Contenido

Arrays and Strings

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.

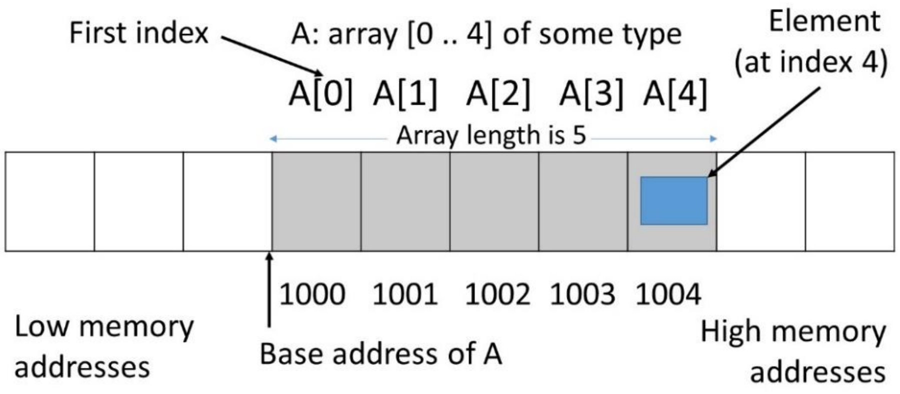


Figure 1.0.1 Array

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;
```

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

```java
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

```java
@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:

```
@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

- $1 \leq m, n \leq 10^5$
- $1 \leq \text{startIndices}[i] \leq \text{endIndices}[i] \leq n$
- 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");
```

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 he 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.

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.

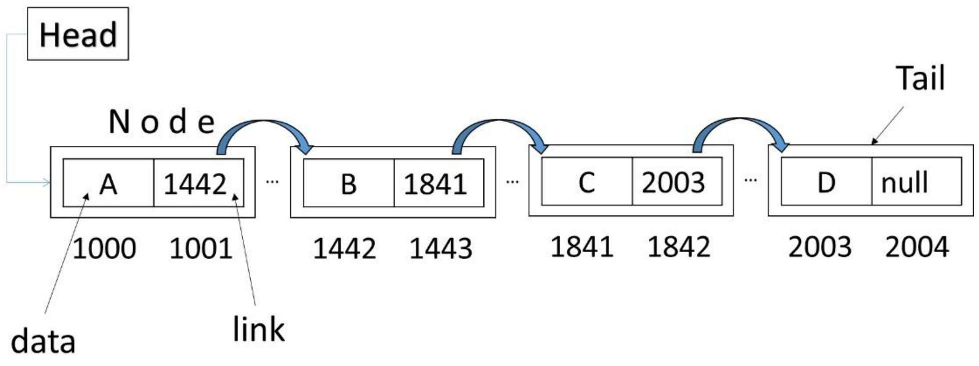


Figure 1.0.1 Linked list

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.

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 + ", ");
      }
    }
  }
}
```

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
```

```java
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);
  }
  //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.

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

```java
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

```java
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.

5.4 Binary Search

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

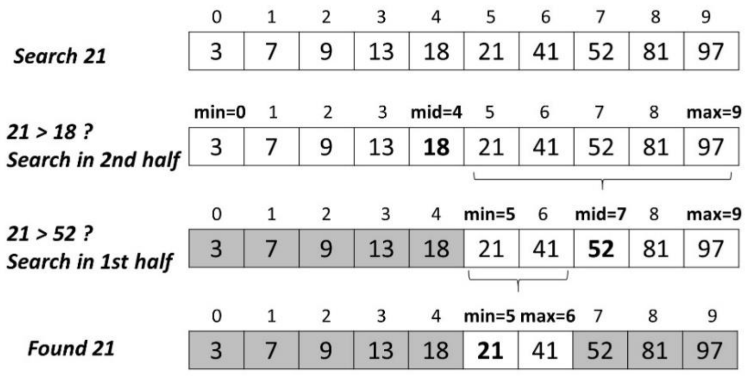


Figure 5.4 Binary Search example

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.

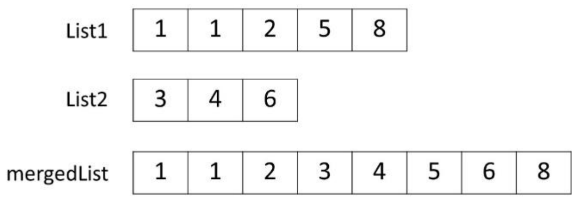


Figure 5.5 Merge Two Sorted Lists

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.

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.

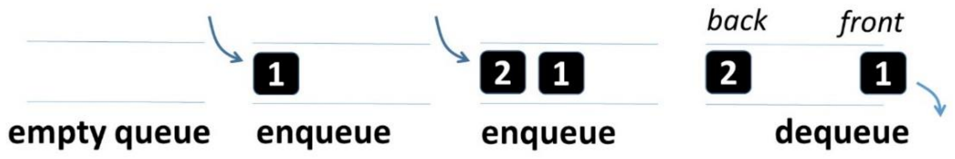


Figure 6.0.2 Queue

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.

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*.

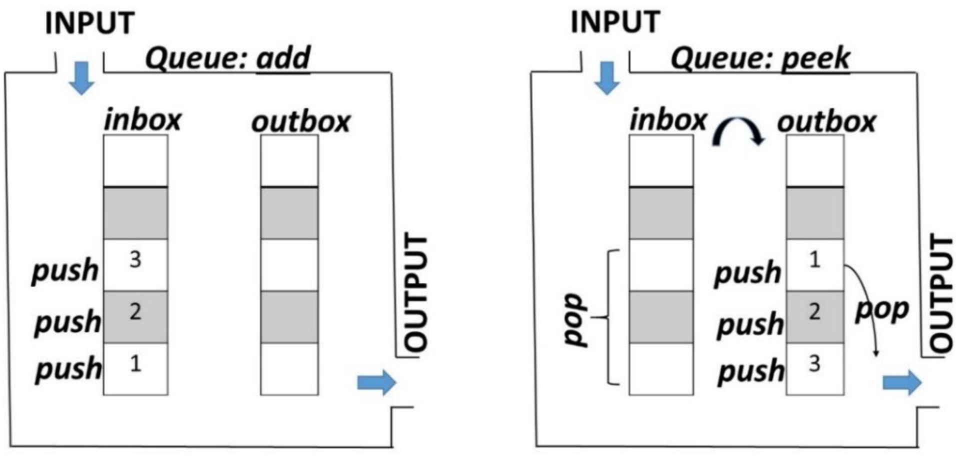


Figure 6.2 Queue via Stacks

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<>();
  }
```

```
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.

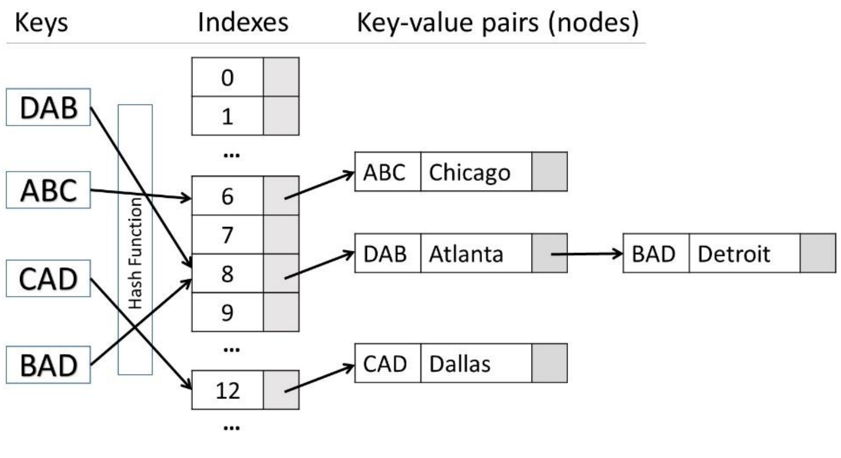


Figure 7.1 Design a Hash Table

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

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);
    }
```

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

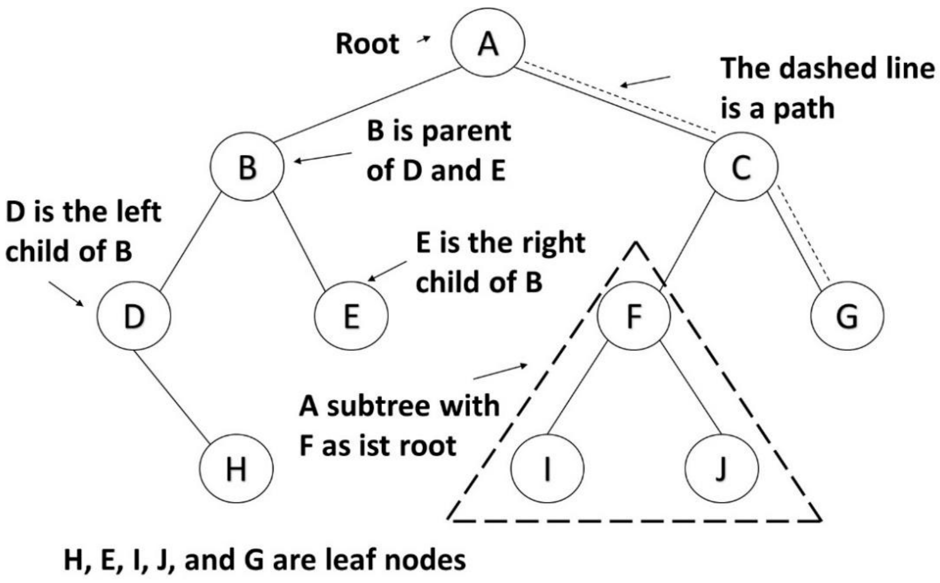

Root
A
The dashed line is a path
B is parent of D and E
B
C
D is the left child of B
E is the right child of B
D
E
F
G
A subtree with F as ist root
H
I
J
H, E, I, J, and G are leaf nodes

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

```java
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

```java
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;
  }
}
```

```
  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.

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.

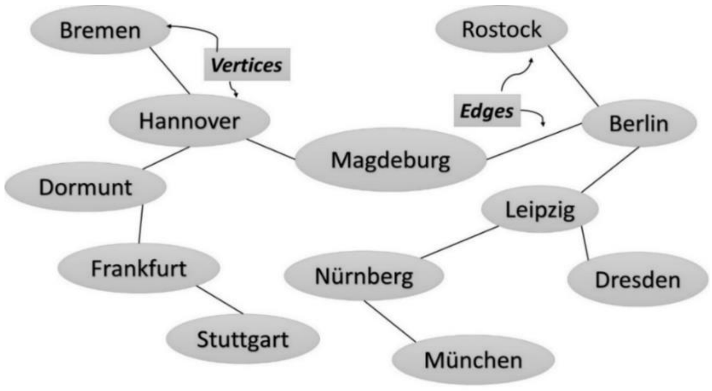


Figure 9.0 Graph - terminology

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.

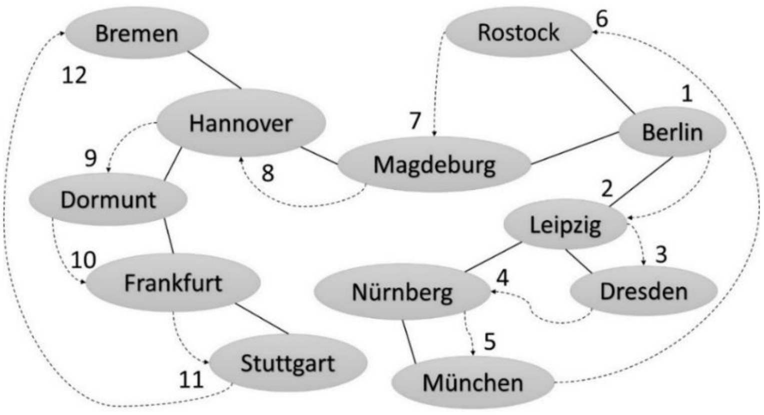


Figure 9.1.1 Depth-first search - the sequence of steps

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.

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.

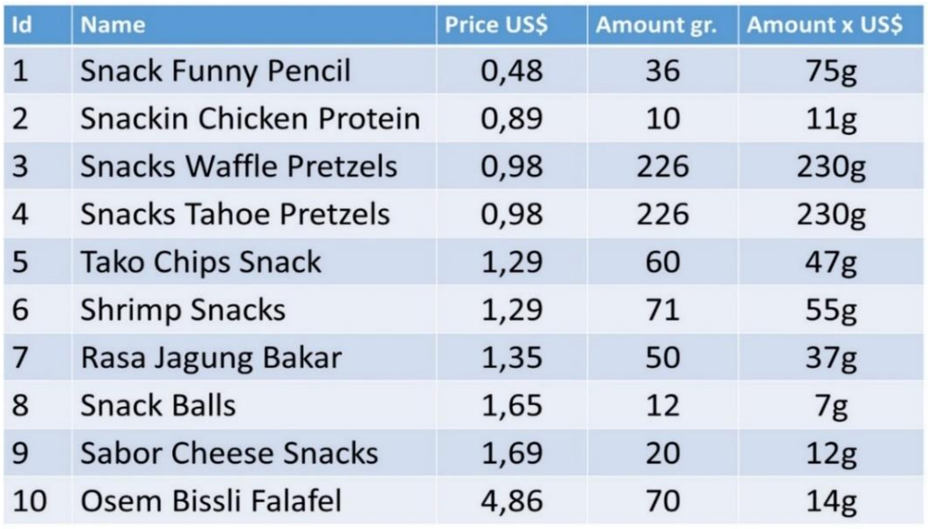
10.1 Optimize Online Purchases

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`

```java
public class BasketOptimizedTest {

  BasketOptimized basketOptimized;

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

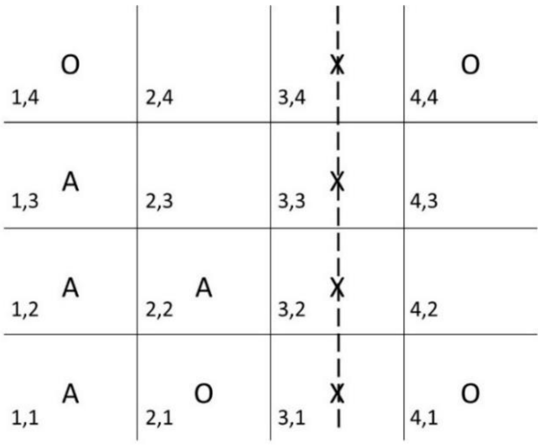


Figure 10.2.1 Tic-tac-toe game

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);
```

Big O Notation

Big O Notation is a mathematical function that helps us analyze how complex an algorithm is in time and space. It matters when we build an application for millions of users.

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

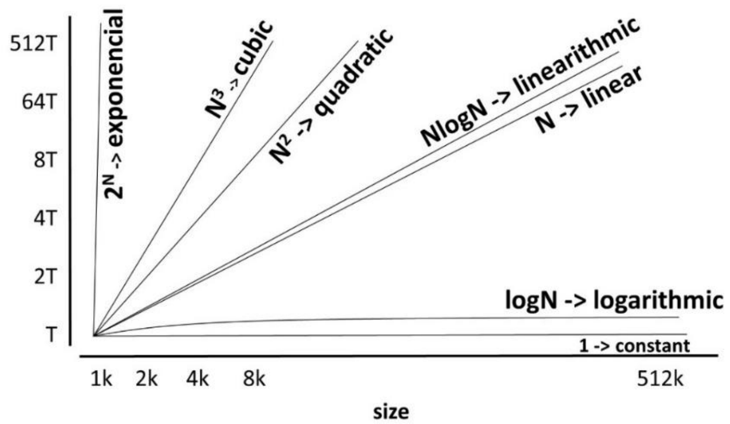


Figure A.1 Big O Notation - order of growth

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.

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.

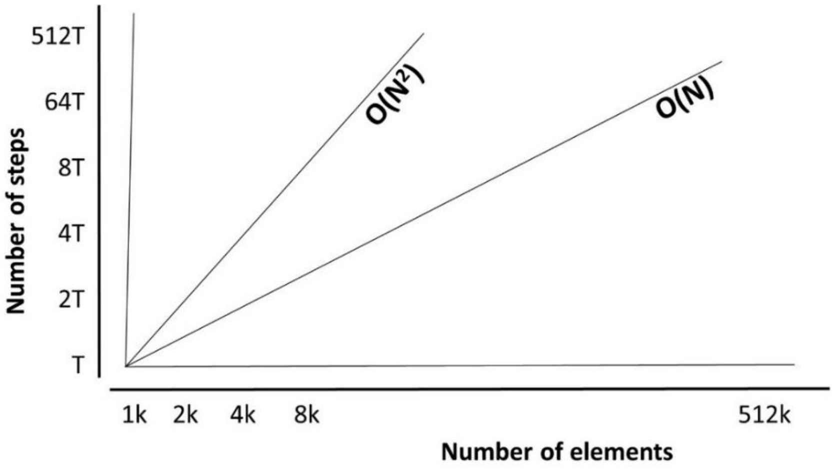


Figure A.2 O(N) is faster than O(N²) for all amounts of data

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.

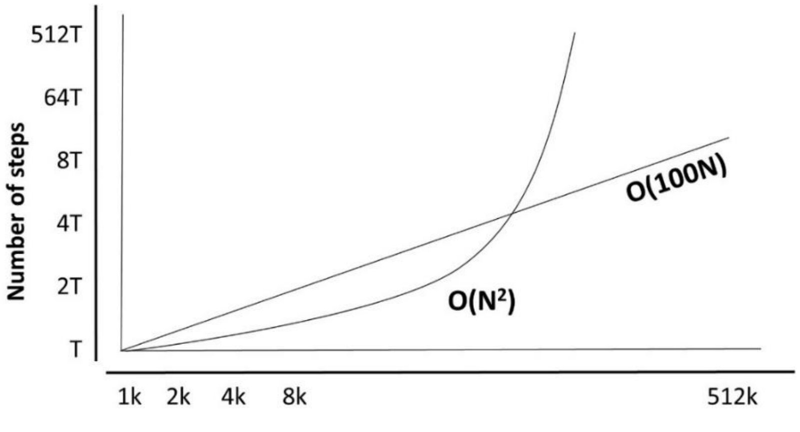


Figure A.3 O(N²) is faster than O(100N) for some amounts of data

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).