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.

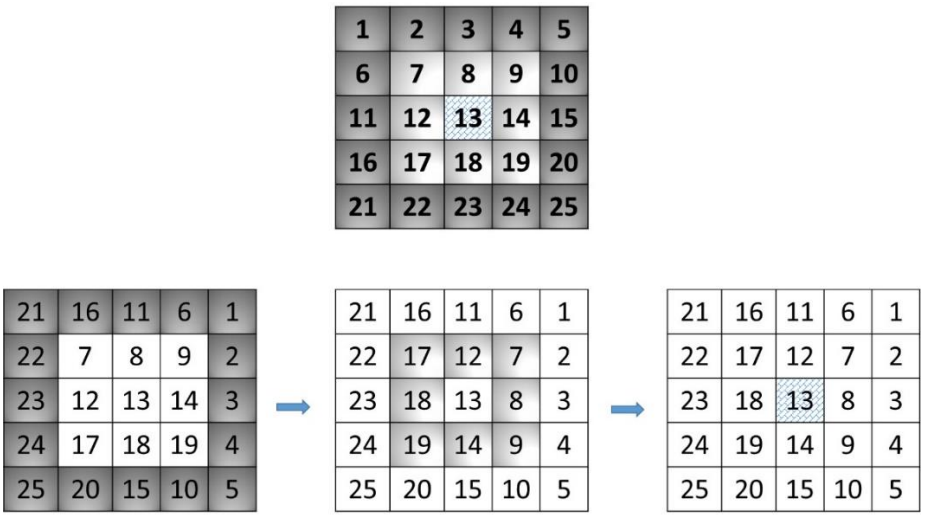

Figure 1.6 Rotate a Matrix by 90 Degrees

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
```

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.

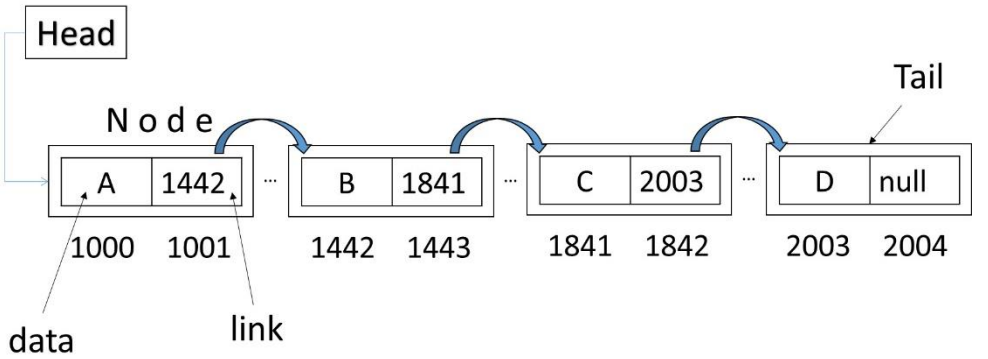

Figure 1.0.1 Linked list

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.

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

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, …`

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

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

Example:

```
First pass-through:
{6, 4, 9, 5} -> {4, 6, 9, 5} swap because of 6 > 4
{4, 6, 9, 5} -> {4, 6, 9, 5}
{4, 6, 9, 5} -> {4, 6, 5, 9} swap because of 9 > 5
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);
}
```

```
@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.

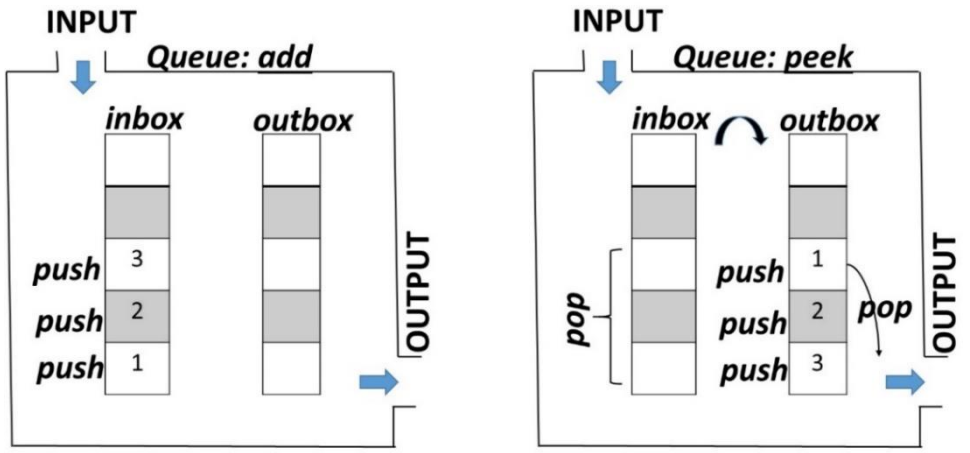

Figure 6.2 Queue via Stacks

`Listing 6.2 – Queue via Stacks`

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.

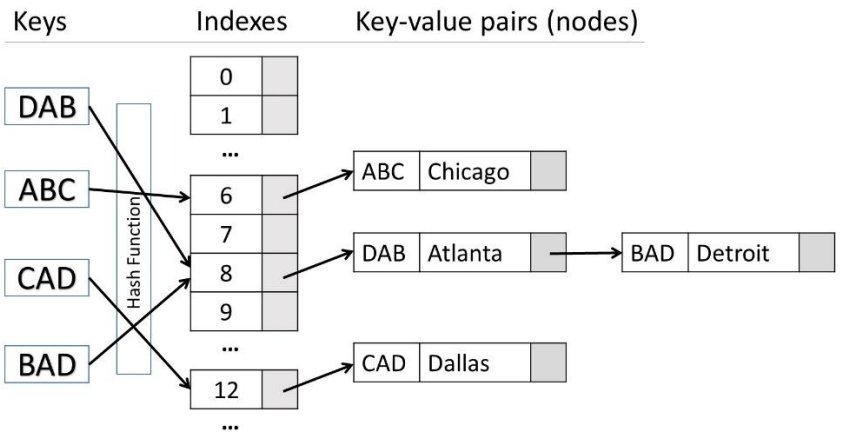

Figure 7.1 Design a Hash Table

Trees

8.0 Fundamentals

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

8.1 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.

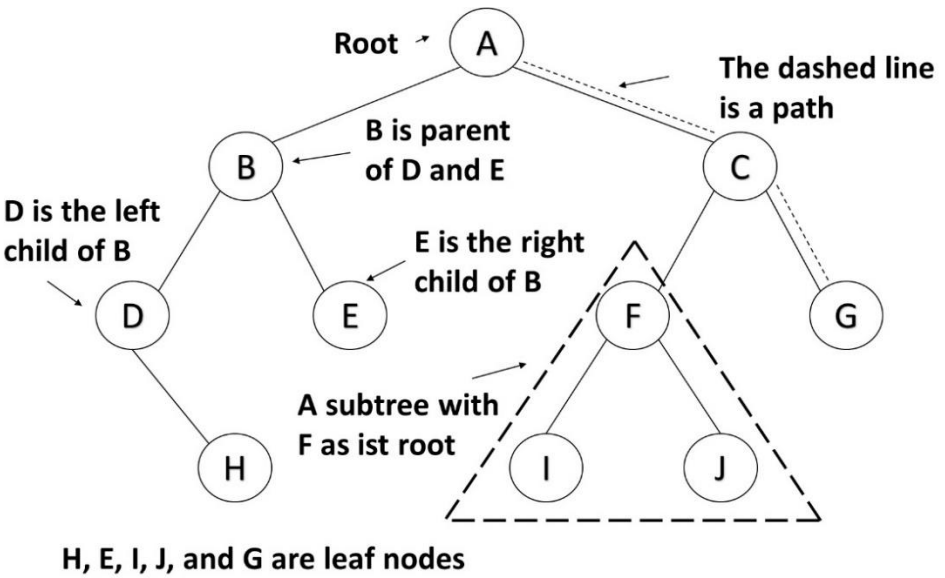

Figure 8.1 Binary Search Tree - terminology

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.

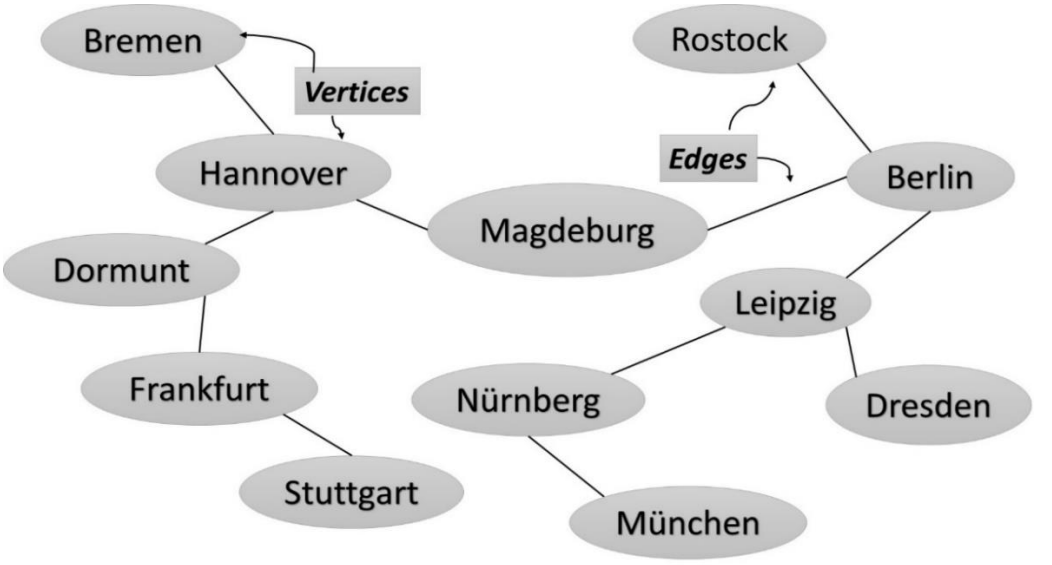

Figure 9.0 Graph - terminology

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

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.

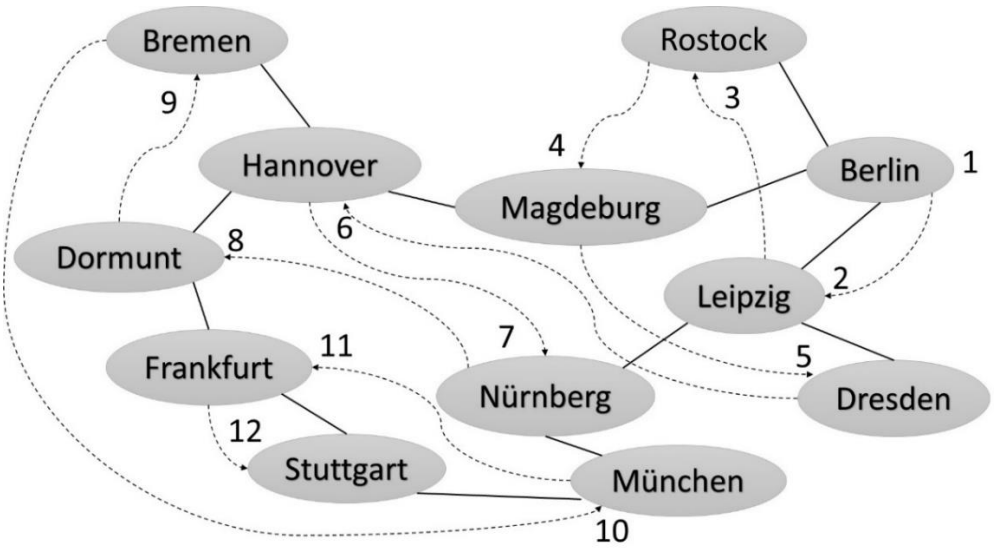

Figure 9.2.1 Breadth-First Search - the sequence of steps

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.

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.

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

```
  @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.

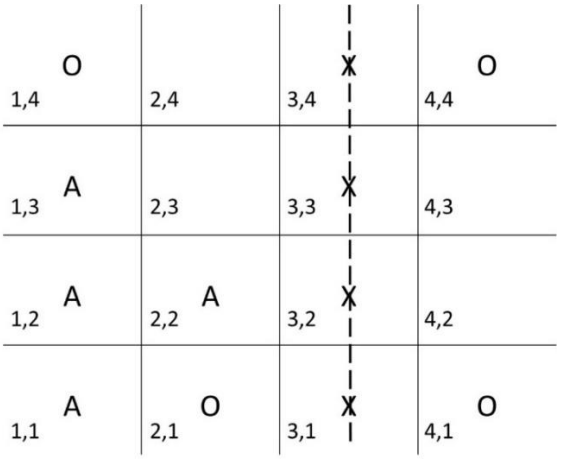

Figure 10.2.1 Tic-tac-toe game

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

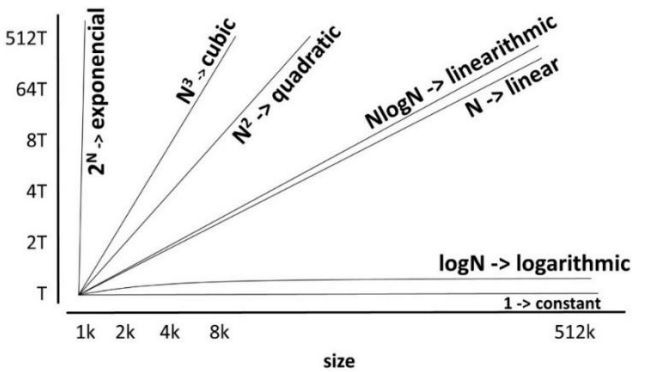

Figure A.1 Big O Notation - order of growth

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");
}
```

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.