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.

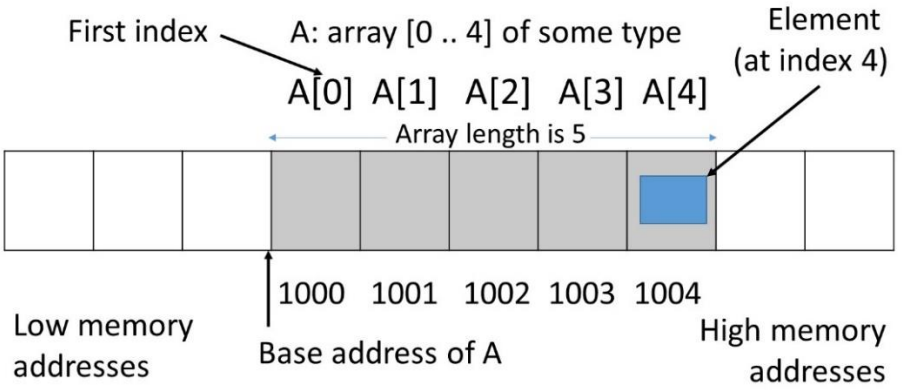

Figure 1.0.1 Array

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.

Characteristics of Arrays:

- Homogeneous Data: All elements in an array must be of the same data type, such as integers, strings, or floating-point numbers.

```
//declares an array of integers

int[] arrayOfInts;
```

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
```

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("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

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.

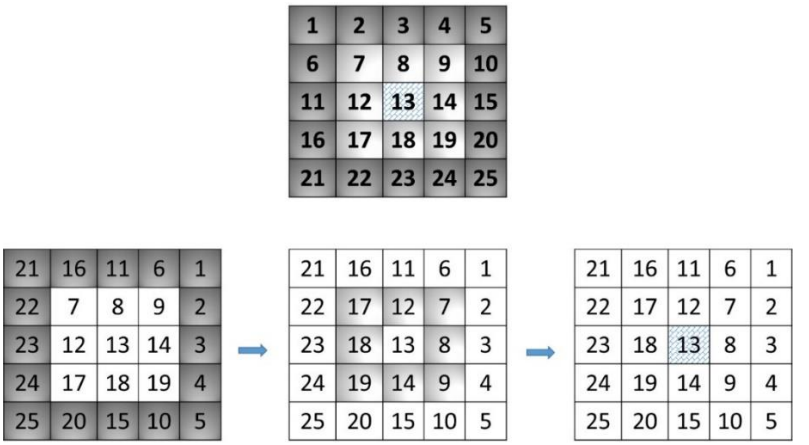

Figure 1.6 Rotate a Matrix by 90 Degrees

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 the 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

```
public class MatrixUtilsTest {

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

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

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) {
```

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.

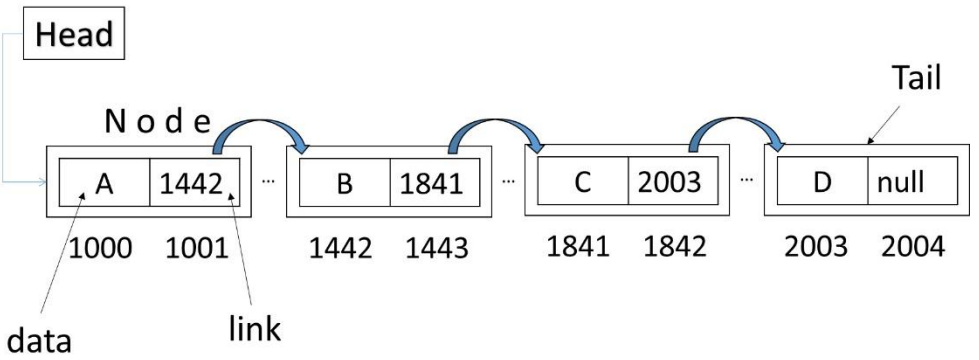

Figure 2.0.1 Linked list

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.

You should use a linked list when you need to store a variable number of items that may have different types or sizes. Linked lists are also preferable when you need to perform frequent insert or delete operations at any position and don't care about random access, sorting, or searching.

Applications:

- In a digital music service, use a linked list to create a playlist to add or remove dynamically your favorite songs.

- In an e-commerce application, use a linked list to create a shopping list to hold frequently requested items.

- You might use a linked list to implement a stack, a queue, or a hash table.

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;
```

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

```
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);
  }
```

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.

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
```

5.4 Binary Search

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

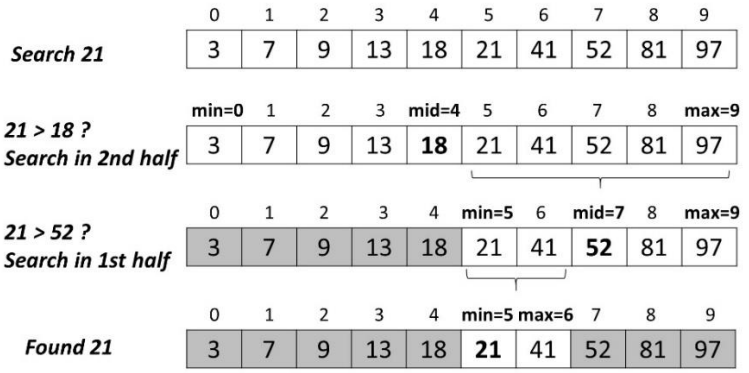

Figure 5.4 Binary Search example

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;
  }
}
```

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*.

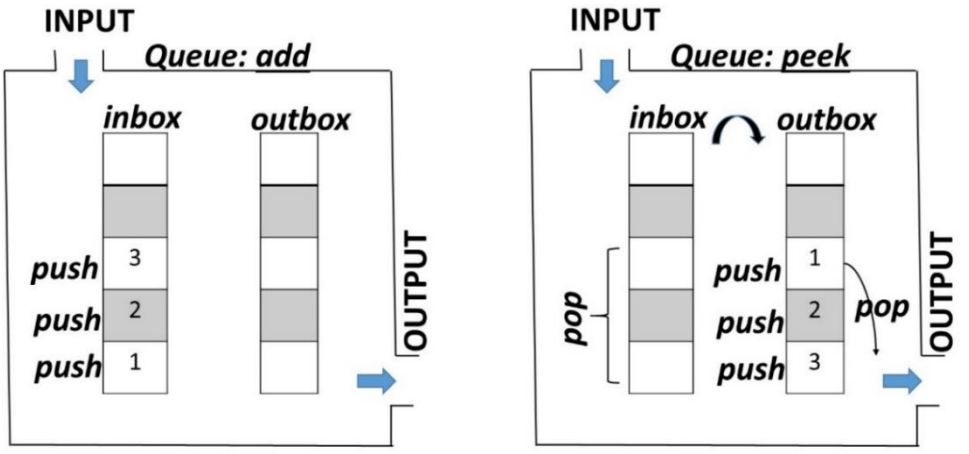

Figure 6.2 Queue via Stacks

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<>();
  }
```

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.

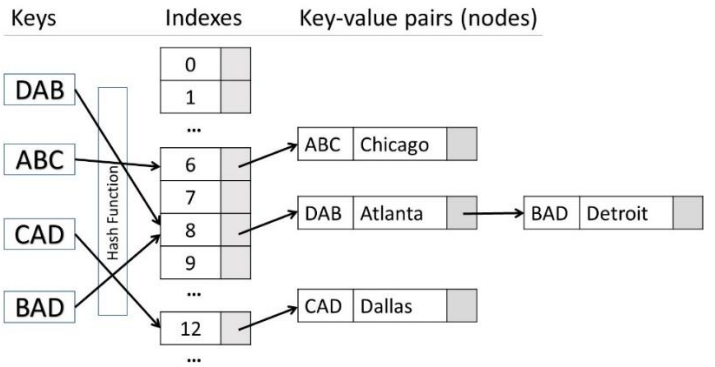

Figure 7.1 Design a Hash Table

```
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.

```
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;
    }
  }
}
```

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

9.1 Depth-First Search (DFS)

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

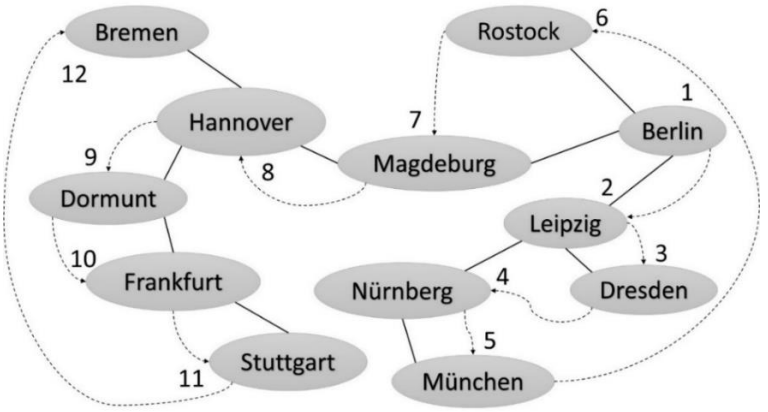

Figure 9.1.1 Depth-first search - the sequence of steps

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.

10.1 Optimize Online Purchases

A **Knapsack** problem is a classic optimization problem in computer science and mathematics. It belongs to a class of problems known as combinatorial optimization problems. The basic idea of the Knapsack problem is to determine the most valuable combination of items to include in a knapsack, given their weights and values, without exceeding the capacity of the knapsack.

Greedy Algorithm

A greedy algorithm is an algorithmic paradigm that makes locally optimal choices at each stage with the hope of finding a global optimum. The idea is to make the best possible choice at each step without worrying about the future consequences. Greedy algorithms are often used for optimization problems where you need to make a series of decisions, and at each decision point, you choose the option that seems most promising at that moment.

Backtracking Algorithm

Backtracking is a general algorithm for finding all (or some) solutions to computational problems, particularly constraint satisfaction problems. It incrementally builds candidates for solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.

Backtracking is used to explore different arrangements until a valid solution is found or all possibilities are exhausted.

In summary, greedy algorithms make locally optimal choices in the hope of finding a global optimum, while backtracking explores different choices systematically and abandons paths that cannot lead to a solution.

Problem

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},
```

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.

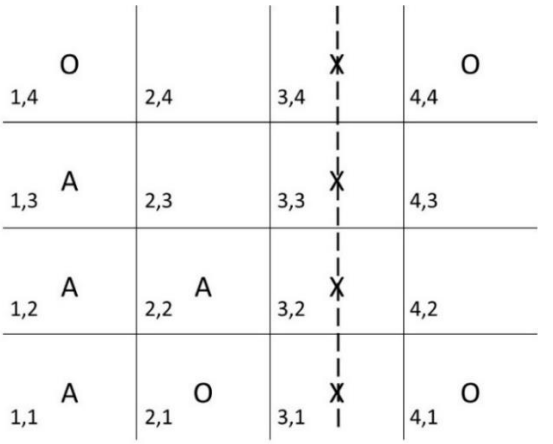

Figure 10.2.1 Tic-tac-toe game

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

Solution

We always have different ways to face a solution, but when you use concepts and techniques from the actual software world, you have more opportunities to crack the interview.

Always explain your assumptions and how you will solve the problem in front of the interviewer. Here is my approach.

We learn from Object-Oriented[1] Design and SOLID principles that we need to delegate responsibilities to different components or classes. For this game, we identify the following classes:

Board – set size, get Winner, draw?

Player – (Human, IA)

Utils – to load configuration files.

---

[1] https://codersite.dev/understanding-oop-concepts/

App – the main Class that assembly and controls our different components.

We are going to adopt Test Driven Development (TDD) to implement the solution. TDD allows us to design, build, and test the smallest methods first and assemble them later. Most importantly, we can refactor independently one test case without breaking the rest of the test cases.

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;
  private int numOfPlaysAllowed = 0;

  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;
```

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

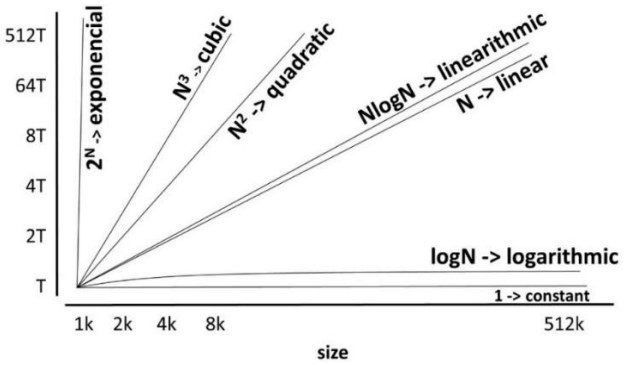

Figure A.1 Big O Notation - order of growth

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

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)