left;
  private NodeP right;
  public NodeP(String gtin, List<Product> data) {
    this.gtin = gtin;
    this.data = data;
  }
}
```

Listing 8.1.3 shows a TreeP Class to implement an abstract data type called binary search tree, which includes a NodeP root variable for the first element to be inserted. We need to implement an insert method, where every time a new GTIN is inserted, it compares the current GTIN versus the new GTIN. Depending on the result, we store the new GTIN on the left or the right Node. In this way, the insert method maintains an ordered binary search tree.

Listing 8.1.3 TreeP and insert method

```
public class TreeP {

  private NodeP root;

  public void insert(String gtin, List<Product> data) {

    NodeP newNode = new NodeP(gtin, data);

    if (root == null)
      root = newNode;
    else {
      NodeP current = root;
      NodeP parent;
      while (true) {
        parent = current;
        if (gtin.compareTo(current.getGtin()) < 0) {
          current = current.getLeft();
          if (current == null) {
            parent.setLeft(newNode);
            return;
          }
        } else if (gtin.compareTo(current.getGtin()) > 0) {
          current = current.getRight();
          if (current == null) {
            parent.setRight(newNode);
            return;
          }
        } else
          return; //already exists
      }
    }
    return;
  }
}
```

Listing 8.1.4 shows a find method, which iterates through all nodes until a GTIN is found. This algorithm reduces the search space to N/2 because the binary search tree is always ordered.

Listing 8.1.4 – find method

```
  public NodeP find(String gtin) {

    NodeP current = root;
    if (current == null)
      return null;

    while (!current.getGtin().equals(gtin)) {
      if (gtin.compareTo(current.getGtin()) < 0) {
        current = current.getLeft();
      } else {
        current = current.getRight();
      }
      if (current == null) //not found in children
        return null;
    }
    return current;
  }
```

Tests

```
@Test
public void test_findNode() {
  tree.insert("04000345706564",
    new ArrayList<>(Arrays.asList(product1)));
  tree.insert("07611400983416",
    new ArrayList<>(Arrays.asList(product2)));
  tree.insert("07611400989104",
    new ArrayList<>(Arrays.asList(product3, product4)));
  tree.insert("07611400989111",
    new ArrayList<>(Arrays.asList(product5)));
  tree.insert("07611400990292",
    new ArrayList<>(Arrays.asList(product6, product7, product8)));
  assertEquals(null, tree.find("07611400983324"));
  tree.insert("07611400983324", new ArrayList<>(Arrays.asList(product9)));
  assertTrue(tree.find("07611400983324") != null);
  assertEquals("07611400983324", tree.find("07611400983324").getGtin());
}
```

This Binary Search Tree works well when the data is inserted in random order. But when the values to be inserted are already ordered, a binary tree becomes unbalanced. With an unbalanced tree, we cannot find data quickly.

One approach to solving unbalanced trees is the red-black tree technique, a binary search tree with some unique features.

Assuming that we already have a balanced tree, listing 8.1.5 shows us how fast in terms of comparisons could be a binary search tree, which depends on a number N of elements. For instance, to find a product by GTIN in 1 billion products, the algorithm needs only 30 comparisons.

Listing 8.1.5 - Tree performance

```
public class TreePerformance {
  public int comparisons(int N) {

    int acumElements = 0;
    int comparisons = 0;
    for (int level = 0; level <= N / 2; level++) {
      int power = (int) Math.pow(2, level);
      acumElements += power;
      if (acumElements >= N) {
        comparisons = ++level;
        break;
      }
    }
    System.out.println("comparisons -> " + comparisons);
    return comparisons;
  }
}
```

Tests

```
@Test
public void whenNelements_return_NroComparisons() {
  assertTrue(treePerformance.comparisons(15) <= 4);
  assertTrue(treePerformance.comparisons(31) <= 5);
  assertTrue(treePerformance.comparisons(1000) <= 10);
  assertTrue(treePerformance.comparisons(1000000000) <= 30);
}
```

8.2 Tree Traversal

Implement a method to visit all the nodes of a tree and print their values.

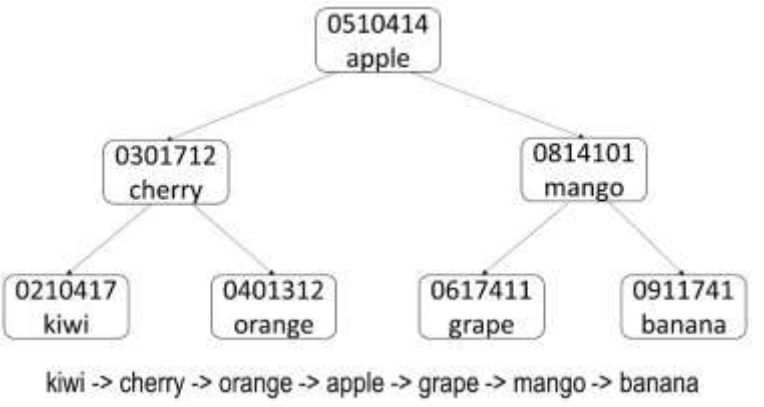


Figure 8.2 In-Order traversal

Solution

We can traverse a tree in three different ways:

- In-order Traversal: We visit the tree in this order: Left, Root, Right.
- Pre-order Traversal: We visit the tree in this order: Root, Left, Right.
- Post-order Traversal: We visit the tree in this order: Left, Right, Root.

For this solution, we traverse the tree in order traversal, where the output will produce the key values in ascending order.

We create a recursive *inOrderTraversal* method that receives the root node as a parameter and does the following instructions:

- call inOrderTraversal for the left node
- print the data
- call inOrderTraversal for the right node

We include a *StringJoiner* class in the method signature to collect the results.

```
Listing 8.2 – In-order traversal
```

```
import java.util.StringJoiner;
public class TreeADT {
  class Product {
    int productId;
    String name;
    Product(int productId, String name) {
      this.productId = productId;
      this.name = name;
    }
    public String getName() {
      return name;
    }
  }

  class NodeP {
    private String gtin;
    private Product data;
    private NodeP left;
    private NodeP right;
    public NodeP(String gtin, Product data) {
      this.gtin = gtin;
      this.data = data;
    }
    //setters and getters omitted
  }
```

```
  private NodeP root;

  public void insert(String gtin, int productId, String name) {
    Product product = new Product(productId, name);
    NodeP newNode = new NodeP(gtin, product);

    if (root == null)
      root = newNode;
    else {
      NodeP current = root;
      NodeP parent;
      while (true) {
        parent = current;
        if (gtin.compareTo(current.getGtin()) < 0) {
          current = current.getLeft();
          if (current == null) {
            parent.setLeft(newNode);
            return;
          }
        } else if (gtin.compareTo(current.getGtin()) > 0) {
          current = current.getRight();
          if (current == null) {
            parent.setRight(newNode);
            return;
          }
        } else {
          current.setData(product);
          return; //already exists
        }
      }
    }
    return;
  }


  public String inOrderTraversal(NodeP localRoot, StringJoiner stringJoiner) {
    if (localRoot != null) {
      inOrderTraversal(localRoot.getLeft(), stringJoiner);
      stringJoiner.add(localRoot.getData().getName());
      inOrderTraversal(localRoot.getRight(), stringJoiner);
    }
    return stringJoiner.toString();
  }
}
```

Tests

```
import java.util.StringJoiner;
public class TreeADTTest {
  private TreeADT tree;
```

```
  @Before
  public void setup() {
    tree = new TreeADT();
  }

  @Test
  public void test_traverseTree() {
    tree.insert("0510414", 12, "apple");
    tree.insert("0301712", 14, "cherry");
    tree.insert("0814101", 13, "mango");
    tree.insert("0401312", 19, "orange");
    tree.insert("0911741", 11, "banana");
    tree.insert("0617411", 16, "grape");
    tree.insert("0210417", 18, "kiwi");
    StringJoiner stringJoiner = new StringJoiner(" -> ", "[", "]");
    assertEquals("[kiwi -> cherry -> orange -> apple -> grape -> mango -> banana]",
            tree.inOrderTraversal(tree.getRoot(), stringJoiner));
  }
}
```

Graphs

9.0 Fundamentals

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 model the ***Hyperloop*** transport to be installed in Germany.

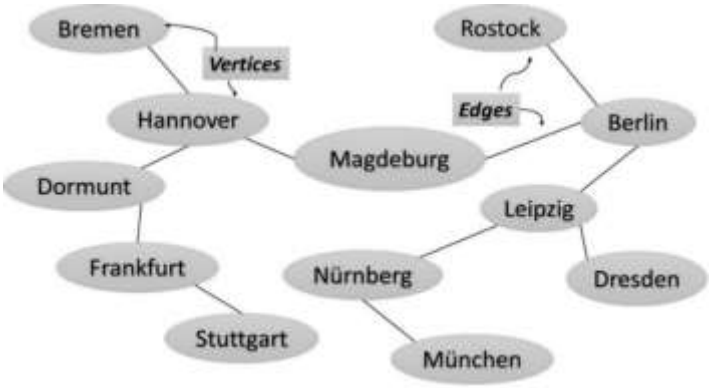


Figure 9.0 Graph - terminology

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

Adjacency

When a single edge connects two vertices, then they are adjacent or neighbors. In the figure above, the vertices represented by Berlin and Leipzig are adjacent, but the cities Berlin and Dresden are not.

Path

A Path is a sequence of edges. The figure above shows a path from Berlin to München that passes through cities Leipzig and Nürnberg. Then the path is Berlin, Leipzig, Nürnberg, München.

Connected Graphs

A Graph is *connected* if there is at least one path from every vertex to every other vertex. The figure above is connected because all cities are connected.

Directed and Weighted Graphs

A Graph is directed when the edges have a *direction*. In the figure above, we have a non-directed graph because the ***hyperloop*** can usually go either way. From Berlin to Leipzig is the same as from Leipzig to Berlin.

Graphs are called a weighted graph when edges are given weight, e.g., the distance between two cities can be weighted in how fast they are connected.

A graph can answer one of the questions: which cities can be reached from a given City? We need to implement search algorithms. There are two different ways of searching in a graph: *depth-first search (DFS)* and *breadth-first search (BFS)*.

9.1 Depth-First Search (DFS)

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

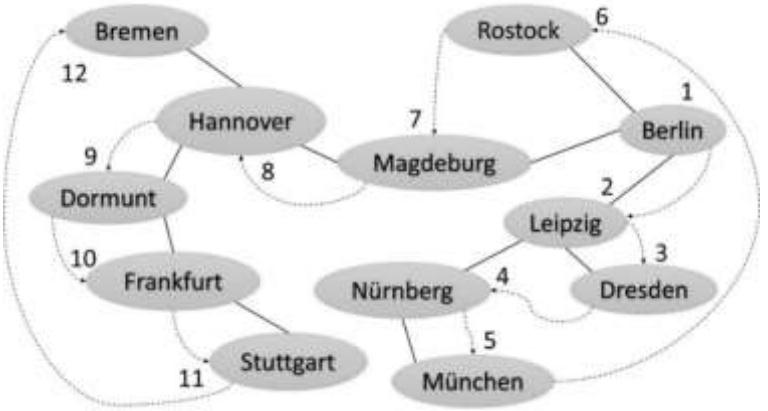


Figure 9.1.1 Depth-first search - the sequence of steps

Solution

Depth-first search (DFS) is an algorithm for traversing the Graph. The algorithm starts at the root node (selecting some arbitrary city as the root node) and explores as far as possible along each path. Figure 9.1.1 shows a sequence of steps if we choose Berlin as the root node.

Implementing the algorithm

Model the problem

We need an Object which supports any data included in the Node. We called it vertex. Inside this vertex, we define a boolean variable to avoid cycles in searching cities, so we will mark each Node when visiting it. Listing 9.1.1 shows a Vertex Class implementation.

Listing 9.1.1 – Vertex Class

```
public class Vertex {
  private String name;
  private boolean visited;

  public Vertex(String name) {
    this.name = name;
    this.visited = false;
  }
  //setters and getter omitted
}
```

We have two approaches to model how the vertices are connected (edges): the *adjacency matrix* and the *adjacency list*. For this algorithm, we are going to implement the adjacency matrix.

The adjacency matrix

In a graph of N vertices, we create a two-dimensional array of NxN. An edge between two vertices (cities) indicates a connection (two adjacent nodes) represented by 1. No connections are represented by 0.

	Berlin	Leipzig	Dresden	Rostock	Nürnberg	...
Berlin	0	1	0	1	0	
Leipzig	1	0	1	0	1	
Dresden	0	1	0	0	0	
Rostock	1	0	0	0	0	
Nürnberg	0	1	0	0	0	

Table 9.1.1 The adjacency matrix

For instance, the table above says Leipzig is adjacent to Berlin, Dresden, and Nürnberg, for example.

Create and initialize an abstract data type

We create an abstract data type called Graph to define the behavior of our data structure.

We need a stack data structure so we can remember the visited vertices. A stack follows the last-in, first-out (LIFO) principle, i.e., the city inserted at last is the first city to come out of the stack.

We define an *arrayOfVertex[]* array to store new Vertices(cities) added to the Graph.

We define a *numOfVertices* variable that indicates the number of Vertices already added to the Graph.

Since we will pass a String argument to our DFS algorithm (city name), a *mapOfVertex* HashMap is defined to register the key-value: *city-index*.

Listing 9.1.2 Graph Class

```
public class Graph {

  private final int MAX_VERTEX = 15;
  private Vertex arrayOfVertex[]; //cities
  private Map<String, Integer> mapOfVertex;
  //matrix of adjacent vertex:
  private int matrixOfAdjVertex[][];
  //register the location at the arrayOfVertex:
  private int numOfVertices;
  private Stack<Integer> stack;

  public Graph() {
    arrayOfVertex = new Vertex[MAX_VERTEX];
    mapOfVertex = new ConcurrentHashMap<>();
    numOfVertices = 0;
    matrixOfAdjVertex = new int[MAX_VERTEX][MAX_VERTEX];
    stack = new Stack<>();
    //initialize matrix
    for (int i = 0; i < MAX_VERTEX; i++) {
      for (int j = 0; j < MAX_VERTEX; j++) {
        matrixOfAdjVertex[i][j] = 0;
      }
    }
  }
}
```

We define the following methods inside the Graph class.

Adding a vertex

```
public void addVertex(Vertex city) {
  mapOfVertex.put(city.getName(), numOfVertices);
  arrayOfVertex[numOfVertices++] = city;
}
```

The *numOfVertices* variable determines the location (index) of the new City in the *arrayOfVertex*[].

Adding an edge

We add two entries to *matrixOfAdjVertex*, because the two cities are connected in both directions.

```
public void addEdge(String city1, String city2) {
  int start = mapOfVertex.get(city1);
  int end = mapOfVertex.get(city2);
  matrixOfAdjVertex[start][end] = 1;
  matrixOfAdjVertex[end][start] = 1;
}
```

The point here is that we need to define the topology of our Graph, adding Vertices(cities) and edges that connect them.

The algorithm

Our *dfs()* method receives the City name as its argument. Then we locate the index of this City in our *HashMap*, and it is marked as visited and push it onto the Stack.

We iterate the stack items *until it is empty*. And this is what we do in every iteration:

1. We retrieve the vertex from the top of the Stack (peek).
2. We try to retrieve at least one unvisited neighbor for this vertex.
3. If one vertex is found, it is marked as visited and pushes onto the Stack.
4. If one vertex is not found, we pop the Stack.

If Berlin were our entry city, then the first adjacent City will be Leipzig, marked as visited and pushed into the Stack. We read (peek) Leipzig from the Stack in the next iteration and look for its neighbors. Therefore, following these iterations, we arrive at München as the last city in this path. That is the *in-depth* essence of the algorithm. To explore as far as possible along each branch before continuing with a new one.

Listing 9.1.3 Deep-First Search algorithm

```
public void dfs(String city) {
  int vertex = mapOfVertex.get(city);
  arrayOfVertex[vertex].setVisited(true);
  System.out.print(city + " ");
  stack.push(vertex);

  while (!stack.isEmpty()) {
    int adjVertex = getAdjVertex(stack.peek());
    if (adjVertex != -1) {
      arrayOfVertex[adjVertex].setVisited(true);
      System.out.print(
              arrayOfVertex[adjVertex].getName() + " ");
      stack.push(adjVertex);
    } else {
      stack.pop();
    }
  }
}
```

```
//adjacent vertices not visited yet
private int getAdjVertex(int vertex) {
  for (int adj = 0; adj < numOfVertices; adj++) {
    if (matrixOfAdjVertex[vertex][adj] == 1 &&
            arrayOfVertex[adj].isVisited() == false)
      return adj; //return first adjacent vertex
  }
  return -1; //not vertices found
}

public Map<String, Integer> getMapOfVertex() {
  return mapOfVertex;
}

public int[][] getMatrixOfAdjVertex() {
  return matrixOfAdjVertex;
}
```

Tests

```
public class GraphTest {

  Graph graph;

  @Before
  public void setup() {
    graph = new Graph();
  }

  @Test
  public void test_addVertex() {
    Vertex city = new Vertex("Berlin");
    graph.addVertex(city);
    assertTrue(graph.getMapOfVertex().size() == 1);
  }

  @Test
  public void test_addEdge() {
    String city1 = "Berlin";
    String city2 = "Leipzig";
    Vertex v1 = new Vertex(city1);
    Vertex v2 = new Vertex(city2);
    graph.addVertex(v1); //location 0
    graph.addVertex(v2); //location 1
    graph.addEdge(city1, city2);
    assertTrue(graph.getMatrixOfAdjVertex()[0][1] == 1);
    assertTrue(graph.getMatrixOfAdjVertex()[1][0] == 1);
  }
```

```
  @Test
  public void test_dfs() {
    String city1 = "Berlin";
    String city2 = "Leipzig";
    String city3 = "Dresden";
    String city4 = "Nürnberg";
    String city5 = "Hannover";
    String city6 = "Rostock";
    String city7 = "Dortmund";
    String city8 = "Frankfurt";
    String city9 = "Stuttgart";
    String city10 = "München";
    String city11 = "Magdeburg";
    String city12 = "Bremen";
    graph.addVertex(new Vertex(city1));
    graph.addVertex(new Vertex(city2));
    graph.addVertex(new Vertex(city3));
    graph.addVertex(new Vertex(city4));
    graph.addVertex(new Vertex(city5));
    graph.addVertex(new Vertex(city6));
    graph.addVertex(new Vertex(city7));
    graph.addVertex(new Vertex(city8));
    graph.addVertex(new Vertex(city9));
    graph.addVertex(new Vertex(city10));
    graph.addVertex(new Vertex(city11));
    graph.addVertex(new Vertex(city12));
    graph.addEdge(city1, city2);
    graph.addEdge(city2, city3);
    graph.addEdge(city3, city4);
    graph.addEdge(city4, city10);
    graph.addEdge(city11, city5);
    graph.addEdge(city5, city7);
    graph.addEdge(city7, city8);
    graph.addEdge(city8, city9);
    graph.addEdge(city1, city6);
    graph.addEdge(city1, city11);
    graph.addEdge(city5, city12);
    graph.dfs(city1);
  }
}
```

```
Output:
Berlin Leipzig Dresden Nürnberg München Rostock Magdeburg Hannover Dortmund
Frankfurt Stuttgart Bremen
```

We can change the entry city (Hannover) and see different traversing paths.

```
Output:
Hannover Dortmund Frankfurt Stuttgart Magdeburg Berlin Leipzig Dresden Nürnberg
München Rostock Bremen
```

9.2 Breadth-First Search (BFS)

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

Solution

In the breadth-first search, the algorithm stays as close as possible to the starting point. It visits all the vertices adjacent to the starting vertex. The algorithm is implemented using a queue.

Figure 9.2.1 shows a sequence of steps if we choose Berlin as the root node. The numbers indicate the order in which the vertices are visited.

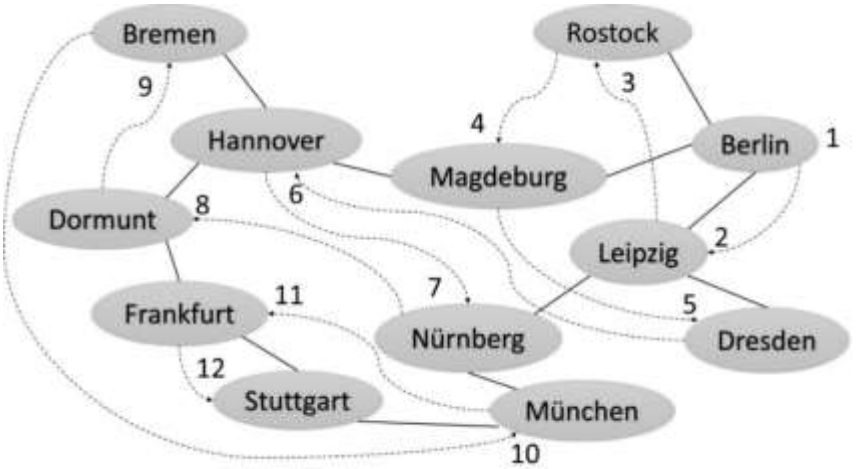


Figure 9.2.1 Breadth-First Search - the sequence of steps

The breath-first search algorithm first finds all the vertices that are one edge away from the starting vertex, then all the vertices that are two edges away, three edges away, and so on. It is useful to answer questions like what is the shortest path from Berlin to another city like München?

We traverse cities that are one edge away from Berlin (first level): Rostock, Magdeburg, and Leipzig. Then we traverse cities that are two edges away from Berlin (second level): Hannover, Nürnberg, and Dresden. Then we traverse cities that are three edges away from Berlin (third level): Bremen, Dortmund, and München. That's the idea. We already found München before traversing another possible path: Berlin, Magdeburg, Hannover, Dortmund, Frankfurt, Stuttgart, München, which corresponds to the sixth level.

Adding an edge

We use a LinkedList data structure to build our Adjacency List.

Two cities are adjacent or neighbors when a single edge connects them. Listing 9.2.1 shows the *addEdge* method.

Listing 9.2.1 - LinkedList, addEdge method

```
private LinkedList<Integer> adjList[];

public void addEdge(String city1, String city2) {
  int start = mapOfVertex.get(city1);
  int end = mapOfVertex.get(city2);
  adjList[start].add(end);
  adjList[end].add(start);
}
```

The algorithm

Our *bfs()* method receives the City name as its argument. Then we locate the index of this City in our *HashMap*, and it is marked as visited, and add it onto the queue.

We iterate the queue items *until it is empty*. And this is what we do in every iteration:

1. We retrieve and remove the head Vertex of this queue (remove).
2. We iterate in an inner loop through all neighbors (adjacent) of this head Vertex until all they were visited. Every adjacent vertex is marked as visited and added to the queue.
3. When the previous iteration cannot find more adjacent vertices, then we retrieve and remove the new head Vertex.

Listing 9.2.2 – Breath-first search algorithm

```
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;

public class Graph {

  private final int MAX_VERTEX;
  private Vertex arrayOfVertex[]; //cities
  private Map<String, Integer> mapOfVertex;
  //Adjacency list:
  private LinkedList<Integer> adjList[];
  //register the location at the arrayOfVertex:
  private int numOfVertices;
  private Queue<Integer> queue;
```

```
public Graph(int vertices) {
  MAX_VERTEX = vertices;
  arrayOfVertex = new Vertex[MAX_VERTEX];
  mapOfVertex = new ConcurrentHashMap<>();
  numOfVertices = 0;
  queue = new LinkedList<>();

  adjList = new LinkedList[MAX_VERTEX];
  for (int i = 0; i < MAX_VERTEX; i++) {
    adjList[i] = new LinkedList();
  }
}

public void addVertex(Vertex city) {
  mapOfVertex.put(city.getName(), numOfVertices);
  arrayOfVertex[numOfVertices++] = city;
}

public void addEdge(String city1, String city2) {
  int start = mapOfVertex.get(city1);
  int end = mapOfVertex.get(city2);
  adjList[start].add(end);
  adjList[end].add(start);
}

public void bfs(String city) {
  int vertex = mapOfVertex.get(city);
  arrayOfVertex[vertex].setVisited(true);
  System.out.print(city + " ");
  queue.add(vertex);

  //iterate until queue empty
  while (!queue.isEmpty()) {
    int headVertex = queue.remove();
    int adjVertex;
    //iterate until it has no unvisited cities
    while ((adjVertex = getAdjVertex(headVertex)) != -1) {
      arrayOfVertex[adjVertex].setVisited(true);
      System.out.print(arrayOfVertex[adjVertex].getName() + " ");
      queue.add(adjVertex);
    }
  }
}
public Map<String, Integer> getMapOfVertex() {
  return mapOfVertex;
}
public LinkedList<Integer>[] getAdjList() {
  return adjList;
}
```

```
  //adjacent vertices not visited yet
  private int getAdjVertex(int vertex) {
    LinkedList linkedList = adjList[vertex];
    for (int adj = 0; adj < linkedList.size(); adj++) {
      if (arrayOfVertex[(int) linkedList.get(adj)].isVisited() == false)
        return (int) linkedList.get(adj); //return first adjacent vertex
    }
    return -1; //not vertices found
  }
}
```

Tests

```
public class GraphTest {

  Graph graph;

  @Before
  public void setup() {
    graph = new Graph(15);
  }

  @Test
  public void test_addVertex() {
    Vertex city = new Vertex("Berlin");
    graph.addVertex(city);
    assertTrue(graph.getMapOfVertex().size() == 1);
  }

  @Test
  public void test_addEdge() {
    String city1 = "Berlin";
    String city2 = "Leipzig";
    Vertex v1 = new Vertex(city1);
    Vertex v2 = new Vertex(city2);
    graph.addVertex(v1); //location 0
    graph.addVertex(v2); //location 1
    graph.addEdge(city1, city2);
    assertTrue(graph.getAdjList()[0].get(0) == 1);
    assertTrue(graph.getAdjList()[1].get(0) == 0);
  }
```

```
  @Test
  public void test_bfs() {
    String city1 = "Berlin";
    String city2 = "Leipzig";
    String city3 = "Dresden";
    String city4 = "Nürnberg";
    String city5 = "Hannover";
    String city6 = "Rostock";
    String city7 = "Dortmund";
    String city8 = "Frankfurt";
    String city9 = "Stuttgart";
    String city10 = "München";
    String city11 = "Magdeburg";
    String city12 = "Bremen";
    graph.addVertex(new Vertex(city1));
    graph.addVertex(new Vertex(city2));
    graph.addVertex(new Vertex(city3));
    graph.addVertex(new Vertex(city4));
    graph.addVertex(new Vertex(city5));
    graph.addVertex(new Vertex(city6));
    graph.addVertex(new Vertex(city7));
    graph.addVertex(new Vertex(city8));
    graph.addVertex(new Vertex(city9));
    graph.addVertex(new Vertex(city10));
    graph.addVertex(new Vertex(city11));
    graph.addVertex(new Vertex(city12));
    graph.addEdge(city1, city2);
    graph.addEdge(city2, city3);
    graph.addEdge(city3, city4);
    graph.addEdge(city4, city10);
    graph.addEdge(city11, city5);
    graph.addEdge(city5, city7);
    graph.addEdge(city7, city8);
    graph.addEdge(city8, city9);
    graph.addEdge(city1, city6);
    graph.addEdge(city1, city11);
    graph.addEdge(city5, city12);
    graph.addEdge(city9, city10);
    graph.bfs(city1);
  }
}
```

```
Output:
Berlin Leipzig Rostock Magdeburg Dresden Hannover Nürnberg Dortmund Bremen
München Frankfurt Stuttgart
```

Coding Challenges

10.0 Fundamentals

There are two ways to receive a coding challenge from a recruiter. First, you receive a description of the coding challenge via email that you need to solve at home. Second, you need to solve the coding challenge in front of other developers at the recruiter's office.

For both ways, keep in mind a few things for making a good impression on the recruitment process:

- Host your final code on a website like Github with a clear README file and clear commit messages
- Include test cases for your code. It shows that you care about maintainability.
- Build a clean code structure. Your code must be readable.
- Apply SOLID principles, which tell you how to arrange your functions into classes and how those classes should be interrelated.
- Think about possible improvements – not included in the challenge specification – because these will be the open questions from recruiters when you change your code in front of them.

10.1 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 table 10.1.1

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. We choose a quantitative approach for this solution based on which product gives us *more grams per dollar invested.*

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 10.1.1 List of snacks

We use the Red-Green Refactor technique to implement our algorithm, 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 product. Our algorithm receives two parameters: an array 2-D, which includes [product-id][price][value], and the budget.

Assumption #1 - Given an array of products ordered by value, return the most valuable products

We start defining a java test creating a new BasketOptimized class.

Listing 10.1.1 – BaskedOptimized Test Case

```
public class BasketOptimizedTest {

  BasketOptimized basketOptimized;

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

```
  @Test
  public void productsOrderedByValue() {

    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);
  }
}
```

The first time, it should fail because the fill method doesn't exist. Then we need to create an easy implementation to pass the test: the sum of the values must be equal to 590 because this represents all selected products whose prices sum less than or equal to 4.

Now, we proceed to implement the *fill* method.

Listing 10.1.2 – BaskedOptimized fill method implementation

```
public class BasketOptimized {
  public double[][] fill(double[][] myProducts, double budget) {
    int len = myProducts.length;
    double[][] mostValueableProducts = new double[len][3];
    double sum = 0;
    for (int idx = 0; idx < len; idx++) {
      sum = sum + myProducts[idx][1]; //price
      if (sum <= budget) {
        mostValueableProducts[idx][0] =
                myProducts[idx][0]; //id
        mostValueableProducts[idx][1] =
                myProducts[idx][1]; //price
        mostValueableProducts[idx][2] =
                myProducts[idx][2]; //value
      }
    }
    return mostValueableProducts;
  }
}
```

Assumption #2 - Given an array of products not ordered by value, return the most valuable products

In this case, we pass a not ordered array, so we can see that our new test will fail.

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

We realize we need to order the Array by value because we want the most valuable products, so it is time to refactor our algorithm. What we need to do is to sort our input array.

```
public double[][] fill(double[][] myProducts, double budget) {
  Arrays.sort(myProducts, Collections.reverseOrder(
          Comparator.comparingDouble(a -> a[2])));
  int len = myProducts.length;
  double[][] mostValueableProducts = new double[len][3];
  double sum = 0;
  for (int idx = 0; idx < len; idx++) {
    sum = sum + myProducts[idx][1]; //price
    if (sum <= budget) {
      mostValueableProducts[idx][0] =
              myProducts[idx][0]; //id
      mostValueableProducts[idx][1] =
              myProducts[idx][1]; //price
      mostValueableProducts[idx][2] =
              myProducts[idx][2]; //value
    }
  }
  return mostValueableProducts;
}
```

Then we can see that our two first test cases were successful.

Assumption #3 - Given an array of products, we need to obtain the most valuable products from all possible combinations of the products

Imagine the following scenario:

```
double[][] myProducts = new double[][] {
                {1, 0.98, 230},
                {2, 0.51, 30},
                {3, 0.49, 28},
                {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);
```

The test expects a result of 590, which corresponds to the final price of 3,73US$ (0.98+0.98+0.48+1.29). Once the algorithm sort by value the input array, we obtain the following new input array:

```
[1.0, 0.98, 230.0]
[5.0, 0.98, 230.0]
[7.0, 0.48, 75.0]
[4.0, 1.29, 55.0]
[2.0, 0.51, 30.0]
[3.0, 0.49, 28.0]
[6.0, 4.86, 14.0]
```

But here we realize another combination of products that give us the most valuable products: 230+230+75+30+28 = 593, which corresponds to the final price of 3,44US$. Then we need to refactor our code to calculate all combinations (subsets) and return the most valuable products under a budget of 4 US$.

The subsets can be represented by the binary options from 0 to 7 (the array size).

Bitwise operators allow us to manipulate the bits within an integer. For example, the int value for 33 in binary is 00100001, where each position represents a power of two, starting with 2^0 at the rightmost bit.

```
int numIterations = (int) Math.pow(2, myProducts.length);
```

So, for seven products, we have 128 iterations. And for every iteration, we build an inner loop to decide which products to include in a subset of products.

Let see an example:

```
When int idx = 33
```

We need to build a subset with those products, which pass the following criteria:

```
if ((idx & (int) Math.pow(2, idx2)) == 0) {
```

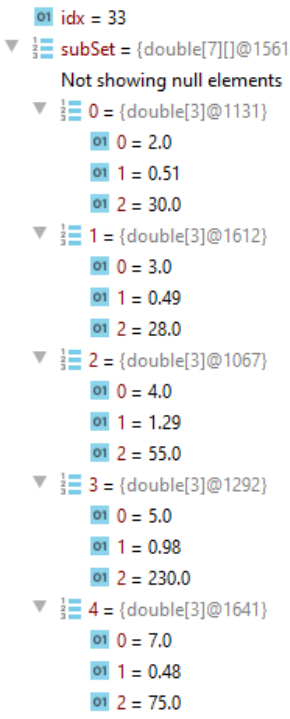
The following table shows the iteration in our inner loop:

binary-idx	idx2	Math.*pow*(2,idx2)	binary-pow(idx2)	binary-idx **AND** binary-pow(idx2)
00100001	0	1	0001	1
00100001	1	2	0010	0
00100001	2	4	0100	0
00100001	3	8	1000	0
00100001	4	16	00010000	0
00100001	5	32	00100000	1
00100001	6	64	01000000	0

That means that a new subset will include those products located at indexes 1, 2, 3, 4, and 6 from our Array of products. And we need to ask if we can afford to buy these products under our limited budget.

```
if (subSet.length > 0 && sumPrice <= budget) {
```

The following figure shows the result of our current iteration:



We build a HashMap to store all combinations and the sum of its values. Finally, we return the first element of the HashMap, ordered by value.

Listing 10.1.3 – BaskedOptimized implementation

```
public class BasketOptimized {

  public double[][] fill(double[][] myProducts, double budget) {
    int len = myProducts.length;
    int numIterations = (int) Math.pow(2, myProducts.length);
    Map<double[][], Double> combinations = new HashMap<>();

    for (int idx = 0; idx < numIterations; idx++) {
      double[][] subSet = new double[len][];
      double sumPrice = 0;
      double sumValue = 0;
      int i = 0;
      for (int idx2 = 0; idx2 < len; idx2++) {
        if ((idx & (int) Math.pow(2, idx2)) == 0) {
          subSet[i++] = myProducts[idx2];
          sumPrice = sumPrice + myProducts[idx2][1];
          sumValue = sumValue + myProducts[idx2][2];
        }
      }
      if (subSet.length > 0 && sumPrice <= budget) {
        //remove nulls
        subSet = Arrays.stream(subSet)
                .filter(s -> (s != null && s.length > 0))
                .toArray(double[][]::new);
        combinations.put(subSet, Double.valueOf(sumValue));
      }
    }

    LinkedHashMap<double[][], Double> reverseSortedMap =
        new LinkedHashMap<>();

    combinations.entrySet()
            .stream()
            .sorted(Map.Entry.comparingByValue(Comparator.reverseOrder()))
            .forEachOrdered(x -> reverseSortedMap
            .put(x.getKey(), x.getValue()));

    double[][] mostValueableProducts = reverseSortedMap
            .keySet().iterator().next();
    return mostValueableProducts;
  }
}
```

Tests

```
public class BasketOptimizedTest {

  BasketOptimized basketOptimized;

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

  @Test
  public void given_productsOrderedByValue_return_mostValuables() {

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

```
  @Test
  public void given_products_return_theMostValuables() {

    double[][] myProducts = new double[][]{
            {1, 0.98, 230},
            {2, 0.51, 30},
            {3, 0.49, 28},
            {4, 1.29, 55},
            {5, 0.98, 230},
            {6, 4.86, 14},
            {7, 0.48, 75}
    };

    double[][] mostValuableProducts =
            basketOptimized.fill(myProducts, 4);
    assertEquals(593d,
            Arrays.stream(mostValuableProducts).
                    mapToDouble(arr -> arr[2]).sum(), 0);
  }

}
```

10.2 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 at 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.

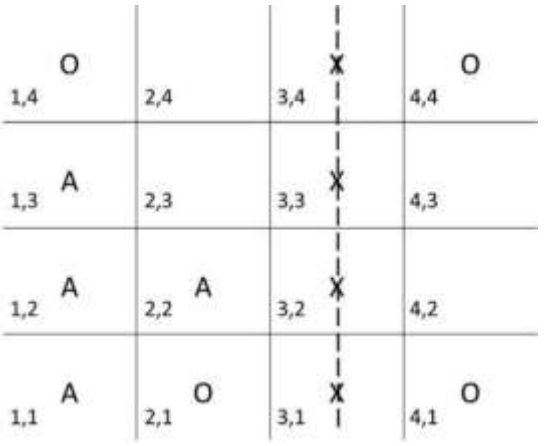


Figure 10.2.1 Tic-tac-toe game

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

Solution

We learn from Object-Oriented Design and SOLID principles 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 controls 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 10.2.1 shows one assumption about the setSize method.

```
Listing 10.2.1 – 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 10.2.2 shows an initial implementation of Board Class and the *setSize* method.

```
Listing 10.2.2 – Board Class, setSize method

public class Board {

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

```
  public void setSize(int size) {
    this.numOfPlaysAllowed = size * 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.

Test case #2: Enter a symbol based on valid coordinates

Once the size is set up, we need to accept valid coordinates and check if that location is still available.

```
@Test
public void whenCoordinatesAreNotBusyThenPutSymbol() throws Exception {
  board.setSize(3);
  board.putSymbol(1, 2, "X");
  board.putSymbol(2, 3, "O");
  assertEquals("O", board.getBoard()[1][2]);
}
```

Listing 10.2.3 shows the implementation of the *putSymbol* method.

Listing 10.2.3 – Board Class, putSymbol method

```
public void putSymbol(int x, int y, String character) {
  if (x < 1 || x > this.board.length)
    throw new RuntimeException(
      "X coordinate invalid, must be between 1 and " +
      this.board.length);

  if (y < 1 || y > this.board.length)
    throw new RuntimeException(
      "Y coordinate invalid, must be between 1 and " +
      this.board.length);
```

```
  if (board[x - 1][y - 1] != " ")
    throw new RuntimeException("Coordinates are busy");

  board[x - 1][y - 1] = character;
  this.numOfPlays++;
}
```

If we set the size to 4, that means that for the player, the lower limit is 1, and the upper limit is 4.

Test case #3: After every move, print the board

Every time a player enters valid coordinates, then the Board is updated and printed.

```
@Test
public void whenBoardIsNotNullThenPrintIsPossible() {
  board.setSize(10);
  board.putSymbol(1, 2, "X");
  board.putSymbol(2, 3, "O");
  board.print();
}
```

Listing 10.2.4 shows the implementation of the *print* method.

```
Listing 10.2.4 – Board Class, print method
```

```
public void print() {
  if (board == null)
    throw new RuntimeException("Board is not initialized");

  int size = board.length;
  for (int y = size - 1; y >= 0; y--) {
    for (int x = 0; x < size; x++) {
      if (x == size - 1) {
        System.out.print(board[x][y] + "");
      } else {
        System.out.print(board[x][y] + "|");
      }
    }
    System.out.println("");
  }
}
```

Here you can even delegate the print of the Board to another component, e.g., Console.print. In this way, when you want to print in XML o HTML format, Console Class will be responsible for implementing the new methods. *There should never be more than one reason for a class to change.*

Test Case #4: Returns a winner

Once we set the size, the game receives different moves. The Board Class checks if in the game exists a winner after a move. Listing 10.2.5 defines a test case to check if a whole horizontal line is filled.

Listing 10.2.5 – Board Class, getWinner Test Case

```
@Test
public void horizontalLineFilledThenWinnerX() {
  board.setSize(3);
  board.putSymbol(1, 1, "X");
  board.putSymbol(1, 2, "O");
  board.putSymbol(2, 1, "X");
  board.putSymbol(2, 3, "O");
  board.putSymbol(3, 1, "X");
  board.print();
  assertEquals("X", board.getWinner());
}
```

Then we iterate every horizontal line and check if it is filled with the same symbol. Listing 10.2.6 shows the implementation of the *getWinner* method.

Listing 10.2.6 – Board Class, getWinner method

```
private String getWinner() {
  String winner = null;
  for (int y = 0; y < board.length; y++) {
    String symbol = board[0][y];
    if (symbol != EMPTY_) {
      int counter = 1;
      for (int x = 1; x < board.length; x++) {
        if (symbol.equals(board[x][y])) {
          counter++;
        }
      }
      if (counter == board.length) {
        winner = symbol;
        return winner;
      }
    }
  }
  return winner;
}
```

We need the same logic for all the ways to win: vertical, diagonal, so we need to abstract every case in sub methods. Listing 10.2.7 shows the new getWinner method after refactoring.

Listing 10.2.7 – Board Class, getWinner method refactored

```
public String getWinner() {

  if (board == null)
    throw new RuntimeException("Board is not initialized");

  String winner = winnerInHorizonzalLine();
  if (winner != null) {
    return winner;
  } else {
    winner = winnerInVerticalLine();
    if (winner != null) {
      return winner;
    } else {
      winner = winnerInDiagonalBottomLeft();
      if (winner != null) {
        return winner;
      } else {
        winner = winnerInDiagonalTopLeft();
        if (winner != null) {
          return winner;
        } else {
          //we avoid to iterate the whole board everytime
          //to look for a draw
          if (this.numOfPlays == this.numOfPlaysAllowed) {
            return DRAW_;
          } else {
            return null;
          }
        }
      }
    }
  }
}
```

The previous code shows how we introduce three new variables:

A constant variable:

```
private final static String DRAW_ = "DRAW!";
```

numOfPlays and **numOfPlaysAllowed** are declared as member variables at the Board Class.

numOfPlaysAllowed is initialized at the setSize method:

```
this.numOfPlaysAllowed = size * size;
```

And the **numOfPlays** is incremented at the putSymbol method:

```
this.numOfPlays++;
```

Here are the other ways to win.

```
private String winnerInVerticalLine() {
  String winner = null;
  for (int x = 0; x < board.length; x++) {
    String symbol = board[x][0];
    if (symbol != EMPTY_) {
      int counter = 1;
      for (int y = 1; y < board.length; y++) {
        if (symbol.equals(board[x][y])) {
          counter++;
        }
      }
      if (counter == board.length) {
        winner = symbol;
        return winner;
      }
    }
  }
  return winner;
}


private String winnerInDiagonalBottomLeft() {
  String winner = null;
  String symbol = board[0][0];
  if (symbol != EMPTY_) {
    int counter = 1;
    for (int idx = 1; idx < board.length; idx++) {
      if (symbol.equals(board[idx][idx])) {
        counter++;
      }
    }
    if (counter == board.length) {
      winner = symbol;
      return winner;
    }
  }
  return winner;
}
```

```
private String winnerInDiagonalTopLeft() {
  String winner = null;
  String symbol = board[0][board.length - 1];
  if (symbol != EMPTY_) {
    int counter = 1;
    for (int idx = 1; idx < board.length; idx++) {
      if (symbol.equals(board[idx][board.length - 1 - idx])) {
        counter++;
      }
    }
    if (counter == board.length) {
      winner = symbol;
      return winner;
    }
  }
  return winner;
}
```

Now let see the Utils Class, which loads the symbols and the size of the Board.

Test case #5: Load file

We need a method that, given a Filename, retrieves the content of that File. Listing 10.2.8 shows the test case.

Listing 10.2.8 – Utils Test Class

```
public class UtilsTest {

  private Utils utils;

  @Before
  public void setUp() {
    utils = new Utils();
  }

  @Test
  public void whenFileExistsThenReturnContent() throws Exception {
    String content = utils.loadFile("symbols.txt");
    assertNotNull(content);
  }
}
```

Assumes that our configuration files are located under the project name. Listing 10.2.9 shows a loadFile implementation.

Listing 10.2.9 – Utils Class, loadFile method

```
import java.nio.file.Files;
import java.nio.file.Paths;

public class Utils {

  public String loadFile(String fileName) throws Exception {
    String content;
    try {
      content = new String(Files.readAllBytes(Paths.get(fileName)));
    } catch (IOException io) {
      throw new RuntimeException("File " + fileName + " not found");
    }
    return content;
  }
}
```

Test case #6: Get the board size

Once a Board File is loaded, we need to validate that it includes valid content. Listing 10.2.10 shows the assumption about how to validate the Board size. Based on the requirements, board size must be a value between 3 and 9.

Listing 10.2.10 – Utils Class, getBoardSize Test case

```
@Test
public void whenGetBoardSizeIs4ThenReturnSize() throws Exception {
  String content = utils.loadFile("board.txt");
  assertEquals(4, utils.getBoardSize(content));
}
```

Listing 10.2.11 shows the implementation of the getBoardSize.

Listing 10.2.11 – Utils Class, getBoardSize method

```
public int getBoardSize(String content) throws Exception {

  if (content == null)
    throw new RuntimeException("Invalid setting for the board");

  if (Integer.valueOf(content) < 3 || Integer.valueOf(content) > 9)
    throw new RuntimeException("Invalid setting for the board");

  return Integer.valueOf(content).intValue();
}
```

```
board.txt
4
```

Try always to build a small method for each thing. If you can divide the problem into small parts, that means that you can create great software for more significant projects

Test case #7: Get symbols

Once a symbol File is loaded, it validates that it is retrieved only three symbols. Listing 10.2.12 shows our assumption in one Test case.

Listing 10.2.12 – Utils Class, getSymbols Test Case

```
@Test
public void whenSymbolsIsThreeThenReturnValues() throws Exception {
  String content = utils.loadFile("symbols.txt");
  String[] symbols = utils.getSymbols(content);
  assertArrayEquals(new String[]{"X","O","A"}, symbols);
}
```

```
symbols.txt
X,O,A
```

Listing 10.2.13 shows the *getSymbols* method implementation.

Listing 10.2.13 – Utils Class, getSymbols method

```
public String[] getSymbols(String content) {
  String[] symbols = content.split(",");
  if (symbols.length != 3)
    throw new RuntimeException("Invalid settings for symbols");

  return symbols;
}
```

Test case #8: Receive input from the console

We create an auxiliary Input class to store the coordinates entered by the player. Listing 10.2.14 shows our assumption in a Test Case Class.

```
public class Input {
  private int x;
  private int y;
}
```

Listing 10.2.14 – Utils Class, getInput Test case

```
@Test
public void inputFromConsoleThenReturnInput() throws Exception {
  Input input = utils.getInputFromConsole("2,3");
  assertEquals(2, input.getX());
  assertEquals(3, input.getY());
}
```

Listing 10.2.15 shows the *getInputFromConsole* method implementation, where we need to accept valid numbers.

Listing 10.2.15 – Utils Class, getInputFromConsole method

```
public Input getInputFromConsole(String inputFromConsole) {

  if (inputFromConsole == null || inputFromConsole.trim().length() <= 0)
    throw new RuntimeException("Invalid input from console");

  Pattern p = Pattern.compile("[0-9]*\\.?,[0-9]+");
  Matcher matcher = p.matcher(inputFromConsole);
  if (!matcher.find())
    throw new RuntimeException("Invalid input from console");

  String[] inputSplitted = inputFromConsole.split(",");

  Input input = new Input();
  input.setX(Integer.valueOf(inputSplitted[0]).intValue());
  input.setY(Integer.valueOf(inputSplitted[1]).intValue());
  return input;
}
```

Now, we implement the Player Class.

Listing 10.2.16 shows a design of a Player Class and its sub-classes.

Listing 10.2.16 – Player Class, and subClasses

```
public class Player {
  private String symbol;
  //getters and setters are omitted
}


public class Human extends Player {}

public class IA extends Player {}
```

Test case #9: Implements how IA plays

To make an easy tic tac toe program, we will allow the IA player to fill the first coordinate available on the Board. Of course, you can write a more complex implementation based on graphs, for example, to decide the best move of the IA player. Listing 10.2.17 shows assumptions about the IA player.

Listing 10.2.17 – Player Class, Test Cases

```
public class PlayerTest {

  private Player humanPlayer;
  private IA iaPlayer;
  private Board board;

  @Before
  public void setUp() {
    humanPlayer = new Human();
    iaPlayer = new IA();
    board = new Board();
    board.setSize(3);
  }

  @Test
  public void whenSetSymbolThenReturnSameSymbol() throws Exception {
    humanPlayer.setSymbol("A");
    assertEquals("A", humanPlayer.getSymbol());
  }

  @Test
  public void whenIAPlaysThenReturnInput() throws Exception {
    iaPlayer.setSymbol("A");
    board.putSymbol(1, 1, "X");
    board.putSymbol(2, 2, "O");
    board.print();
    Input input = iaPlayer.play(board.getBoard());
    assertEquals(1, input.getX());
    assertEquals(2, input.getY());
  }
```

Listing 10.2.18 shows the play method implementation.

Listing 10.2.18 – Player Class, play method

```
public class IA extends Player {
  private final static String EMPTY_ = " ";
```

```
  public Input play(String[][] board) {
    Input input = new Input();
    //Assumption: find the first coordinate available
    for (int x = 0; x < board.length; x++) {
      for (int y = 0; y < board.length; y++) {
        String symbol = board[x][y];
        if (symbol.equals(EMPTY_)) {
          input.setX(x + 1);
          input.setY(y + 1);
          return input;
        }
      }
    }
    return input;
  }
}
```

Now that we have all components ready, it is time to build the main Class. Listing 10.2.19 shows how App Class assembly all components.

Listing 10.2.19 – App Class

```
public class App {
  private final static String DRAW_ = "DRAW!";

  public static void main(String[] args) throws Exception {

    System.out.println("Welcome to TIC TAC TOE 2.0!");

    Scanner scanner = new Scanner(System.in);

    Utils utils = new Utils();
    int boardSize = utils.getBoardSize(utils.loadFile("board.txt"));

    Board board = new Board();
    board.setSize(boardSize);

    System.out.println("You are playing in a board " +
      boardSize + "x" + boardSize);
    String[] symbols = utils.getSymbols(utils.loadFile("symbols.txt"));

    Player player1 = new Human();
    player1.setSymbol(symbols[0]);
    Player player2 = new Human();
    player2.setSymbol(symbols[1]);
    Player player3 = new IA();
    player3.setSymbol(symbols[2]);

    List<Player> playersIterator = new ArrayList<>();
```

```
    playersIterator.add(player1);
    playersIterator.add(player2);
    playersIterator.add(player3);
    Collections.shuffle(playersIterator);

    int idx = 0;
    boolean stillPlaying = true;
    while (stillPlaying) {
      Player player = playersIterator.get(idx++);
      if (player instanceof IA) {
        System.out.println("Player " + player.getSymbol() +
        " enter your coordinates in Format x,y: ");
        Input input = ((IA) player).play(board.getBoard());
        board.putSymbol(input.getX(), input.getY(), player.getSymbol());
      } else {
        boolean coordinateOk = false;
        while (!coordinateOk) {
          System.out.println("Player " + player.getSymbol() +
          " enter your coordinates in Format x,y: ");
          try {
            Input input = utils.getInputFromConsole(scanner.nextLine());
            board.putSymbol(input.getX(), input.getY(), player.getSymbol());
            coordinateOk = true;
          } catch (RuntimeException ex) {
            System.out.println(ex.getMessage());
          }
        }
      }
      board.print();
      String winner = board.getWinner();
      if (winner != null) {
        stillPlaying = false;
        if (winner.equals(DRAW_)) {
          System.out.println(DRAW_);
        } else {
          System.out.println("Player " + winner + " is the Winner!");
        }
      }
      if (idx == 3) //to control the turn of every player
        idx = 0;
    }
  }
}
```

DEMO:

Welcome to TIC TAC TOE 2.0!

You are playing in a board 4x4

Player A enter your coordinates in Format x,y:

```
1,1
 | | |
 | | |
 | | |
A| | |
```

Player O enter your coordinates in Format x,y:

```
1,1
```

Coordinates are busy

Player O enter your coordinates in Format x,y:

```
2,1
 | | |
 | | |
 | | |
A|O| |
```

Player X enter your coordinates in Format x,y:

```
3,4
 | |X|
 | | |
 | | |
A|O| |
```

Player A enter your coordinates in Format x,y:

```
 | |X|
 | | |
A| | |
A|O| |
```

Player O enter your coordinates in Format x,y:

```
.
.
.
.
O| |X|O
A| | |
A|A|X|
A|O|X|O
```

Player X enter your coordinates in Format x,y:

```
3,3
O| |X|O
A| |X|
A|A|X|
A|O|X|O
```

Player X is the Winner!

The next section includes all test cases:

Tests

```
public class BoardTest {

  private Board board;

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

  @Rule
  public ExpectedException thrownException = ExpectedException.none();

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

  @Test
  public void whenPutCharacterWrongCoordinateThenThrowError() throws Exception {
    board.setSize(4);
    thrownException.expect(RuntimeException.class);
    board.putSymbol(1, 5, "X");
  }

  @Test
  public void whenCoordinateIsValidThenPutCharacterIsOk() throws Exception {
    board.setSize(3);
    board.putSymbol(1, 2, "X");
    assertEquals("X", board.getBoard()[0][1]);
  }

  @Test
  public void whenCoordinatesAreBusyThenPutCharacterThrowError() throws Exception {
    board.setSize(3);
    board.putSymbol(1, 2, "X");
    thrownException.expect(RuntimeException.class);
    board.putSymbol(1, 2, "O");
  }

  @Test
  public void whenCoordinatesAreNotBusyThenPutSymbol() throws Exception {
    board.setSize(3);
    board.putSymbol(1, 2, "X");
    board.putSymbol(2, 3, "O");
    assertEquals("O", board.getBoard()[1][2]);
  }
```

```
@Test
public void whenBoardIsNullThenPrintThrowsError() {
  thrownException.expect(RuntimeException.class);
  board.print();
}

@Test
public void whenBoardIsNotNullThenPrintIsPossible() {
  board.setSize(10);
  board.print();
}

@Test
public void horizontalLineIsNotSameSymbolThenReturnNull() {
  board.setSize(3);
  board.putSymbol(2, 2, "X");
  board.putSymbol(1, 2, "O");
  board.putSymbol(1, 3, "X");
  board.putSymbol(2, 1, "O");
  board.putSymbol(3, 2, "X");
  board.print();
  assertEquals(null, board.getWinner());
}

@Test
public void horizontalLineFilledThenWinnerX() {
  board.setSize(3);
  board.putSymbol(1, 1, "X");
  board.putSymbol(1, 2, "O");
  board.putSymbol(2, 1, "X");
  board.putSymbol(2, 3, "O");
  board.putSymbol(3, 1, "X");
  board.print();
  assertEquals("X", board.getWinner());
}

@Test
public void horizontalLineFilledThenWinnerO() {
  board.setSize(4);
  board.putSymbol(1, 1, "X");
  board.putSymbol(1, 4, "O");
  board.putSymbol(2, 1, "X");
  board.putSymbol(2, 4, "O");
  board.putSymbol(3, 1, "X");
  board.putSymbol(3, 4, "O");
  board.putSymbol(3, 3, "X");
  board.putSymbol(4, 4, "O");
  board.print();
  assertEquals("O", board.getWinner());
}
```

```
@Test
public void verticalLineIsSameSymbolThenReturnWinnerX() {
  board.setSize(3);
  board.putSymbol(2, 1, "X");
  board.putSymbol(1, 3, "O");
  board.putSymbol(2, 2, "X");
  board.putSymbol(3, 3, "O");
  board.putSymbol(2, 3, "X");
  board.print();
  assertEquals("X", board.getWinner());
}

@Test
public void diagonalLineSameSymbolThenReturnWinnerX() {
  board.setSize(3);
  board.putSymbol(1, 1, "X");
  board.putSymbol(1, 2, "O");
  board.putSymbol(2, 2, "X");
  board.putSymbol(2, 1, "O");
  board.putSymbol(3, 3, "X");
  board.print();
  assertEquals("X", board.getWinner());
}

@Test
public void diagonalLineSameSymbolThenReturnWinnerO() {
  board.setSize(3);
  board.putSymbol(1, 1, "X");
  board.putSymbol(1, 3, "O");
  board.putSymbol(3, 2, "X");
  board.putSymbol(2, 2, "O");
  board.putSymbol(3, 3, "X");
  board.putSymbol(3, 1, "O");
  board.print();
  assertEquals("O", board.getWinner());
}

@Test
public void diagonalLineSameSymbolThenReturnWinnerX_2() {
  board.setSize(5);
  board.putSymbol(1, 1, "O");
  board.putSymbol(1, 5, "X");
  board.putSymbol(3, 2, "O");
  board.putSymbol(2, 4, "X");
  board.putSymbol(3, 4, "O");
  board.putSymbol(3, 3, "X");
  board.putSymbol(1, 2, "O");
  board.putSymbol(4, 2, "X");
  board.putSymbol(5, 3, "O");
  board.putSymbol(5, 1, "X");
  board.print();
  assertEquals("X", board.getWinner());
}
```

```
  @Test
  public void whenIsDrawThenReturnTrue() {
    board.setSize(3);
    board.putSymbol(1, 1, "X");
    board.putSymbol(1, 2, "O");
    board.putSymbol(3, 2, "X");
    board.putSymbol(2, 2, "X");
    board.putSymbol(3, 3, "O");
    board.putSymbol(2, 1, "O");
    board.putSymbol(1, 3, "X");
    board.putSymbol(3, 1, "O");
    board.putSymbol(2, 3, "X");
    board.print();
    assertEquals("DRAW!", board.getWinner());
  }
}

import org.junit.rules.ExpectedException;
import static org.junit.Assert.*;

public class UtilsTest {

  private Utils utils;

  @Before
  public void setUp() {
    utils = new Utils();
  }

  @Rule
  public ExpectedException thrownException = ExpectedException.none();

  @Test
  public void whenFileNotExistsThenThrowError() throws Exception {
    thrownException.expect(RuntimeException.class);
    utils.loadFile("board2.txt");
  }

  @Test
  public void whenFileExistsThenReturnContent() throws Exception {
    String content = utils.loadFile("symbols.txt");
    assertNotNull(content);
  }

  @Test
  public void whenGetBoardSizeIs11ThenThrowError() throws Exception {
    thrownException.expect(RuntimeException.class);
    utils.getBoardSize("11");
  }
```

```
  @Test
  public void whenGetBoardSizeIs4ThenReturnSize() throws Exception {
    String content = utils.loadFile("board.txt");
    assertEquals(4, utils.getBoardSize(content));
  }

  @Test
  public void whenGetSymbolsIsNotThreeThenThrowError() throws Exception {
    thrownException.expect(RuntimeException.class);
    utils.getSymbols("A,B,C,D");
  }

  @Test
  public void whenSymbolsIsThreeThenReturnValues() throws Exception {
    String content = utils.loadFile("symbols.txt");
    String[] symbols = utils.getSymbols(content);
    assertArrayEquals(new String[]{"X","O","A"}, symbols);
  }

  @Test
  public void whenInputFromConsoleIsWrongThrowException() throws Exception {
    thrownException.expect(RuntimeException.class);
    utils.getInputFromConsole("");
  }

  @Test
  public void inputFromConsoleIsWrongFormatThrowException() throws Exception {
    thrownException.expect(RuntimeException.class);
    utils.getInputFromConsole("10");
  }

  @Test
  public void inputFromConsoleThenReturnInput() throws Exception {
    Input input = utils.getInputFromConsole("2,3");
    assertEquals(2, input.getX());
    assertEquals(3, input.getY());
  }
}
```

Big O Notation

Big O Notation is a mathematical notation that helps us analyze how complex an algorithm is in terms of time and space. When we build an application for one user or millions of users, it matters.

We usually 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 independently of the input size in one step.
- 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 a new input size.

Order of Growth of Common Algorithms

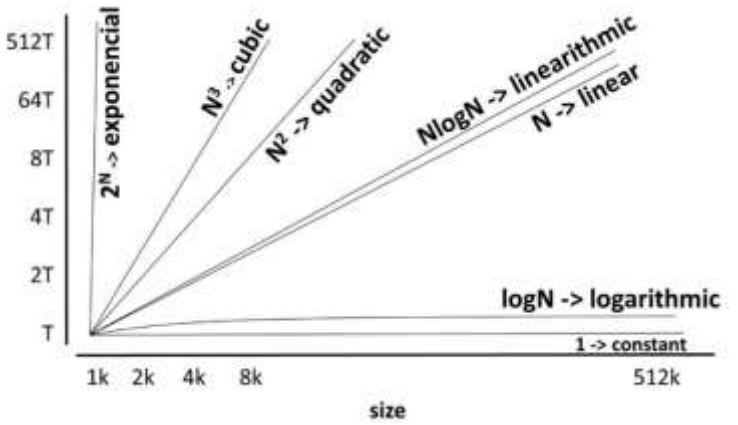


Figure A.1 Big O Notation - order of growth

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

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");
}
```

In a best-case scenario, an *add* method takes O(1) time. The worst-case scenario takes O(n).

O(N) – Linear

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);
}
```

O(N^2) – Quadratic

When we have two loops nested in our code, we say it runs in quadratic time O(N^2). For

example, when a 2D matrix is initialized in a tic-tac-toe game.

```
private String [][] board;

public void initializeBoard(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] = " ";
    }
  }
}


@Test
public void test_quadraticTime() {
  bigONotation.initializeBoard(3);
}
```

O(N³) - Cubic

We say that our algorithm runs in Cubic time when the code includes at the most three nested loops. For example, given N integers, how many triples sum to precisely zero? One approach (not the best) is to use three nested loops.

```
public int countThreeSum(int[] numbers) {
  int N =numbers.length;
  int count =0;
  for (int i = 0; i<N; i++)
    for (int j = i+1; j<N; j++)
      for (int k = j+1; k<N; k++)
        if (numbers[i] + numbers[j] + numbers[k] == 0)
          count++;
  return count;
}


@Test
public void test_countThreeSum() {
  final int[] numbers = {30, -40, -20, -10, 40, 0, 10, 5};
  assertTrue(bigONotation.countThreeSum(numbers) == 4);
}
```

O(LogN) – Logarithmic

This kind of algorithm produces a growth curve that peaks at the beginning and slowly flattens out as the size of the input increase.

$Log_2 8 = 3$

$Log_2 16 = 4$

$Log_2 32 = 5$

The binary search uses at most LogN key compares to search in a sorted array of size N. With eight elements, it takes three comparisons, with 16 elements takes four comparisons, with 32 elements takes five comparisons, and so on.

How to Find the Time Complexity of an Algorithm

To find the Big O complexity of an algorithm follows the following rules:

- Ignore the lower order terms
- Drop the leading constants

Example: If the time complexity of an algorithm is $2n^3 + 4n + 3$. Its Big O complexity simplifies to $O(n^3)$.

Example: Given the following algorithm:

```
public Integer sumOfEvenNumbers(Integer N) {
  int sum = 0;
  for (int number = 1; number <= N; number++)
    if ((number % 2) == 0)
      sum = sum + number;

  return sum;
}
```

First, we split the code into individual operations and then compute how many times it is being executed, as is shown in the following table.

Operation	Number of executions
`int sum = 0;`	1
`int number = 1;`	1
`number <= N;`	N
`number++`	N
`if ((number % 2) == 0)`	N
`sum = sum + number;`	N
`return sum;`	1

Now, we need to sum how many times each operation is executing.

Time complexity = 1 + 1 + N + N + N + N + 1 => 4N + 3

Big O complexity = O(N)

Why Big O Notation ignores Constants?

Big O Notation describes how many steps are required relative to the number of data elements. And it serves as a way to classify the long-term growth rate of algorithms.

For instance, O(N) will be faster than O(N2) for all amounts of data, as shown in Figure A.2.

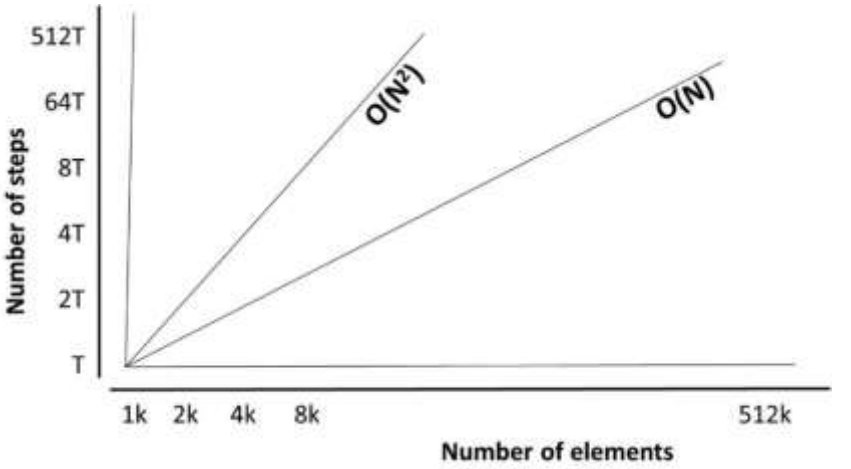


Figure A.2 O(N) is faster than O(N²) for all amounts of data

Now, if we compare O(100N) with O(N²), we can see that O(N²) is faster than O(100N) for some amounts of data, as shown in Figure A.3.

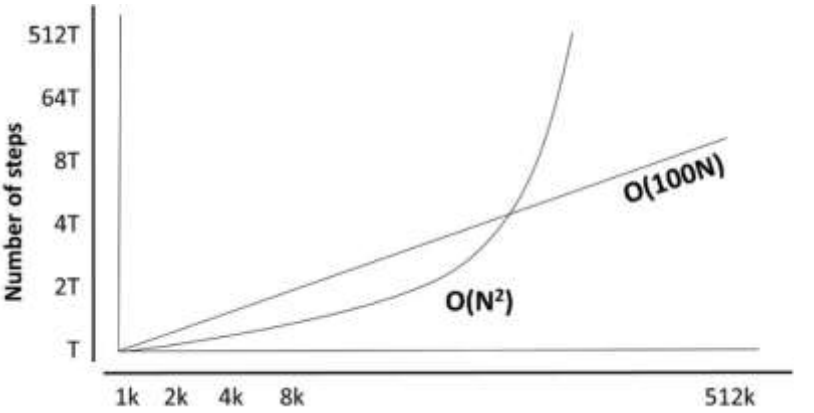


Figure A.3 O(N²) is faster than O(100N) for some amounts of data

But after a point, O(100N) becomes faster and remains faster for all increasing amounts of data from that point onward. And that is the reason why Big O Notation ignores constants. Because of this, O(100N) is written as O(N).