JavaScript Repository-to-Pdf

3 Mart 2020

Listing 1: jest.config.js

```
module.exports = {
  // The bail config option can be used here to have Jest stop running tests after
  // the first failure.
  bail: false,

  // Indicates whether each individual test should be reported during the run.
  verbose: false,

  // Indicates whether the coverage information should be collected while executing the test
  collectCoverage: false,

  // The directory where Jest should output its coverage files.
  coverageDirectory: './coverage/',

  // If the test path matches any of the patterns, it will be skipped.
  testPathIgnorePatterns: ['<rootDir>/node_modules/'],

  // If the file path matches any of the patterns, coverage information will be skipped.
  coveragePathIgnorePatterns: ['<rootDir>/node_modules/'],

  // The pattern Jest uses to detect test files.
  testRegex: '(/__tests__/.*|(\\.|/)(test|spec))\\.jsx?$',

  // This option sets the URL for the jsdom environment.
  // It is reflected in properties such as location.href.
  // @see: https://github.com/facebook/jest/issues/6769
  testURL: 'http://localhost/',
};
```

Listing 2: PolynomialHash.test.js

```
import PolynomialHash from '../PolynomialHash';

describe('PolynomialHash', () => {
  it('should calculate new hash based on previous one', () => {
    const bases = [3, 79, 101, 3251, 13229, 122743, 3583213];
    const mods = [79, 101];
    const frameSizes = [5, 20];

    // @TODO: Provide Unicode support.
    const text = 'Lorem Ipsum is simply dummy text of the printing and '
      + 'typesetting industry. Lorem Ipsum has been the industry\'s standard '
      + 'galley of type and \u{ffff} scrambled it to make a type specimen book. It '
      + 'electronic  typesetting, remaining essentially unchanged. It was '
      // + 'popularised in the \u{20005} \u{20000}1960s with the release of Letraset sheets '
      + 'publishing software like Aldus PageMaker  including versions of Lorem.';

    // Check hashing for different prime base.
    bases.forEach((base) => {
      mods.forEach((modulus) => {
        const polynomialHash = new PolynomialHash({ base, modulus });

        // Check hashing for different word lengths.
        frameSizes.forEach((frameSize) => {
          let previousWord = text.substr(0, frameSize);
          let previousHash = polynomialHash.hash(previousWord);

          // Shift frame through the whole text.
          for (let frameShift = 1; frameShift < (text.length - frameSize); frameShift += 1) {
            const currentWord = text.substr(frameShift, frameSize);
            const currentHash = polynomialHash.hash(currentWord);
            const currentRollingHash = polynomialHash.roll(previousHash, previousWord, currentWord);

            // Check that rolling hash is the same as directly calculated hash.
            expect(currentRollingHash).toBe(currentHash);

            previousWord = currentWord;
            previousHash = currentHash;
          }
        });
      });
    });
  });

  it('should generate numeric hashed less than 100', () => {
    const polynomialHash = new PolynomialHash({ modulus: 100 });

    expect(polynomialHash.hash('Some long text that is used as a key')).toBe(41);
    expect(polynomialHash.hash('Test')).toBe(92);
    expect(polynomialHash.hash('a')).toBe(97);
    expect(polynomialHash.hash('b')).toBe(98);
    expect(polynomialHash.hash('c')).toBe(99);
    expect(polynomialHash.hash('d')).toBe(0);
    expect(polynomialHash.hash('e')).toBe(1);
    expect(polynomialHash.hash('ab')).toBe(87);

    // @TODO: Provide Unicode support.
    expect(polynomialHash.hash('\u{20000}')).toBe(92);
  });
});
```

Listing 3: SimplePolynomialHash.test.js

```
import SimplePolynomialHash from '../SimplePolynomialHash';

describe('PolynomialHash', () => {
  it('should calculate new hash based on previous one', () => {
    const bases = [3, 5];
    const frameSizes = [5, 10];

    const text = 'Lorem Ipsum is simply dummy text of the printing and '
      + 'typesetting industry. Lorem Ipsum has been the industry\'s standard '
      + 'galley of type and \u{ffff} scrambled it to make a type specimen book. It '
      + 'electronic  typesetting, remaining essentially unchanged. It was '
      + 'popularised in the 1960s with the release of Letraset sheets '
      + 'publishing software like Aldus  PageMaker including versions of Lorem.';

    // Check hashing for different prime base.
    bases.forEach((base) => {
      const polynomialHash = new SimplePolynomialHash(base);

      // Check hashing for different word lengths.
      frameSizes.forEach((frameSize) => {
        let previousWord = text.substr(0, frameSize);
        let previousHash = polynomialHash.hash(previousWord);

        // Shift frame through the whole text.
        for (let frameShift = 1; frameShift < (text.length - frameSize); frameShift += 1) {
          const currentWord = text.substr(frameShift, frameSize);
          const currentHash = polynomialHash.hash(currentWord);
          const currentRollingHash = polynomialHash.roll(previousHash, previousWord, currentWord);

          // Check that rolling hash is the same as directly calculated hash.
          expect(currentRollingHash).toBe(currentHash);

          previousWord = currentWord;
          previousHash = currentHash;
        }
      });
    });
  });

  it('should generate numeric hashed', () => {
    const polynomialHash = new SimplePolynomialHash();

    expect(polynomialHash.hash('Test')).toBe(604944);
    expect(polynomialHash.hash('a')).toBe(97);
    expect(polynomialHash.hash('b')).toBe(98);
    expect(polynomialHash.hash('c')).toBe(99);
    expect(polynomialHash.hash('d')).toBe(100);
    expect(polynomialHash.hash('e')).toBe(101);
    expect(polynomialHash.hash('ab')).toBe(1763);
    expect(polynomialHash.hash('abc')).toBe(30374);
  });
});
```

Listing 4: PolynomialHash.js

```
const DEFAULT_BASE = 37;
const DEFAULT_MODULUS = 101;

export default class PolynomialHash {
  /**
   * @param {number} [base] - Base number that is used to create the polynomial.
   * @param {number} [modulus] - Modulus number that keeps the hash from overflowing.
   */
  constructor({ base = DEFAULT_BASE, modulus = DEFAULT_MODULUS } = {}) {
    this.base = base;
    this.modulus = modulus;
  }

  /**
   * Function that creates hash representation of the word.
   *
   * Time complexity: O(word.length).
   *
   * @param {string} word - String that needs to be hashed.
   * @return {number}
   */
  hash(word) {
    const charCodes = Array.from(word).map(char => this.charToNumber(char));

    let hash = 0;
    for (let charIndex = 0; charIndex < charCodes.length; charIndex += 1) {
      hash *= this.base;
      hash += charCodes[charIndex];
      hash %= this.modulus;
    }

    return hash;
  }

  /**
   * Function that creates hash representation of the word
   * based on previous word (shifted by one character left) hash value.
   *
   * Recalculates the hash representation of a word so that it isn't
   * necessary to traverse the whole word again.
   *
   * Time complexity: O(1).
   *
   * @param {number} prevHash
   * @param {string} prevWord
   * @param {string} newWord
   * @return {number}
   */
  roll(prevHash, prevWord, newWord) {
    let hash = prevHash;

    const prevValue = this.charToNumber(prevWord[0]);
    const newValue = this.charToNumber(newWord[newWord.length - 1]);

    let prevValueMultiplier = 1;
    for (let i = 1; i < prevWord.length; i += 1) {
      prevValueMultiplier *= this.base;
      prevValueMultiplier %= this.modulus;
    }

    hash += this.modulus;
    hash -= (prevValue * prevValueMultiplier) % this.modulus;

    hash *= this.base;
    hash += newValue;
    hash %= this.modulus;

    return hash;
  }

  /**
   * Converts char to number.
   *
   * @param {string} char
   * @return {number}
   */
  charToNumber(char) {
    let charCode = char.codePointAt(0);
```

```
    // Check if character has surrogate pair.
    const surrogate = char.codePointAt(1);
    if (surrogate !== undefined) {
      const surrogateShift = 2 ** 16;
      charCode += surrogate * surrogateShift;
    }

    return charCode;
  }
}
```

Listing 5: SimplePolynomialHash.js

```
const DEFAULT_BASE = 17;

export default class SimplePolynomialHash {
  /**
   * @param {number} [base] - Base number that is used to create the polynomial.
   */
  constructor(base = DEFAULT_BASE) {
    this.base = base;
  }

  /**
   * Function that creates hash representation of the word.
   *
   * Time complexity: O(word.length).
   *
   * @assumption: This version of the function  doesn't use modulo operator.
   * Thus it may produce number overflows by generating numbers that are
   * bigger than Number.MAX_SAFE_INTEGER. This function is mentioned here
   * for simplicity and LEARNING reasons.
   *
   * @param {string} word - String that needs to be hashed.
   * @return {number}
   */
  hash(word) {
    let hash = 0;
    for (let charIndex = 0; charIndex < word.length; charIndex += 1) {
      hash += word.charCodeAt(charIndex) * (this.base ** charIndex);
    }

    return hash;
  }

  /**
   * Function that creates hash representation of the word
   * based on previous word (shifted by one character left) hash value.
   *
   * Recalculates the hash representation of a word so that it isn't
   * necessary to traverse the whole word again.
   *
   * Time complexity: O(1).
   *
   * @assumption: This function doesn't use modulo operator and thus is not safe since
   * it may deal with numbers that are bigger than Number.MAX_SAFE_INTEGER. This
   * function is mentioned here for simplicity and LEARNING reasons.
   *
   * @param {number} prevHash
   * @param {string} prevWord
   * @param {string} newWord
   * @return {number}
   */
  roll(prevHash, prevWord, newWord) {
    let hash = prevHash;

    const prevValue = prevWord.charCodeAt(0);
    const newValue = newWord.charCodeAt(newWord.length - 1);

    hash -= prevValue;
    hash /= this.base;
    hash += newValue * (this.base ** (newWord.length - 1));

    return hash;
  }
}
```

Listing 6: articulationPoints.test.js

```
import GraphVertex from '../../../../data-structures/graph/GraphVertex';
import GraphEdge from '../../../../data-structures/graph/GraphEdge';
import Graph from '../../../../data-structures/graph/Graph';
import articulationPoints from '../articulationPoints';

describe('articulationPoints', () => {
  it('should find articulation points in simple graph', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeCD = new GraphEdge(vertexC, vertexD);

    const graph = new Graph();

    graph
      .addEdge(edgeAB)
      .addEdge(edgeBC)
      .addEdge(edgeCD);

    const articulationPointsSet = Object.values(articulationPoints(graph));

    expect(articulationPointsSet.length).toBe(2);
    expect(articulationPointsSet[0].getKey()).toBe(vertexC.getKey());
    expect(articulationPointsSet[1].getKey()).toBe(vertexB.getKey());
  });

  it('should find articulation points in simple graph with back edge', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeCD = new GraphEdge(vertexC, vertexD);
    const edgeAC = new GraphEdge(vertexA, vertexC);

    const graph = new Graph();

    graph
      .addEdge(edgeAB)
      .addEdge(edgeAC)
      .addEdge(edgeBC)
      .addEdge(edgeCD);

    const articulationPointsSet = Object.values(articulationPoints(graph));

    expect(articulationPointsSet.length).toBe(1);
    expect(articulationPointsSet[0].getKey()).toBe(vertexC.getKey());
  });

  it('should find articulation points in simple graph with back edge #2', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeCD = new GraphEdge(vertexC, vertexD);
    const edgeAE = new GraphEdge(vertexA, vertexE);
    const edgeCE = new GraphEdge(vertexC, vertexE);

    const graph = new Graph();

    graph
      .addEdge(edgeAB)
      .addEdge(edgeAE)
      .addEdge(edgeCE)
      .addEdge(edgeBC)
      .addEdge(edgeCD);

    const articulationPointsSet = Object.values(articulationPoints(graph));
```

```
    expect(articulationPointsSet.length).toBe(1);
    expect(articulationPointsSet[0].getKey()).toBe(vertexC.getKey());
  });

  it('should find articulation points in graph', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');
    const vertexF = new GraphVertex('F');
    const vertexG = new GraphVertex('G');
    const vertexH = new GraphVertex('H');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeAC = new GraphEdge(vertexA, vertexC);
    const edgeCD = new GraphEdge(vertexC, vertexD);
    const edgeDE = new GraphEdge(vertexD, vertexE);
    const edgeEG = new GraphEdge(vertexE, vertexG);
    const edgeEF = new GraphEdge(vertexE, vertexF);
    const edgeGF = new GraphEdge(vertexG, vertexF);
    const edgeFH = new GraphEdge(vertexF, vertexH);

    const graph = new Graph();

    graph
      .addEdge(edgeAB)
      .addEdge(edgeBC)
      .addEdge(edgeAC)
      .addEdge(edgeCD)
      .addEdge(edgeDE)
      .addEdge(edgeEG)
      .addEdge(edgeEF)
      .addEdge(edgeGF)
      .addEdge(edgeFH);

    const articulationPointsSet = Object.values(articulationPoints(graph));

    expect(articulationPointsSet.length).toBe(4);
    expect(articulationPointsSet[0].getKey()).toBe(vertexF.getKey());
    expect(articulationPointsSet[1].getKey()).toBe(vertexE.getKey());
    expect(articulationPointsSet[2].getKey()).toBe(vertexD.getKey());
    expect(articulationPointsSet[3].getKey()).toBe(vertexC.getKey());
  });

  it('should find articulation points in graph starting with articulation root vertex', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');
    const vertexF = new GraphVertex('F');
    const vertexG = new GraphVertex('G');
    const vertexH = new GraphVertex('H');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeAC = new GraphEdge(vertexA, vertexC);
    const edgeCD = new GraphEdge(vertexC, vertexD);
    const edgeDE = new GraphEdge(vertexD, vertexE);
    const edgeEG = new GraphEdge(vertexE, vertexG);
    const edgeEF = new GraphEdge(vertexE, vertexF);
    const edgeGF = new GraphEdge(vertexG, vertexF);
    const edgeFH = new GraphEdge(vertexF, vertexH);

    const graph = new Graph();

    graph
      .addEdge(edgeDE)
      .addEdge(edgeAB)
      .addEdge(edgeBC)
      .addEdge(edgeAC)
      .addEdge(edgeCD)
      .addEdge(edgeEG)
      .addEdge(edgeEF)
      .addEdge(edgeGF)
      .addEdge(edgeFH);

    const articulationPointsSet = Object.values(articulationPoints(graph));

    expect(articulationPointsSet.length).toBe(4);
```

```
    expect(articulationPointsSet[0].getKey()).toBe(vertexF.getKey());
    expect(articulationPointsSet[1].getKey()).toBe(vertexE.getKey());
    expect(articulationPointsSet[2].getKey()).toBe(vertexC.getKey());
    expect(articulationPointsSet[3].getKey()).toBe(vertexD.getKey());
  });

  it('should find articulation points in yet another graph #1', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeAC = new GraphEdge(vertexA, vertexC);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeCD = new GraphEdge(vertexC, vertexD);
    const edgeDE = new GraphEdge(vertexD, vertexE);

    const graph = new Graph();

    graph
      .addEdge(edgeAB)
      .addEdge(edgeAC)
      .addEdge(edgeBC)
      .addEdge(edgeCD)
      .addEdge(edgeDE);

    const articulationPointsSet = Object.values(articulationPoints(graph));

    expect(articulationPointsSet.length).toBe(2);
    expect(articulationPointsSet[0].getKey()).toBe(vertexD.getKey());
    expect(articulationPointsSet[1].getKey()).toBe(vertexC.getKey());
  });

  it('should find articulation points in yet another graph #2', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');
    const vertexF = new GraphVertex('F');
    const vertexG = new GraphVertex('G');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeAC = new GraphEdge(vertexA, vertexC);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeCD = new GraphEdge(vertexC, vertexD);
    const edgeCE = new GraphEdge(vertexC, vertexE);
    const edgeCF = new GraphEdge(vertexC, vertexF);
    const edgeEG = new GraphEdge(vertexE, vertexG);
    const edgeFG = new GraphEdge(vertexF, vertexG);

    const graph = new Graph();

    graph
      .addEdge(edgeAB)
      .addEdge(edgeAC)
      .addEdge(edgeBC)
      .addEdge(edgeCD)
      .addEdge(edgeCE)
      .addEdge(edgeCF)
      .addEdge(edgeEG)
      .addEdge(edgeFG);

    const articulationPointsSet = Object.values(articulationPoints(graph));

    expect(articulationPointsSet.length).toBe(1);
    expect(articulationPointsSet[0].getKey()).toBe(vertexC.getKey());
  });
});
```

Listing 7: articulationPoints.js

```
import depthFirstSearch from '../depth-first-search/depthFirstSearch';

/**
 * Helper class for visited vertex metadata.
 */
class VisitMetadata {
  constructor({ discoveryTime, lowDiscoveryTime }) {
    this.discoveryTime = discoveryTime;
    this.lowDiscoveryTime = lowDiscoveryTime;
    // We need this in order to check graph root node, whether it has two
    // disconnected children or not.
    this.independentChildrenCount = 0;
  }
}

/**
 * Tarjan's algorithm for finding articulation points in graph.
 *
 * @param {Graph} graph
 * @return {Object}
 */
export default function articulationPoints(graph) {
  // Set of vertices we've already visited during DFS.
  const visitedSet = {};

  // Set of articulation points.
  const articulationPointsSet = {};

  // Time needed to discover to the current vertex.
  let discoveryTime = 0;

  // Peek the start vertex for DFS traversal.
  const startVertex = graph.getAllVertices()[0];

  const dfsCallbacks = {
    /**
     * @param {GraphVertex} currentVertex
     * @param {GraphVertex} previousVertex
     */
    enterVertex: ({ currentVertex, previousVertex }) => {
      // Tick discovery time.
      discoveryTime += 1;

      // Put current vertex to visited set.
      visitedSet[currentVertex.getKey()] = new VisitMetadata({
        discoveryTime,
        lowDiscoveryTime: discoveryTime,
      });

      if (previousVertex) {
        // Update children counter for previous vertex.
        visitedSet[previousVertex.getKey()].independentChildrenCount += 1;
      }
    },
    /**
     * @param {GraphVertex} currentVertex
     * @param {GraphVertex} previousVertex
     */
    leaveVertex: ({ currentVertex, previousVertex }) => {
      if (previousVertex === null) {
        // Don't do anything for the root vertex if it is already current (not previous one)
        return;
      }

      // Update the low time with the smallest time of adjacent vertices.
      // Get minimum low discovery time from all neighbors.
      /** @param {GraphVertex} neighbor */
      visitedSet[currentVertex.getKey()].lowDiscoveryTime = currentVertex.getNeighbors()
        .filter(earlyNeighbor => earlyNeighbor.getKey() !== previousVertex.getKey())
        /**
         * @param {number} lowestDiscoveryTime
         * @param {GraphVertex} neighbor
         */
        .reduce(
          (lowestDiscoveryTime, neighbor) => {
            const neighborLowTime = visitedSet[neighbor.getKey()].lowDiscoveryTime;
            return neighborLowTime < lowestDiscoveryTime ? neighborLowTime : lowestDiscoveryTime;
          },
          visitedSet[currentVertex.getKey()].lowDiscoveryTime,
```

```
        );

      // Detect whether previous vertex is articulation point or not.
      // To do so we need to check two [OR] conditions:
      // 1. Is it a root vertex with at least two independent children.
      // 2. If its visited time is <= low time of adjacent vertex.
      if (previousVertex === startVertex) {
        // Check that root vertex has at least two independent children.
        if (visitedSet[previousVertex.getKey()].independentChildrenCount >= 2) {
          articulationPointsSet[previousVertex.getKey()] = previousVertex;
        }
      } else {
        // Get current vertex low discovery time.
        const currentLowDiscoveryTime = visitedSet[currentVertex.getKey()].lowDiscoveryTime;

        // Compare current vertex low discovery time with parent discovery time. Check if there
        // are any short path (back edge) exists. If we can't get to current vertex other then
        // via parent then the parent vertex is articulation point for current one.
        const parentDiscoveryTime = visitedSet[previousVertex.getKey()].discoveryTime;
        if (parentDiscoveryTime <= currentLowDiscoveryTime) {
          articulationPointsSet[previousVertex.getKey()] = previousVertex;
        }
      }
    },
    allowTraversal: ({ nextVertex }) => {
      return !visitedSet[nextVertex.getKey()];
    },
  };

  // Do Depth First Search traversal over submitted graph.
  depthFirstSearch(graph, startVertex, dfsCallbacks);

  return articulationPointsSet;
}
```

Listing 8: bellmanFord.test.js

```
import GraphVertex from '../../../../data-structures/graph/GraphVertex';
import GraphEdge from '../../../../data-structures/graph/GraphEdge';
import Graph from '../../../../data-structures/graph/Graph';
import bellmanFord from '../bellmanFord';

describe('bellmanFord', () => {
  it('should find minimum paths to all vertices for undirected graph', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');
    const vertexF = new GraphVertex('F');
    const vertexG = new GraphVertex('G');
    const vertexH = new GraphVertex('H');

    const edgeAB = new GraphEdge(vertexA, vertexB, 4);
    const edgeAE = new GraphEdge(vertexA, vertexE, 7);
    const edgeAC = new GraphEdge(vertexA, vertexC, 3);
    const edgeBC = new GraphEdge(vertexB, vertexC, 6);
    const edgeBD = new GraphEdge(vertexB, vertexD, 5);
    const edgeEC = new GraphEdge(vertexE, vertexC, 8);
    const edgeED = new GraphEdge(vertexE, vertexD, 2);
    const edgeDC = new GraphEdge(vertexD, vertexC, 11);
    const edgeDG = new GraphEdge(vertexD, vertexG, 10);
    const edgeDF = new GraphEdge(vertexD, vertexF, 2);
    const edgeFG = new GraphEdge(vertexF, vertexG, 3);
    const edgeEG = new GraphEdge(vertexE, vertexG, 5);

    const graph = new Graph();
    graph
      .addVertex(vertexH)
      .addEdge(edgeAB)
      .addEdge(edgeAE)
      .addEdge(edgeAC)
      .addEdge(edgeBC)
      .addEdge(edgeBD)
      .addEdge(edgeEC)
      .addEdge(edgeED)
      .addEdge(edgeDC)
      .addEdge(edgeDG)
      .addEdge(edgeDF)
      .addEdge(edgeFG)
      .addEdge(edgeEG);

    const { distances, previousVertices } = bellmanFord(graph, vertexA);

    expect(distances).toEqual({
      H: Infinity,
      A: 0,
      B: 4,
      E: 7,
      C: 3,
      D: 9,
      G: 12,
      F: 11,
    });

    expect(previousVertices.F.getKey()).toBe('D');
    expect(previousVertices.D.getKey()).toBe('B');
    expect(previousVertices.B.getKey()).toBe('A');
    expect(previousVertices.G.getKey()).toBe('E');
    expect(previousVertices.C.getKey()).toBe('A');
    expect(previousVertices.A).toBeNull();
    expect(previousVertices.H).toBeNull();
  });

  it('should find minimum paths to all vertices for directed graph with negative edge weights', () => {
    const vertexS = new GraphVertex('S');
    const vertexE = new GraphVertex('E');
    const vertexA = new GraphVertex('A');
    const vertexD = new GraphVertex('D');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexH = new GraphVertex('H');

    const edgeSE = new GraphEdge(vertexS, vertexE, 8);
    const edgeSA = new GraphEdge(vertexS, vertexA, 10);
    const edgeED = new GraphEdge(vertexE, vertexD, 1);
```

```
    const edgeDA = new GraphEdge(vertexD, vertexA, -4);
    const edgeDC = new GraphEdge(vertexD, vertexC, -1);
    const edgeAC = new GraphEdge(vertexA, vertexC, 2);
    const edgeCB = new GraphEdge(vertexC, vertexB, -2);
    const edgeBA = new GraphEdge(vertexB, vertexA, 1);

    const graph = new Graph(true);
    graph
      .addVertex(vertexH)
      .addEdge(edgeSE)
      .addEdge(edgeSA)
      .addEdge(edgeED)
      .addEdge(edgeDA)
      .addEdge(edgeDC)
      .addEdge(edgeAC)
      .addEdge(edgeCB)
      .addEdge(edgeBA);

    const { distances, previousVertices } = bellmanFord(graph, vertexS);

    expect(distances).toEqual({
      H: Infinity,
      S: 0,
      A: 5,
      B: 5,
      C: 7,
      D: 9,
      E: 8,
    });

    expect(previousVertices.H).toBeNull();
    expect(previousVertices.S).toBeNull();
    expect(previousVertices.B.getKey()).toBe('C');
    expect(previousVertices.C.getKey()).toBe('A');
    expect(previousVertices.A.getKey()).toBe('D');
    expect(previousVertices.D.getKey()).toBe('E');
  });
});
```

Listing 9: bellmanFord.js

```
/**
* @param {Graph} graph
* @param {GraphVertex} startVertex
* @return {{distances, previousVertices}}
*/
export default function bellmanFord(graph, startVertex) {
  const distances = {};
  const previousVertices = {};

  // Init all distances with infinity assuming that currently we can't reach
  // any of the vertices except start one.
  distances[startVertex.getKey()] = 0;
  graph.getAllVertices().forEach((vertex) => {
    previousVertices[vertex.getKey()] = null;
    if (vertex.getKey() !== startVertex.getKey()) {
      distances[vertex.getKey()] = Infinity;
    }
  });

  // We need (|V| - 1) iterations.
  for (let iteration = 0; iteration < (graph.getAllVertices().length - 1); iteration += 1) {
    // During each iteration go through all vertices.
    Object.keys(distances).forEach((vertexKey) => {
      const vertex = graph.getVertexByKey(vertexKey);

      // Go through all vertex edges.
      graph.getNeighbors(vertex).forEach((neighbor) => {
        const edge = graph.findEdge(vertex, neighbor);
        // Find out if the distance to the neighbor is shorter in this iteration
        // then in previous one.
        const distanceToVertex = distances[vertex.getKey()];
        const distanceToNeighbor = distanceToVertex + edge.weight;
        if (distanceToNeighbor < distances[neighbor.getKey()]) {
          distances[neighbor.getKey()] = distanceToNeighbor;
          previousVertices[neighbor.getKey()] = vertex;
        }
      });
    });
  }

  return {
    distances,
    previousVertices,
  };
}
```

Listing 10: breadthFirstSearch.test.js

```
 import Graph from '../../../../data-structures/graph/Graph';
import GraphVertex from '../../../../data-structures/graph/GraphVertex';
import GraphEdge from '../../../../data-structures/graph/GraphEdge';
import breadthFirstSearch from '../breadthFirstSearch';

describe('breadthFirstSearch', () => {
  it('should perform BFS operation on graph', () => {
    const graph = new Graph(true);

    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');
    const vertexF = new GraphVertex('F');
    const vertexG = new GraphVertex('G');
    const vertexH = new GraphVertex('H');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeCG = new GraphEdge(vertexC, vertexG);
    const edgeAD = new GraphEdge(vertexA, vertexD);
    const edgeAE = new GraphEdge(vertexA, vertexE);
    const edgeEF = new GraphEdge(vertexE, vertexF);
    const edgeFD = new GraphEdge(vertexF, vertexD);
    const edgeDH = new GraphEdge(vertexD, vertexH);
    const edgeGH = new GraphEdge(vertexG, vertexH);

    graph
      .addEdge(edgeAB)
      .addEdge(edgeBC)
      .addEdge(edgeCG)
      .addEdge(edgeAD)
      .addEdge(edgeAE)
      .addEdge(edgeEF)
      .addEdge(edgeFD)
      .addEdge(edgeDH)
      .addEdge(edgeGH);

    expect(graph.toString()).toBe('A,B,C,G,D,E,F,H');

    const enterVertexCallback = jest.fn();
    const leaveVertexCallback = jest.fn();

    // Traverse graphs without callbacks first.
    breadthFirstSearch(graph, vertexA);

    // Traverse graph with enterVertex and leaveVertex callbacks.
    breadthFirstSearch(graph, vertexA, {
      enterVertex: enterVertexCallback,
      leaveVertex: leaveVertexCallback,
    });

    expect(enterVertexCallback).toHaveBeenCalledTimes(8);
    expect(leaveVertexCallback).toHaveBeenCalledTimes(8);

    const enterVertexParamsMap = [
      { currentVertex: vertexA, previousVertex: null },
      { currentVertex: vertexB, previousVertex: vertexA },
      { currentVertex: vertexD, previousVertex: vertexB },
      { currentVertex: vertexE, previousVertex: vertexD },
      { currentVertex: vertexC, previousVertex: vertexE },
      { currentVertex: vertexH, previousVertex: vertexC },
      { currentVertex: vertexF, previousVertex: vertexH },
      { currentVertex: vertexG, previousVertex: vertexF },
    ];

    for (let callIndex = 0; callIndex < graph.getAllVertices().length; callIndex += 1) {
      const params = enterVertexCallback.mock.calls[callIndex][0];
      expect(params.currentVertex).toEqual(enterVertexParamsMap[callIndex].currentVertex);
      expect(params.previousVertex).toEqual(enterVertexParamsMap[callIndex].previousVertex);
    }

    const leaveVertexParamsMap = [
      { currentVertex: vertexA, previousVertex: null },
      { currentVertex: vertexB, previousVertex: vertexA },
      { currentVertex: vertexD, previousVertex: vertexB },
      { currentVertex: vertexE, previousVertex: vertexD },
      { currentVertex: vertexC, previousVertex: vertexE },
```

```
    { currentVertex: vertexH, previousVertex: vertexC },
    { currentVertex: vertexF, previousVertex: vertexH },
    { currentVertex: vertexG, previousVertex: vertexF },
  ];

  for (let callIndex = 0; callIndex < graph.getAllVertices().length; callIndex += 1) {
    const params = leaveVertexCallback.mock.calls[callIndex][0];
    expect(params.currentVertex).toEqual(leaveVertexParamsMap[callIndex].currentVertex);
    expect(params.previousVertex).toEqual(leaveVertexParamsMap[callIndex].previousVertex);
  }
});

it('should allow to create custom vertex visiting logic', () => {
  const graph = new Graph(true);

  const vertexA = new GraphVertex('A');
  const vertexB = new GraphVertex('B');
  const vertexC = new GraphVertex('C');
  const vertexD = new GraphVertex('D');
  const vertexE = new GraphVertex('E');
  const vertexF = new GraphVertex('F');
  const vertexG = new GraphVertex('G');
  const vertexH = new GraphVertex('H');

  const edgeAB = new GraphEdge(vertexA, vertexB);
  const edgeBC = new GraphEdge(vertexB, vertexC);
  const edgeCG = new GraphEdge(vertexC, vertexG);
  const edgeAD = new GraphEdge(vertexA, vertexD);
  const edgeAE = new GraphEdge(vertexA, vertexE);
  const edgeEF = new GraphEdge(vertexE, vertexF);
  const edgeFD = new GraphEdge(vertexF, vertexD);
  const edgeDH = new GraphEdge(vertexD, vertexH);
  const edgeGH = new GraphEdge(vertexG, vertexH);

  graph
    .addEdge(edgeAB)
    .addEdge(edgeBC)
    .addEdge(edgeCG)
    .addEdge(edgeAD)
    .addEdge(edgeAE)
    .addEdge(edgeEF)
    .addEdge(edgeFD)
    .addEdge(edgeDH)
    .addEdge(edgeGH);

  expect(graph.toString()).toBe('A,B,C,G,D,E,F,H');

  const enterVertexCallback = jest.fn();
  const leaveVertexCallback = jest.fn();

  // Traverse graph with enterVertex and leaveVertex callbacks.
  breadthFirstSearch(graph, vertexA, {
    enterVertex: enterVertexCallback,
    leaveVertex: leaveVertexCallback,
    allowTraversal: ({ currentVertex, nextVertex }) => {
      return !(currentVertex === vertexA && nextVertex === vertexB);
    },
  });

  expect(enterVertexCallback).toHaveBeenCalledTimes(7);
  expect(leaveVertexCallback).toHaveBeenCalledTimes(7);

  const enterVertexParamsMap = [
    { currentVertex: vertexA, previousVertex: null },
    { currentVertex: vertexD, previousVertex: vertexA },
    { currentVertex: vertexE, previousVertex: vertexD },
    { currentVertex: vertexH, previousVertex: vertexE },
    { currentVertex: vertexF, previousVertex: vertexH },
    { currentVertex: vertexD, previousVertex: vertexF },
    { currentVertex: vertexH, previousVertex: vertexD },
  ];

  for (let callIndex = 0; callIndex < 7; callIndex += 1) {
    const params = enterVertexCallback.mock.calls[callIndex][0];
    expect(params.currentVertex).toEqual(enterVertexParamsMap[callIndex].currentVertex);
    expect(params.previousVertex).toEqual(enterVertexParamsMap[callIndex].previousVertex);
  }

  const leaveVertexParamsMap = [
    { currentVertex: vertexA, previousVertex: null },
    { currentVertex: vertexD, previousVertex: vertexA },
    { currentVertex: vertexE, previousVertex: vertexD },
```

```
      { currentVertex: vertexH, previousVertex: vertexE },
      { currentVertex: vertexF, previousVertex: vertexH },
      { currentVertex: vertexD, previousVertex: vertexF },
      { currentVertex: vertexH, previousVertex: vertexD },
    ];

    for (let callIndex = 0; callIndex < 7; callIndex += 1) {
      const params = leaveVertexCallback.mock.calls[callIndex][0];
      expect(params.currentVertex).toEqual(leaveVertexParamsMap[callIndex].currentVertex);
      expect(params.previousVertex).toEqual(leaveVertexParamsMap[callIndex].previousVertex);
    }
  });
});
```

Listing 11: breadthFirstSearch.js

```
import Queue from '../../../data-structures/queue/Queue';

/**
 * @typedef {Object} Callbacks
 *
 * @property {function(vertices: Object): boolean} [allowTraversal] -
 *   Determines whether DFS should traverse from the vertex to its neighbor
 *   (along the edge). By default prohibits visiting the same vertex again.
 *
 * @property {function(vertices: Object)} [enterVertex] - Called when BFS enters the vertex.
 *
 * @property {function(vertices: Object)} [leaveVertex] - Called when BFS leaves the vertex.
 */

/**
 * @param {Callbacks} [callbacks]
 * @returns {Callbacks}
 */
function initCallbacks(callbacks = {}) {
  const initiatedCallback = callbacks;

  const stubCallback = () => {};

  const allowTraversalCallback = (
    () => {
      const seen = {};
      return ({ nextVertex }) => {
        if (!seen[nextVertex.getKey()]) {
          seen[nextVertex.getKey()] = true;
          return true;
        }
        return false;
      };
    }
  )();

  initiatedCallback.allowTraversal = callbacks.allowTraversal || allowTraversalCallback;
  initiatedCallback.enterVertex = callbacks.enterVertex || stubCallback;
  initiatedCallback.leaveVertex = callbacks.leaveVertex || stubCallback;

  return initiatedCallback;
}

/**
 * @param {Graph} graph
 * @param {GraphVertex} startVertex
 * @param {Callbacks} [originalCallbacks]
 */
export default function breadthFirstSearch(graph, startVertex, originalCallbacks) {
  const callbacks = initCallbacks(originalCallbacks);
  const vertexQueue = new Queue();

  // Do initial queue setup.
  vertexQueue.enqueue(startVertex);

  let previousVertex = null;

  // Traverse all vertices from the queue.
  while (!vertexQueue.isEmpty()) {
    const currentVertex = vertexQueue.dequeue();
    callbacks.enterVertex({ currentVertex, previousVertex });

    // Add all neighbors to the queue for future traversals.
    graph.getNeighbors(currentVertex).forEach((nextVertex) => {
      if (callbacks.allowTraversal({ previousVertex, currentVertex, nextVertex })) {
        vertexQueue.enqueue(nextVertex);
      }
    });

    callbacks.leaveVertex({ currentVertex, previousVertex });

    // Memorize current vertex before next loop.
    previousVertex = currentVertex;
  }
}
```

Listing 12: graphBridges.test.js

```
import GraphVertex from '../../../../data-structures/graph/GraphVertex';
import GraphEdge from '../../../../data-structures/graph/GraphEdge';
import Graph from '../../../../data-structures/graph/Graph';
import graphBridges from '../graphBridges';

describe('graphBridges', () => {
  it('should find bridges in simple graph', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeCD = new GraphEdge(vertexC, vertexD);

    const graph = new Graph();

    graph
      .addEdge(edgeAB)
      .addEdge(edgeBC)
      .addEdge(edgeCD);

    const bridges = Object.values(graphBridges(graph));

    expect(bridges.length).toBe(3);
    expect(bridges[0].getKey()).toBe(edgeCD.getKey());
    expect(bridges[1].getKey()).toBe(edgeBC.getKey());
    expect(bridges[2].getKey()).toBe(edgeAB.getKey());
  });

  it('should find bridges in simple graph with back edge', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeCD = new GraphEdge(vertexC, vertexD);
    const edgeAC = new GraphEdge(vertexA, vertexC);

    const graph = new Graph();

    graph
      .addEdge(edgeAB)
      .addEdge(edgeAC)
      .addEdge(edgeBC)
      .addEdge(edgeCD);

    const bridges = Object.values(graphBridges(graph));

    expect(bridges.length).toBe(1);
    expect(bridges[0].getKey()).toBe(edgeCD.getKey());
  });

  it('should find bridges in graph', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');
    const vertexF = new GraphVertex('F');
    const vertexG = new GraphVertex('G');
    const vertexH = new GraphVertex('H');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeAC = new GraphEdge(vertexA, vertexC);
    const edgeCD = new GraphEdge(vertexC, vertexD);
    const edgeDE = new GraphEdge(vertexD, vertexE);
    const edgeEG = new GraphEdge(vertexE, vertexG);
    const edgeEF = new GraphEdge(vertexE, vertexF);
    const edgeGF = new GraphEdge(vertexG, vertexF);
    const edgeFH = new GraphEdge(vertexF, vertexH);

    const graph = new Graph();

    graph
```

```
      .addEdge(edgeAB)
      .addEdge(edgeBC)
      .addEdge(edgeAC)
      .addEdge(edgeCD)
      .addEdge(edgeDE)
      .addEdge(edgeEG)
      .addEdge(edgeEF)
      .addEdge(edgeGF)
      .addEdge(edgeFH);

    const bridges = Object.values(graphBridges(graph));

    expect(bridges.length).toBe(3);
    expect(bridges[0].getKey()).toBe(edgeFH.getKey());
    expect(bridges[1].getKey()).toBe(edgeDE.getKey());
    expect(bridges[2].getKey()).toBe(edgeCD.getKey());
  });

  it('should find bridges in graph starting with different root vertex', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');
    const vertexF = new GraphVertex('F');
    const vertexG = new GraphVertex('G');
    const vertexH = new GraphVertex('H');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeAC = new GraphEdge(vertexA, vertexC);
    const edgeCD = new GraphEdge(vertexC, vertexD);
    const edgeDE = new GraphEdge(vertexD, vertexE);
    const edgeEG = new GraphEdge(vertexE, vertexG);
    const edgeEF = new GraphEdge(vertexE, vertexF);
    const edgeGF = new GraphEdge(vertexG, vertexF);
    const edgeFH = new GraphEdge(vertexF, vertexH);

    const graph = new Graph();

    graph
      .addEdge(edgeDE)
      .addEdge(edgeAB)
      .addEdge(edgeBC)
      .addEdge(edgeAC)
      .addEdge(edgeCD)
      .addEdge(edgeEG)
      .addEdge(edgeEF)
      .addEdge(edgeGF)
      .addEdge(edgeFH);

    const bridges = Object.values(graphBridges(graph));

    expect(bridges.length).toBe(3);
    expect(bridges[0].getKey()).toBe(edgeFH.getKey());
    expect(bridges[1].getKey()).toBe(edgeDE.getKey());
    expect(bridges[2].getKey()).toBe(edgeCD.getKey());
  });

  it('should find bridges in yet another graph #1', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeAC = new GraphEdge(vertexA, vertexC);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeCD = new GraphEdge(vertexC, vertexD);
    const edgeDE = new GraphEdge(vertexD, vertexE);

    const graph = new Graph();

    graph
      .addEdge(edgeAB)
      .addEdge(edgeAC)
      .addEdge(edgeBC)
      .addEdge(edgeCD)
      .addEdge(edgeDE);

    const bridges = Object.values(graphBridges(graph));
```

```
    expect(bridges.length).toBe(2);
    expect(bridges[0].getKey()).toBe(edgeDE.getKey());
    expect(bridges[1].getKey()).toBe(edgeCD.getKey());
  });

  it('should find bridges in yet another graph #2', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');
    const vertexF = new GraphVertex('F');
    const vertexG = new GraphVertex('G');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeAC = new GraphEdge(vertexA, vertexC);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeCD = new GraphEdge(vertexC, vertexD);
    const edgeCE = new GraphEdge(vertexC, vertexE);
    const edgeCF = new GraphEdge(vertexC, vertexF);
    const edgeEG = new GraphEdge(vertexE, vertexG);
    const edgeFG = new GraphEdge(vertexF, vertexG);

    const graph = new Graph();

    graph
      .addEdge(edgeAB)
      .addEdge(edgeAC)
      .addEdge(edgeBC)
      .addEdge(edgeCD)
      .addEdge(edgeCE)
      .addEdge(edgeCF)
      .addEdge(edgeEG)
      .addEdge(edgeFG);

    const bridges = Object.values(graphBridges(graph));

    expect(bridges.length).toBe(1);
    expect(bridges[0].getKey()).toBe(edgeCD.getKey());
  });
});
```

Listing 13: graphBridges.js

```
import depthFirstSearch from '../depth-first-search/depthFirstSearch';

/**
 * Helper class for visited vertex metadata.
 */
class VisitMetadata {
  constructor({ discoveryTime, lowDiscoveryTime }) {
    this.discoveryTime = discoveryTime;
    this.lowDiscoveryTime = lowDiscoveryTime;
  }
}

/**
 * @param {Graph} graph
 * @return {Object}
 */
export default function graphBridges(graph) {
  // Set of vertices we've already visited during DFS.
  const visitedSet = {};

  // Set of bridges.
  const bridges = {};

  // Time needed to discover to the current vertex.
  let discoveryTime = 0;

  // Peek the start vertex for DFS traversal.
  const startVertex = graph.getAllVertices()[0];

  const dfsCallbacks = {
    /**
     * @param {GraphVertex} currentVertex
     */
    enterVertex: ({ currentVertex }) => {
      // Tick discovery time.
      discoveryTime += 1;

      // Put current vertex to visited set.
      visitedSet[currentVertex.getKey()] = new VisitMetadata({
        discoveryTime,
        lowDiscoveryTime: discoveryTime,
      });
    },
    /**
     * @param {GraphVertex} currentVertex
     * @param {GraphVertex} previousVertex
     */
    leaveVertex: ({ currentVertex, previousVertex }) => {
      if (previousVertex === null) {
        // Don't do anything for the root vertex if it is already current (not previous one).
        return;
      }

      // Check if current node is connected to any early node other then previous one.
      visitedSet[currentVertex.getKey()].lowDiscoveryTime = currentVertex.getNeighbors()
        .filter(earlyNeighbor => earlyNeighbor.getKey() !== previousVertex.getKey())
        .reduce(
          /**
           * @param {number} lowestDiscoveryTime
           * @param {GraphVertex} neighbor
           */
          (lowestDiscoveryTime, neighbor) => {
            const neighborLowTime = visitedSet[neighbor.getKey()].lowDiscoveryTime;
            return neighborLowTime < lowestDiscoveryTime ? neighborLowTime : lowestDiscoveryTime;
          },
          visitedSet[currentVertex.getKey()].lowDiscoveryTime,
        );

      // Compare low discovery times. In case if current low discovery time is less than the one
      // in previous vertex then update previous vertex low time.
      const currentLowDiscoveryTime = visitedSet[currentVertex.getKey()].lowDiscoveryTime;
      const previousLowDiscoveryTime = visitedSet[previousVertex.getKey()].lowDiscoveryTime;
      if (currentLowDiscoveryTime < previousLowDiscoveryTime) {
        visitedSet[previousVertex.getKey()].lowDiscoveryTime = currentLowDiscoveryTime;
      }

      // Compare current vertex low discovery time with parent discovery time. Check if there
      // are any short path (back edge) exists. If we can't get to current vertex other then
      // via parent then the parent vertex is articulation point for current one.
```

```
      const parentDiscoveryTime = visitedSet[previousVertex.getKey()].discoveryTime;
      if (parentDiscoveryTime < currentLowDiscoveryTime) {
        const bridge = graph.findEdge(previousVertex, currentVertex);
        bridges[bridge.getKey()] = bridge;
      }
    },
    allowTraversal: ({ nextVertex }) => {
      return !visitedSet[nextVertex.getKey()];
    },
  };

  // Do Depth First Search traversal over submitted graph.
  depthFirstSearch(graph, startVertex, dfsCallbacks);

  return bridges;
}
```

Listing 14: depthFirstSearch.test.js

```
import Graph from '../../../../data-structures/graph/Graph';
import GraphVertex from '../../../../data-structures/graph/GraphVertex';
import GraphEdge from '../../../../data-structures/graph/GraphEdge';
import depthFirstSearch from '../depthFirstSearch';

describe('depthFirstSearch', () => {
  it('should perform DFS operation on graph', () => {
    const graph = new Graph(true);

    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');
    const vertexF = new GraphVertex('F');
    const vertexG = new GraphVertex('G');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeCG = new GraphEdge(vertexC, vertexG);
    const edgeAD = new GraphEdge(vertexA, vertexD);
    const edgeAE = new GraphEdge(vertexA, vertexE);
    const edgeEF = new GraphEdge(vertexE, vertexF);
    const edgeFD = new GraphEdge(vertexF, vertexD);
    const edgeDG = new GraphEdge(vertexD, vertexG);

    graph
      .addEdge(edgeAB)
      .addEdge(edgeBC)
      .addEdge(edgeCG)
      .addEdge(edgeAD)
      .addEdge(edgeAE)
      .addEdge(edgeEF)
      .addEdge(edgeFD)
      .addEdge(edgeDG);

    expect(graph.toString()).toBe('A,B,C,G,D,E,F');

    const enterVertexCallback = jest.fn();
    const leaveVertexCallback = jest.fn();

    // Traverse graphs without callbacks first to check default ones.
    depthFirstSearch(graph, vertexA);

    // Traverse graph with enterVertex and leaveVertex callbacks.
    depthFirstSearch(graph, vertexA, {
      enterVertex: enterVertexCallback,
      leaveVertex: leaveVertexCallback,
    });

    expect(enterVertexCallback).toHaveBeenCalledTimes(graph.getAllVertices().length);
    expect(leaveVertexCallback).toHaveBeenCalledTimes(graph.getAllVertices().length);

    const enterVertexParamsMap = [
      { currentVertex: vertexA, previousVertex: null },
      { currentVertex: vertexB, previousVertex: vertexA },
      { currentVertex: vertexC, previousVertex: vertexB },
      { currentVertex: vertexG, previousVertex: vertexC },
      { currentVertex: vertexD, previousVertex: vertexA },
      { currentVertex: vertexE, previousVertex: vertexA },
      { currentVertex: vertexF, previousVertex: vertexE },
    ];

    for (let callIndex = 0; callIndex < graph.getAllVertices().length; callIndex += 1) {
      const params = enterVertexCallback.mock.calls[callIndex][0];
      expect(params.currentVertex).toEqual(enterVertexParamsMap[callIndex].currentVertex);
      expect(params.previousVertex).toEqual(enterVertexParamsMap[callIndex].previousVertex);
    }

    const leaveVertexParamsMap = [
      { currentVertex: vertexG, previousVertex: vertexC },
      { currentVertex: vertexC, previousVertex: vertexB },
      { currentVertex: vertexB, previousVertex: vertexA },
      { currentVertex: vertexD, previousVertex: vertexA },
      { currentVertex: vertexF, previousVertex: vertexE },
      { currentVertex: vertexE, previousVertex: vertexA },
      { currentVertex: vertexA, previousVertex: null },
    ];
```

```
    for (let callIndex = 0; callIndex < graph.getAllVertices().length; callIndex += 1) {
      const params = leaveVertexCallback.mock.calls[callIndex][0];
      expect(params.currentVertex).toEqual(leaveVertexParamsMap[callIndex].currentVertex);
      expect(params.previousVertex).toEqual(leaveVertexParamsMap[callIndex].previousVertex);
    }
  });

  it('allow users to redefine vertex visiting logic', () => {
    const graph = new Graph(true);

    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');
    const vertexF = new GraphVertex('F');
    const vertexG = new GraphVertex('G');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeCG = new GraphEdge(vertexC, vertexG);
    const edgeAD = new GraphEdge(vertexA, vertexD);
    const edgeAE = new GraphEdge(vertexA, vertexE);
    const edgeEF = new GraphEdge(vertexE, vertexF);
    const edgeFD = new GraphEdge(vertexF, vertexD);
    const edgeDG = new GraphEdge(vertexD, vertexG);

    graph
      .addEdge(edgeAB)
      .addEdge(edgeBC)
      .addEdge(edgeCG)
      .addEdge(edgeAD)
      .addEdge(edgeAE)
      .addEdge(edgeEF)
      .addEdge(edgeFD)
      .addEdge(edgeDG);

    expect(graph.toString()).toBe('A,B,C,G,D,E,F');

    const enterVertexCallback = jest.fn();
    const leaveVertexCallback = jest.fn();

    depthFirstSearch(graph, vertexA, {
      enterVertex: enterVertexCallback,
      leaveVertex: leaveVertexCallback,
      allowTraversal: ({ currentVertex, nextVertex }) => {
        return !(currentVertex === vertexA && nextVertex === vertexB);
      },
    });

    expect(enterVertexCallback).toHaveBeenCalledTimes(7);
    expect(leaveVertexCallback).toHaveBeenCalledTimes(7);

    const enterVertexParamsMap = [
      { currentVertex: vertexA, previousVertex: null },
      { currentVertex: vertexD, previousVertex: vertexA },
      { currentVertex: vertexG, previousVertex: vertexD },
      { currentVertex: vertexE, previousVertex: vertexA },
      { currentVertex: vertexF, previousVertex: vertexE },
      { currentVertex: vertexD, previousVertex: vertexF },
      { currentVertex: vertexG, previousVertex: vertexD },
    ];

    for (let callIndex = 0; callIndex < graph.getAllVertices().length; callIndex += 1) {
      const params = enterVertexCallback.mock.calls[callIndex][0];
      expect(params.currentVertex).toEqual(enterVertexParamsMap[callIndex].currentVertex);
      expect(params.previousVertex).toEqual(enterVertexParamsMap[callIndex].previousVertex);
    }

    const leaveVertexParamsMap = [
      { currentVertex: vertexG, previousVertex: vertexD },
      { currentVertex: vertexD, previousVertex: vertexA },
      { currentVertex: vertexG, previousVertex: vertexD },
      { currentVertex: vertexD, previousVertex: vertexF },
      { currentVertex: vertexF, previousVertex: vertexE },
      { currentVertex: vertexE, previousVertex: vertexA },
      { currentVertex: vertexA, previousVertex: null },
    ];

    for (let callIndex = 0; callIndex < graph.getAllVertices().length; callIndex += 1) {
      const params = leaveVertexCallback.mock.calls[callIndex][0];
      expect(params.currentVertex).toEqual(leaveVertexParamsMap[callIndex].currentVertex);
```

```
      expect(params.previousVertex).toEqual(leaveVertexParamsMap[callIndex].previousVertex);
    }
  });
});
```

Listing 15: depthFirstSearch.js

```
/**
 * @typedef {Object} Callbacks
 *
 * @property {function(vertices: Object): boolean} [allowTraversal] -
 *  Determines whether DFS should traverse from the vertex to its neighbor
 *  (along the edge). By default prohibits visiting the same vertex again.
 *
 * @property {function(vertices: Object)} [enterVertex] - Called when DFS enters the vertex.
 *
 * @property {function(vertices: Object)} [leaveVertex] - Called when DFS leaves the vertex.
 */

/**
 * @param {Callbacks} [callbacks]
 * @returns {Callbacks}
 */
function initCallbacks(callbacks = {}) {
  const initiatedCallback = callbacks;

  const stubCallback = () => {};

  const allowTraversalCallback = (
    () => {
      const seen = {};
      return ({ nextVertex }) => {
        if (!seen[nextVertex.getKey()]) {
          seen[nextVertex.getKey()] = true;
          return true;
        }
        return false;
      };
    }
  )();

  initiatedCallback.allowTraversal = callbacks.allowTraversal || allowTraversalCallback;
  initiatedCallback.enterVertex = callbacks.enterVertex || stubCallback;
  initiatedCallback.leaveVertex = callbacks.leaveVertex || stubCallback;

  return initiatedCallback;
}

/**
 * @param {Graph} graph
 * @param {GraphVertex} currentVertex
 * @param {GraphVertex} previousVertex
 * @param {Callbacks} callbacks
 */
function depthFirstSearchRecursive(graph, currentVertex, previousVertex, callbacks) {
  callbacks.enterVertex({ currentVertex, previousVertex });

  graph.getNeighbors(currentVertex).forEach((nextVertex) => {
    if (callbacks.allowTraversal({ previousVertex, currentVertex, nextVertex })) {
      depthFirstSearchRecursive(graph, nextVertex, currentVertex, callbacks);
    }
  });

  callbacks.leaveVertex({ currentVertex, previousVertex });
}

/**
 * @param {Graph} graph
 * @param {GraphVertex} startVertex
 * @param {Callbacks} [callbacks]
 */
export default function depthFirstSearch(graph, startVertex, callbacks) {
  const previousVertex = null;
  depthFirstSearchRecursive(graph, startVertex, previousVertex, initCallbacks(callbacks));
}
```

Listing 16: detectDirectedCycle.test.js

```
import GraphVertex from '../../../../data-structures/graph/GraphVertex';
import GraphEdge from '../../../../data-structures/graph/GraphEdge';
import Graph from '../../../../data-structures/graph/Graph';
import detectDirectedCycle from '../detectDirectedCycle';

describe('detectDirectedCycle', () => {
  it('should detect directed cycle', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');
    const vertexF = new GraphVertex('F');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeAC = new GraphEdge(vertexA, vertexC);
    const edgeDA = new GraphEdge(vertexD, vertexA);
    const edgeDE = new GraphEdge(vertexD, vertexE);
    const edgeEF = new GraphEdge(vertexE, vertexF);
    const edgeFD = new GraphEdge(vertexF, vertexD);

    const graph = new Graph(true);
    graph
      .addEdge(edgeAB)
      .addEdge(edgeBC)
      .addEdge(edgeAC)
      .addEdge(edgeDA)
      .addEdge(edgeDE)
      .addEdge(edgeEF);

    expect(detectDirectedCycle(graph)).toBeNull();

    graph.addEdge(edgeFD);

    expect(detectDirectedCycle(graph)).toEqual({
      D: vertexF,
      F: vertexE,
      E: vertexD,
    });
  });
});
```

Listing 17: detectUndirectedCycle.test.js

```
import GraphVertex from '../../../../data-structures/graph/GraphVertex';
import GraphEdge from '../../../../data-structures/graph/GraphEdge';
import Graph from '../../../../data-structures/graph/Graph';
import detectUndirectedCycle from '../detectUndirectedCycle';

describe('detectUndirectedCycle', () => {
  it('should detect undirected cycle', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');
    const vertexF = new GraphVertex('F');

    const edgeAF = new GraphEdge(vertexA, vertexF);
    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeBE = new GraphEdge(vertexB, vertexE);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeCD = new GraphEdge(vertexC, vertexD);
    const edgeDE = new GraphEdge(vertexD, vertexE);

    const graph = new Graph();
    graph
      .addEdge(edgeAF)
      .addEdge(edgeAB)
      .addEdge(edgeBE)
      .addEdge(edgeBC)
      .addEdge(edgeCD);

    expect(detectUndirectedCycle(graph)).toBeNull();

    graph.addEdge(edgeDE);

    expect(detectUndirectedCycle(graph)).toEqual({
      B: vertexC,
      C: vertexD,
      D: vertexE,
      E: vertexB,
    });
  });
});
```

Listing 18: detectUndirectedCycleUsingDisjointSet.test.js

```
import GraphVertex from '../../../../data-structures/graph/GraphVertex';
import GraphEdge from '../../../../data-structures/graph/GraphEdge';
import Graph from '../../../../data-structures/graph/Graph';
import detectUndirectedCycleUsingDisjointSet from '../detectUndirectedCycleUsingDisjointSet';

describe('detectUndirectedCycleUsingDisjointSet', () => {
  it('should detect undirected cycle', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');
    const vertexF = new GraphVertex('F');

    const edgeAF = new GraphEdge(vertexA, vertexF);
    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeBE = new GraphEdge(vertexB, vertexE);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeCD = new GraphEdge(vertexC, vertexD);
    const edgeDE = new GraphEdge(vertexD, vertexE);

    const graph = new Graph();
    graph
      .addEdge(edgeAF)
      .addEdge(edgeAB)
      .addEdge(edgeBE)
      .addEdge(edgeBC)
      .addEdge(edgeCD);

    expect(detectUndirectedCycleUsingDisjointSet(graph)).toBe(false);

    graph.addEdge(edgeDE);

    expect(detectUndirectedCycleUsingDisjointSet(graph)).toBe(true);
  });
});
```

Listing 19: detectDirectedCycle.js

```
import depthFirstSearch from '../depth-first-search/depthFirstSearch';

/**
 * Detect cycle in directed graph using Depth First Search.
 *
 * @param {Graph} graph
 */
export default function detectDirectedCycle(graph) {
  let cycle = null;

  // Will store parents (previous vertices) for all visited nodes.
  // This will be needed in order to specify what path exactly is a cycle.
  const dfsParentMap = {};

  // White set (UNVISITED) contains all the vertices that haven't been visited at all.
  const whiteSet = {};

  // Gray set (VISITING) contains all the vertices that are being visited right now
  // (in current path).
  const graySet = {};

  // Black set (VISITED) contains all the vertices that has been fully visited.
  // Meaning that all children of the vertex has been visited.
  const blackSet = {};

  // If we encounter vertex in gray set it means that we've found a cycle.
  // Because when vertex in gray set it means that its neighbors or its neighbors
  // neighbors are still being explored.

  // Init white set and add all vertices to it.
  /** @param {GraphVertex} vertex */
  graph.getAllVertices().forEach((vertex) => {
    whiteSet[vertex.getKey()] = vertex;
  });

  // Describe BFS callbacks.
  const callbacks = {
    enterVertex: ({ currentVertex, previousVertex }) => {
      if (graySet[currentVertex.getKey()]) {
        // If current vertex already in grey set it means that cycle is detected.
        // Let's detect cycle path.
        cycle = {};

        let currentCycleVertex = currentVertex;
        let previousCycleVertex = previousVertex;

        while (previousCycleVertex.getKey() !== currentVertex.getKey()) {
          cycle[currentCycleVertex.getKey()] = previousCycleVertex;
          currentCycleVertex = previousCycleVertex;
          previousCycleVertex = dfsParentMap[previousCycleVertex.getKey()];
        }

        cycle[currentCycleVertex.getKey()] = previousCycleVertex;
      } else {
        // Otherwise let's add current vertex to gray set and remove it from white set.
        graySet[currentVertex.getKey()] = currentVertex;
        delete whiteSet[currentVertex.getKey()];

        // Update DFS parents list.
        dfsParentMap[currentVertex.getKey()] = previousVertex;
      }
    },
    leaveVertex: ({ currentVertex }) => {
      // If all node's children has been visited let's remove it from gray set
      // and move it to the black set meaning that all its neighbors are visited.
      blackSet[currentVertex.getKey()] = currentVertex;
      delete graySet[currentVertex.getKey()];
    },
    allowTraversal: ({ nextVertex }) => {
      // If cycle was detected we must forbid all further traversing since it will
      // cause infinite traversal loop.
      if (cycle) {
        return false;
      }

      // Allow traversal only for the vertices that are not in black set
      // since all black set vertices have been already visited.
      return !blackSet[nextVertex.getKey()];
    },
```

```
  };

  // Start exploring vertices.
  while (Object.keys(whiteSet).length) {
    // Pick fist vertex to start BFS from.
    const firstWhiteKey = Object.keys(whiteSet)[0];
    const startVertex = whiteSet[firstWhiteKey];

    // Do Depth First Search.
    depthFirstSearch(graph, startVertex, callbacks);
  }

  return cycle;
}
```

Listing 20: detectUndirectedCycle.js

```
import depthFirstSearch from '../depth-first-search/depthFirstSearch';

/**
 * Detect cycle in undirected graph using Depth First Search.
 *
 * @param {Graph} graph
 */
export default function detectUndirectedCycle(graph) {
  let cycle = null;

  // List of vertices that we have visited.
  const visitedVertices = {};

  // List of parents vertices for every visited vertex.
  const parents = {};

  // Callbacks for DFS traversing.
  const callbacks = {
    allowTraversal: ({ currentVertex, nextVertex }) => {
      // Don't allow further traversal in case if cycle has been detected.
      if (cycle) {
        return false;
      }

      // Don't allow traversal from child back to its parent.
      const currentVertexParent = parents[currentVertex.getKey()];
      const currentVertexParentKey = currentVertexParent ? currentVertexParent.getKey() : null;

      return currentVertexParentKey !== nextVertex.getKey();
    },
    enterVertex: ({ currentVertex, previousVertex }) => {
      if (visitedVertices[currentVertex.getKey()]) {
        // Compile cycle path based on parents of previous vertices.
        cycle = {};

        let currentCycleVertex = currentVertex;
        let previousCycleVertex = previousVertex;

        while (previousCycleVertex.getKey() !== currentVertex.getKey()) {
          cycle[currentCycleVertex.getKey()] = previousCycleVertex;
          currentCycleVertex = previousCycleVertex;
          previousCycleVertex = parents[previousCycleVertex.getKey()];
        }

        cycle[currentCycleVertex.getKey()] = previousCycleVertex;
      } else {
        // Add next vertex to visited set.
        visitedVertices[currentVertex.getKey()] = currentVertex;
        parents[currentVertex.getKey()] = previousVertex;
      }
    },
  };

  // Start DFS traversing.
  const startVertex = graph.getAllVertices()[0];
  depthFirstSearch(graph, startVertex, callbacks);

  return cycle;
}
```

Listing 21: detectUndirectedCycleUsingDisjointSet.js

```
 import DisjointSet from '../../../data-structures/disjoint-set/DisjointSet';

/**
 * Detect cycle in undirected graph using disjoint sets.
 *
 * @param {Graph} graph
 */
export default function detectUndirectedCycleUsingDisjointSet(graph) {
  // Create initial singleton disjoint sets for each graph vertex.
  /** @param {GraphVertex} graphVertex */
  const keyExtractor = graphVertex => graphVertex.getKey();
  const disjointSet = new DisjointSet(keyExtractor);
  graph.getAllVertices().forEach(graphVertex => disjointSet.makeSet(graphVertex));

  // Go trough all graph edges one by one and check if edge vertices are from the
  // different sets. In this case joint those sets together. Do this until you find
  // an edge where to edge vertices are already in one set. This means that current
  // edge will create a cycle.
  let cycleFound = false;
  /** @param {GraphEdge} graphEdge */
  graph.getAllEdges().forEach((graphEdge) => {
    if (disjointSet.inSameSet(graphEdge.startVertex, graphEdge.endVertex)) {
      // Cycle found.
      cycleFound = true;
    } else {
      disjointSet.union(graphEdge.startVertex, graphEdge.endVertex);
    }
  });

  return cycleFound;
}
```

Listing 22: dijkstra.test.js

```
import GraphVertex from '../../../../data-structures/graph/GraphVertex';
import GraphEdge from '../../../../data-structures/graph/GraphEdge';
import Graph from '../../../../data-structures/graph/Graph';
import dijkstra from '../dijkstra';

describe('dijkstra', () => {
  it('should find minimum paths to all vertices for undirected graph', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');
    const vertexF = new GraphVertex('F');
    const vertexG = new GraphVertex('G');
    const vertexH = new GraphVertex('H');

    const edgeAB = new GraphEdge(vertexA, vertexB, 4);
    const edgeAE = new GraphEdge(vertexA, vertexE, 7);
    const edgeAC = new GraphEdge(vertexA, vertexC, 3);
    const edgeBC = new GraphEdge(vertexB, vertexC, 6);
    const edgeBD = new GraphEdge(vertexB, vertexD, 5);
    const edgeEC = new GraphEdge(vertexE, vertexC, 8);
    const edgeED = new GraphEdge(vertexE, vertexD, 2);
    const edgeDC = new GraphEdge(vertexD, vertexC, 11);
    const edgeDG = new GraphEdge(vertexD, vertexG, 10);
    const edgeDF = new GraphEdge(vertexD, vertexF, 2);
    const edgeFG = new GraphEdge(vertexF, vertexG, 3);
    const edgeEG = new GraphEdge(vertexE, vertexG, 5);

    const graph = new Graph();
    graph
      .addVertex(vertexH)
      .addEdge(edgeAB)
      .addEdge(edgeAE)
      .addEdge(edgeAC)
      .addEdge(edgeBC)
      .addEdge(edgeBD)
      .addEdge(edgeEC)
      .addEdge(edgeED)
      .addEdge(edgeDC)
      .addEdge(edgeDG)
      .addEdge(edgeDF)
      .addEdge(edgeFG)
      .addEdge(edgeEG);

    const { distances, previousVertices } = dijkstra(graph, vertexA);

    expect(distances).toEqual({
      H: Infinity,
      A: 0,
      B: 4,
      E: 7,
      C: 3,
      D: 9,
      G: 12,
      F: 11,
    });

    expect(previousVertices.F.getKey()).toBe('D');
    expect(previousVertices.D.getKey()).toBe('B');
    expect(previousVertices.B.getKey()).toBe('A');
    expect(previousVertices.G.getKey()).toBe('E');
    expect(previousVertices.C.getKey()).toBe('A');
    expect(previousVertices.A).toBeNull();
    expect(previousVertices.H).toBeNull();
  });

  it('should find minimum paths to all vertices for directed graph with negative edge weights', () => {
    const vertexS = new GraphVertex('S');
    const vertexE = new GraphVertex('E');
    const vertexA = new GraphVertex('A');
    const vertexD = new GraphVertex('D');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexH = new GraphVertex('H');

    const edgeSE = new GraphEdge(vertexS, vertexE, 8);
    const edgeSA = new GraphEdge(vertexS, vertexA, 10);
    const edgeED = new GraphEdge(vertexE, vertexD, 1);
```

```
    const edgeDA = new GraphEdge(vertexD, vertexA, -4);
    const edgeDC = new GraphEdge(vertexD, vertexC, -1);
    const edgeAC = new GraphEdge(vertexA, vertexC, 2);
    const edgeCB = new GraphEdge(vertexC, vertexB, -2);
    const edgeBA = new GraphEdge(vertexB, vertexA, 1);

    const graph = new Graph(true);
    graph
      .addVertex(vertexH)
      .addEdge(edgeSE)
      .addEdge(edgeSA)
      .addEdge(edgeED)
      .addEdge(edgeDA)
      .addEdge(edgeDC)
      .addEdge(edgeAC)
      .addEdge(edgeCB)
      .addEdge(edgeBA);

    const { distances, previousVertices } = dijkstra(graph, vertexS);

    expect(distances).toEqual({
      H: Infinity,
      S: 0,
      A: 5,
      B: 5,
      C: 7,
      D: 9,
      E: 8,
    });

    expect(previousVertices.H).toBeNull();
    expect(previousVertices.S).toBeNull();
    expect(previousVertices.B.getKey()).toBe('C');
    expect(previousVertices.C.getKey()).toBe('A');
    expect(previousVertices.A.getKey()).toBe('D');
    expect(previousVertices.D.getKey()).toBe('E');
  });
});
```

Listing 23: dijkstra.js

```
import PriorityQueue from '../../../data-structures/priority-queue/PriorityQueue';

/**
 * @typedef {Object} ShortestPaths
 * @property {Object} distances - shortest distances to all vertices
 * @property {Object} previousVertices - shortest paths to all vertices.
 */

/**
 * Implementation of Dijkstra algorithm of finding the shortest paths to graph nodes.
 * @param {Graph} graph - graph we're going to traverse.
 * @param {GraphVertex} startVertex - traversal start vertex.
 * @return {ShortestPaths}
 */
export default function dijkstra(graph, startVertex) {
  // Init helper variables that we will need for Dijkstra algorithm.
  const distances = {};
  const visitedVertices = {};
  const previousVertices = {};
  const queue = new PriorityQueue();

  // Init all distances with infinity assuming that currently we can't reach
  // any of the vertices except the start one.
  graph.getAllVertices().forEach((vertex) => {
    distances[vertex.getKey()] = Infinity;
    previousVertices[vertex.getKey()] = null;
  });

  // We are already at the startVertex so the distance to it is zero.
  distances[startVertex.getKey()] = 0;

  // Init vertices queue.
  queue.add(startVertex, distances[startVertex.getKey()]);

  // Iterate over the priority queue of vertices until it is empty.
  while (!queue.isEmpty()) {
    // Fetch next closest vertex.
    const currentVertex = queue.poll();

    // Iterate over every unvisited neighbor of the current vertex.
    currentVertex.getNeighbors().forEach((neighbor) => {
      // Don't visit already visited vertices.
      if (!visitedVertices[neighbor.getKey()]) {
        // Update distances to every neighbor from current vertex.
        const edge = graph.findEdge(currentVertex, neighbor);

        const existingDistanceToNeighbor = distances[neighbor.getKey()];
        const distanceToNeighborFromCurrent = distances[currentVertex.getKey()] + edge.weight;

        // If we've found shorter path to the neighbor - update it.
        if (distanceToNeighborFromCurrent < existingDistanceToNeighbor) {
          distances[neighbor.getKey()] = distanceToNeighborFromCurrent;

          // Change priority of the neighbor in a queue since it might have became closer.
          if (queue.hasValue(neighbor)) {
            queue.changePriority(neighbor, distances[neighbor.getKey()]);
          }

          // Remember previous closest vertex.
          previousVertices[neighbor.getKey()] = currentVertex;
        }

        // Add neighbor to the queue for further visiting.
        if (!queue.hasValue(neighbor)) {
          queue.add(neighbor, distances[neighbor.getKey()]);
        }
      }
    });

    // Add current vertex to visited ones to avoid visiting it again later.
    visitedVertices[currentVertex.getKey()] = currentVertex;
  }

  // Return the set of shortest distances to all vertices and the set of
  // shortest paths to all vertices in a graph.
  return {
    distances,
    previousVertices,
  };
```

```
}
```

Listing 24: eulerianPath.test.js

```
 import GraphVertex from '../../../../data-structures/graph/GraphVertex';
import GraphEdge from '../../../../data-structures/graph/GraphEdge';
import Graph from '../../../../data-structures/graph/Graph';
import eulerianPath from '../eulerianPath';

describe('eulerianPath', () => {
  it('should throw an error when graph is not Eulerian', () => {
    function findEulerianPathInNotEulerianGraph() {
      const vertexA = new GraphVertex('A');
      const vertexB = new GraphVertex('B');
      const vertexC = new GraphVertex('C');
      const vertexD = new GraphVertex('D');
      const vertexE = new GraphVertex('E');

      const edgeAB = new GraphEdge(vertexA, vertexB);
      const edgeAC = new GraphEdge(vertexA, vertexC);
      const edgeBC = new GraphEdge(vertexB, vertexC);
      const edgeBD = new GraphEdge(vertexB, vertexD);
      const edgeCE = new GraphEdge(vertexC, vertexE);

      const graph = new Graph();

      graph
        .addEdge(edgeAB)
        .addEdge(edgeAC)
        .addEdge(edgeBC)
        .addEdge(edgeBD)
        .addEdge(edgeCE);

      eulerianPath(graph);
    }

    expect(findEulerianPathInNotEulerianGraph).toThrowError();
  });

  it('should find Eulerian Circuit in graph', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');
    const vertexF = new GraphVertex('F');
    const vertexG = new GraphVertex('G');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeAE = new GraphEdge(vertexA, vertexE);
    const edgeAF = new GraphEdge(vertexA, vertexF);
    const edgeAG = new GraphEdge(vertexA, vertexG);
    const edgeGF = new GraphEdge(vertexG, vertexF);
    const edgeBE = new GraphEdge(vertexB, vertexE);
    const edgeEB = new GraphEdge(vertexE, vertexB);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeED = new GraphEdge(vertexE, vertexD);
    const edgeCD = new GraphEdge(vertexC, vertexD);

    const graph = new Graph();

    graph
      .addEdge(edgeAB)
      .addEdge(edgeAE)
      .addEdge(edgeAF)
      .addEdge(edgeAG)
      .addEdge(edgeGF)
      .addEdge(edgeBE)
      .addEdge(edgeEB)
      .addEdge(edgeBC)
      .addEdge(edgeED)
      .addEdge(edgeCD);

    const graphEdgesCount = graph.getAllEdges().length;

    const eulerianPathSet = eulerianPath(graph);

    expect(eulerianPathSet.length).toBe(graphEdgesCount + 1);

    expect(eulerianPathSet[0].getKey()).toBe(vertexA.getKey());
    expect(eulerianPathSet[1].getKey()).toBe(vertexB.getKey());
    expect(eulerianPathSet[2].getKey()).toBe(vertexE.getKey());
    expect(eulerianPathSet[3].getKey()).toBe(vertexB.getKey());
```

```
    expect(eulerianPathSet[4].getKey()).toBe(vertexC.getKey());
    expect(eulerianPathSet[5].getKey()).toBe(vertexD.getKey());
    expect(eulerianPathSet[6].getKey()).toBe(vertexE.getKey());
    expect(eulerianPathSet[7].getKey()).toBe(vertexA.getKey());
    expect(eulerianPathSet[8].getKey()).toBe(vertexF.getKey());
    expect(eulerianPathSet[9].getKey()).toBe(vertexG.getKey());
    expect(eulerianPathSet[10].getKey()).toBe(vertexA.getKey());
  });

  it('should find Eulerian Path in graph', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');
    const vertexF = new GraphVertex('F');
    const vertexG = new GraphVertex('G');
    const vertexH = new GraphVertex('H');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeAC = new GraphEdge(vertexA, vertexC);
    const edgeBD = new GraphEdge(vertexB, vertexD);
    const edgeDC = new GraphEdge(vertexD, vertexC);
    const edgeCE = new GraphEdge(vertexC, vertexE);
    const edgeEF = new GraphEdge(vertexE, vertexF);
    const edgeFH = new GraphEdge(vertexF, vertexH);
    const edgeFG = new GraphEdge(vertexF, vertexG);
    const edgeHG = new GraphEdge(vertexH, vertexG);

    const graph = new Graph();

    graph
      .addEdge(edgeAB)
      .addEdge(edgeAC)
      .addEdge(edgeBD)
      .addEdge(edgeDC)
      .addEdge(edgeCE)
      .addEdge(edgeEF)
      .addEdge(edgeFH)
      .addEdge(edgeFG)
      .addEdge(edgeHG);

    const graphEdgesCount = graph.getAllEdges().length;

    const eulerianPathSet = eulerianPath(graph);

    expect(eulerianPathSet.length).toBe(graphEdgesCount + 1);

    expect(eulerianPathSet[0].getKey()).toBe(vertexC.getKey());
    expect(eulerianPathSet[1].getKey()).toBe(vertexA.getKey());
    expect(eulerianPathSet[2].getKey()).toBe(vertexB.getKey());
    expect(eulerianPathSet[3].getKey()).toBe(vertexD.getKey());
    expect(eulerianPathSet[4].getKey()).toBe(vertexC.getKey());
    expect(eulerianPathSet[5].getKey()).toBe(vertexE.getKey());
    expect(eulerianPathSet[6].getKey()).toBe(vertexF.getKey());
    expect(eulerianPathSet[7].getKey()).toBe(vertexH.getKey());
    expect(eulerianPathSet[8].getKey()).toBe(vertexG.getKey());
    expect(eulerianPathSet[9].getKey()).toBe(vertexF.getKey());
  });
});
```

Listing 25: eulerianPath.js

```
import graphBridges from '../bridges/graphBridges';

/**
 * Fleury's algorithm of finding Eulerian Path (visit all graph edges exactly once).
 *
 * @param {Graph} graph
 * @return {GraphVertex[]}
 */
export default function eulerianPath(graph) {
  const eulerianPathVertices = [];

  // Set that contains all vertices with even rank (number of neighbors).
  const evenRankVertices = {};

  // Set that contains all vertices with odd rank (number of neighbors).
  const oddRankVertices = {};

  // Set of all not visited edges.
  const notVisitedEdges = {};
  graph.getAllEdges().forEach((vertex) => {
    notVisitedEdges[vertex.getKey()] = vertex;
  });

  // Detect whether graph contains Eulerian Circuit or Eulerian Path or none of them.
  /** @params {GraphVertex} vertex */
  graph.getAllVertices().forEach((vertex) => {
    if (vertex.getDegree() % 2) {
      oddRankVertices[vertex.getKey()] = vertex;
    } else {
      evenRankVertices[vertex.getKey()] = vertex;
    }
  });

  // Check whether we're dealing with Eulerian Circuit or Eulerian Path only.
  // Graph would be an Eulerian Circuit in case if all its vertices has even degree.
  // If not all vertices have even degree then graph must contain only two odd-degree
  // vertices in order to have Euler Path.
  const isCircuit = !Object.values(oddRankVertices).length;

  if (!isCircuit && Object.values(oddRankVertices).length !== 2) {
    throw new Error('Eulerian path must contain two odd-ranked vertices');
  }

  // Pick start vertex for traversal.
  let startVertex = null;

  if (isCircuit) {
    // For Eulerian Circuit it doesn't matter from what vertex to start thus we'll just
    // peek a first node.
    const evenVertexKey = Object.keys(evenRankVertices)[0];
    startVertex = evenRankVertices[evenVertexKey];
  } else {
    // For Eulerian Path we need to start from one of two odd-degree vertices.
    const oddVertexKey = Object.keys(oddRankVertices)[0];
    startVertex = oddRankVertices[oddVertexKey];
  }

  // Start traversing the graph.
  let currentVertex = startVertex;
  while (Object.values(notVisitedEdges).length) {
    // Add current vertex to Eulerian path.
    eulerianPathVertices.push(currentVertex);

    // Detect all bridges in graph.
    // We need to do it in order to not delete bridges if there are other edges
    // exists for deletion.
    const bridges = graphBridges(graph);

    // Peek the next edge to delete from graph.
    const currentEdges = currentVertex.getEdges();
    /** @var {GraphEdge} edgeToDelete */
    let edgeToDelete = null;
    if (currentEdges.length === 1) {
      // If there is only one edge left we need to peek it.
      [edgeToDelete] = currentEdges;
    } else {
      // If there are many edges left then we need to peek any of those except bridges.
      [edgeToDelete] = currentEdges.filter(edge => !bridges[edge.getKey()]);
    }
```

```
    // Detect next current vertex.
    if (currentVertex.getKey() === edgeToDelete.startVertex.getKey()) {
      currentVertex = edgeToDelete.endVertex;
    } else {
      currentVertex = edgeToDelete.startVertex;
    }

    // Delete edge from not visited edges set.
    delete notVisitedEdges[edgeToDelete.getKey()];

    // If last edge were deleted then add finish vertex to Eulerian Path.
    if (Object.values(notVisitedEdges).length === 0) {
      eulerianPathVertices.push(currentVertex);
    }

    // Delete the edge from graph.
    graph.deleteEdge(edgeToDelete);
  }

  return eulerianPathVertices;
}
```

Listing 26: floydWarshall.test.js

```
import GraphVertex from '../../../../data-structures/graph/GraphVertex';
import GraphEdge from '../../../../data-structures/graph/GraphEdge';
import Graph from '../../../../data-structures/graph/Graph';
import floydWarshall from '../floydWarshall';

describe('floydWarshall', () => {
  it('should find minimum paths to all vertices for undirected graph', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');
    const vertexF = new GraphVertex('F');
    const vertexG = new GraphVertex('G');
    const vertexH = new GraphVertex('H');

    const edgeAB = new GraphEdge(vertexA, vertexB, 4);
    const edgeAE = new GraphEdge(vertexA, vertexE, 7);
    const edgeAC = new GraphEdge(vertexA, vertexC, 3);
    const edgeBC = new GraphEdge(vertexB, vertexC, 6);
    const edgeBD = new GraphEdge(vertexB, vertexD, 5);
    const edgeEC = new GraphEdge(vertexE, vertexC, 8);
    const edgeED = new GraphEdge(vertexE, vertexD, 2);
    const edgeDC = new GraphEdge(vertexD, vertexC, 11);
    const edgeDG = new GraphEdge(vertexD, vertexG, 10);
    const edgeDF = new GraphEdge(vertexD, vertexF, 2);
    const edgeFG = new GraphEdge(vertexF, vertexG, 3);
    const edgeEG = new GraphEdge(vertexE, vertexG, 5);

    const graph = new Graph();

    // Add vertices first just to have them in desired order.
    graph
      .addVertex(vertexA)
      .addVertex(vertexB)
      .addVertex(vertexC)
      .addVertex(vertexD)
      .addVertex(vertexE)
      .addVertex(vertexF)
      .addVertex(vertexG)
      .addVertex(vertexH);

    // Now, when vertices are in correct order let's add edges.
    graph
      .addEdge(edgeAB)
      .addEdge(edgeAE)
      .addEdge(edgeAC)
      .addEdge(edgeBC)
      .addEdge(edgeBD)
      .addEdge(edgeEC)
      .addEdge(edgeED)
      .addEdge(edgeDC)
      .addEdge(edgeDG)
      .addEdge(edgeDF)
      .addEdge(edgeFG)
      .addEdge(edgeEG);

    const { distances, nextVertices } = floydWarshall(graph);

    const vertices = graph.getAllVertices();

    const vertexAIndex = vertices.indexOf(vertexA);
    const vertexBIndex = vertices.indexOf(vertexB);
    const vertexCIndex = vertices.indexOf(vertexC);
    const vertexDIndex = vertices.indexOf(vertexD);
    const vertexEIndex = vertices.indexOf(vertexE);
    const vertexFIndex = vertices.indexOf(vertexF);
    const vertexGIndex = vertices.indexOf(vertexG);
    const vertexHIndex = vertices.indexOf(vertexH);

    expect(distances[vertexAIndex][vertexHIndex]).toBe(Infinity);
    expect(distances[vertexAIndex][vertexAIndex]).toBe(0);
    expect(distances[vertexAIndex][vertexBIndex]).toBe(4);
    expect(distances[vertexAIndex][vertexEIndex]).toBe(7);
    expect(distances[vertexAIndex][vertexCIndex]).toBe(3);
    expect(distances[vertexAIndex][vertexDIndex]).toBe(9);
    expect(distances[vertexAIndex][vertexGIndex]).toBe(12);
    expect(distances[vertexAIndex][vertexFIndex]).toBe(11);
```

```
    expect(nextVertices[vertexAIndex][vertexFIndex]).toBe(vertexD);
    expect(nextVertices[vertexAIndex][vertexDIndex]).toBe(vertexB);
    expect(nextVertices[vertexAIndex][vertexBIndex]).toBe(vertexA);
    expect(nextVertices[vertexAIndex][vertexGIndex]).toBe(vertexE);
    expect(nextVertices[vertexAIndex][vertexCIndex]).toBe(vertexA);
    expect(nextVertices[vertexAIndex][vertexAIndex]).toBe(null);
    expect(nextVertices[vertexAIndex][vertexHIndex]).toBe(null);
  });

  it('should find minimum paths to all vertices for directed graph', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');

    const edgeAB = new GraphEdge(vertexA, vertexB, 3);
    const edgeBA = new GraphEdge(vertexB, vertexA, 8);
    const edgeAD = new GraphEdge(vertexA, vertexD, 7);
    const edgeDA = new GraphEdge(vertexD, vertexA, 2);
    const edgeBC = new GraphEdge(vertexB, vertexC, 2);
    const edgeCA = new GraphEdge(vertexC, vertexA, 5);
    const edgeCD = new GraphEdge(vertexC, vertexD, 1);

    const graph = new Graph(true);

    // Add vertices first just to have them in desired order.
    graph
      .addVertex(vertexA)
      .addVertex(vertexB)
      .addVertex(vertexC)
      .addVertex(vertexD);

    // Now, when vertices are in correct order let's add edges.
    graph
      .addEdge(edgeAB)
      .addEdge(edgeBA)
      .addEdge(edgeAD)
      .addEdge(edgeDA)
      .addEdge(edgeBC)
      .addEdge(edgeCA)
      .addEdge(edgeCD);

    const { distances, nextVertices } = floydWarshall(graph);

    const vertices = graph.getAllVertices();

    const vertexAIndex = vertices.indexOf(vertexA);
    const vertexBIndex = vertices.indexOf(vertexB);
    const vertexCIndex = vertices.indexOf(vertexC);
    const vertexDIndex = vertices.indexOf(vertexD);

    expect(distances[vertexAIndex][vertexAIndex]).toBe(0);
    expect(distances[vertexAIndex][vertexBIndex]).toBe(3);
    expect(distances[vertexAIndex][vertexCIndex]).toBe(5);
    expect(distances[vertexAIndex][vertexDIndex]).toBe(6);

    expect(distances).toEqual([
      [0, 3, 5, 6],
      [5, 0, 2, 3],
      [3, 6, 0, 1],
      [2, 5, 7, 0],
    ]);

    expect(nextVertices[vertexAIndex][vertexDIndex]).toBe(vertexC);
    expect(nextVertices[vertexAIndex][vertexCIndex]).toBe(vertexB);
    expect(nextVertices[vertexBIndex][vertexDIndex]).toBe(vertexC);
    expect(nextVertices[vertexAIndex][vertexAIndex]).toBe(null);
    expect(nextVertices[vertexAIndex][vertexBIndex]).toBe(vertexA);
  });

  it('should find minimum paths to all vertices for directed graph with negative edge weights', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');
    const vertexF = new GraphVertex('F');
    const vertexG = new GraphVertex('G');

    const edgeFE = new GraphEdge(vertexF, vertexE, 8);
    const edgeFA = new GraphEdge(vertexF, vertexA, 10);
    const edgeED = new GraphEdge(vertexE, vertexD, 1);
```

```
    const edgeDA = new GraphEdge(vertexD, vertexA, -4);
    const edgeDC = new GraphEdge(vertexD, vertexC, -1);
    const edgeAC = new GraphEdge(vertexA, vertexC, 2);
    const edgeCB = new GraphEdge(vertexC, vertexB, -2);
    const edgeBA = new GraphEdge(vertexB, vertexA, 1);

    const graph = new Graph(true);

    // Add vertices first just to have them in desired order.
    graph
      .addVertex(vertexA)
      .addVertex(vertexB)
      .addVertex(vertexC)
      .addVertex(vertexD)
      .addVertex(vertexE)
      .addVertex(vertexF)
      .addVertex(vertexG);

    // Now, when vertices are in correct order let's add edges.
    graph
      .addEdge(edgeFE)
      .addEdge(edgeFA)
      .addEdge(edgeED)
      .addEdge(edgeDA)
      .addEdge(edgeDC)
      .addEdge(edgeAC)
      .addEdge(edgeCB)
      .addEdge(edgeBA);

    const { distances, nextVertices } = floydWarshall(graph);

    const vertices = graph.getAllVertices();

    const vertexAIndex = vertices.indexOf(vertexA);
    const vertexBIndex = vertices.indexOf(vertexB);
    const vertexCIndex = vertices.indexOf(vertexC);
    const vertexDIndex = vertices.indexOf(vertexD);
    const vertexEIndex = vertices.indexOf(vertexE);
    const vertexGIndex = vertices.indexOf(vertexG);
    const vertexFIndex = vertices.indexOf(vertexF);

    expect(distances[vertexFIndex][vertexGIndex]).toBe(Infinity);
    expect(distances[vertexFIndex][vertexFIndex]).toBe(0);
    expect(distances[vertexFIndex][vertexAIndex]).toBe(5);
    expect(distances[vertexFIndex][vertexBIndex]).toBe(5);
    expect(distances[vertexFIndex][vertexCIndex]).toBe(7);
    expect(distances[vertexFIndex][vertexDIndex]).toBe(9);
    expect(distances[vertexFIndex][vertexEIndex]).toBe(8);

    expect(nextVertices[vertexFIndex][vertexGIndex]).toBe(null);
    expect(nextVertices[vertexFIndex][vertexFIndex]).toBe(null);
    expect(nextVertices[vertexAIndex][vertexBIndex]).toBe(vertexC);
    expect(nextVertices[vertexAIndex][vertexCIndex]).toBe(vertexA);
    expect(nextVertices[vertexFIndex][vertexBIndex]).toBe(vertexE);
    expect(nextVertices[vertexEIndex][vertexBIndex]).toBe(vertexD);
    expect(nextVertices[vertexDIndex][vertexBIndex]).toBe(vertexC);
    expect(nextVertices[vertexCIndex][vertexBIndex]).toBe(vertexC);
  });
});
```

Listing 27: floydWarshall.js

```
/**
 * @param {Graph} graph
 * @return {{distances: number[][], nextVertices: GraphVertex[][]}}
 */
export default function floydWarshall(graph) {
  // Get all graph vertices.
  const vertices = graph.getAllVertices();

  // Init previous vertices matrix with nulls meaning that there are no
  // previous vertices exist that will give us shortest path.
  const nextVertices = Array(vertices.length).fill(null).map(() => {
    return Array(vertices.length).fill(null);
  });

  // Init distances matrix with Infinities meaning there are no paths
  // between vertices exist so far.
  const distances = Array(vertices.length).fill(null).map(() => {
    return Array(vertices.length).fill(Infinity);
  });

  // Init distance matrix with the distance we already now (from existing edges).
  // And also init previous vertices from the edges.
  vertices.forEach((startVertex, startIndex) => {
    vertices.forEach((endVertex, endIndex) => {
      if (startVertex === endVertex) {
        // Distance to the vertex itself is 0.
        distances[startIndex][endIndex] = 0;
      } else {
        // Find edge between the start and end vertices.
        const edge = graph.findEdge(startVertex, endVertex);

        if (edge) {
          // There is an edge from vertex with startIndex to vertex with endIndex.
          // Save distance and previous vertex.
          distances[startIndex][endIndex] = edge.weight;
          nextVertices[startIndex][endIndex] = startVertex;
        } else {
          distances[startIndex][endIndex] = Infinity;
        }
      }
    });
  });

  // Now let's go to the core of the algorithm.
  // Let's all pair of vertices (from start to end ones) and try to check if there
  // is a shorter path exists between them via middle vertex. Middle vertex may also
  // be one of the graph vertices. As you may see now we're going to have three
  // loops over all graph vertices: for start, end and middle vertices.
  vertices.forEach((middleVertex, middleIndex) => {
    // Path starts from startVertex with startIndex.
    vertices.forEach((startVertex, startIndex) => {
      // Path ends to endVertex with endIndex.
      vertices.forEach((endVertex, endIndex) => {
        // Compare existing distance from startVertex to endVertex, with distance
        // from startVertex to endVertex but via middleVertex.
        // Save the shortest distance and previous vertex that allows
        // us to have this shortest distance.
        const distViaMiddle = distances[startIndex][middleIndex] + distances[middleIndex][endIndex];

        if (distances[startIndex][endIndex] > distViaMiddle) {
          // We've found a shortest pass via middle vertex.
          distances[startIndex][endIndex] = distViaMiddle;
          nextVertices[startIndex][endIndex] = middleVertex;
        }
      });
    });
  });

  // Shortest distance from x to y: distance[x][y].
  // Next vertex after x one in path from x to y: nextVertices[x][y].
  return { distances, nextVertices };
}
```

Listing 28: hamiltonianCycle.test.js

```
 import GraphVertex from '../../../../data-structures/graph/GraphVertex';
import GraphEdge from '../../../../data-structures/graph/GraphEdge';
import Graph from '../../../../data-structures/graph/Graph';
import hamiltonianCycle from '../hamiltonianCycle';

describe('hamiltonianCycle', () => {
  it('should find hamiltonian paths in graph', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeAE = new GraphEdge(vertexA, vertexE);
    const edgeAC = new GraphEdge(vertexA, vertexC);
    const edgeBE = new GraphEdge(vertexB, vertexE);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeBD = new GraphEdge(vertexB, vertexD);
    const edgeCD = new GraphEdge(vertexC, vertexD);
    const edgeDE = new GraphEdge(vertexD, vertexE);

    const graph = new Graph();
    graph
      .addEdge(edgeAB)
      .addEdge(edgeAE)
      .addEdge(edgeAC)
      .addEdge(edgeBE)
      .addEdge(edgeBC)
      .addEdge(edgeBD)
      .addEdge(edgeCD)
      .addEdge(edgeDE);

    const hamiltonianCycleSet = hamiltonianCycle(graph);

    expect(hamiltonianCycleSet.length).toBe(8);

    expect(hamiltonianCycleSet[0][0].getKey()).toBe(vertexA.getKey());
    expect(hamiltonianCycleSet[0][1].getKey()).toBe(vertexB.getKey());
    expect(hamiltonianCycleSet[0][2].getKey()).toBe(vertexE.getKey());
    expect(hamiltonianCycleSet[0][3].getKey()).toBe(vertexD.getKey());
    expect(hamiltonianCycleSet[0][4].getKey()).toBe(vertexC.getKey());

    expect(hamiltonianCycleSet[1][0].getKey()).toBe(vertexA.getKey());
    expect(hamiltonianCycleSet[1][1].getKey()).toBe(vertexB.getKey());
    expect(hamiltonianCycleSet[1][2].getKey()).toBe(vertexC.getKey());
    expect(hamiltonianCycleSet[1][3].getKey()).toBe(vertexD.getKey());
    expect(hamiltonianCycleSet[1][4].getKey()).toBe(vertexE.getKey());

    expect(hamiltonianCycleSet[2][0].getKey()).toBe(vertexA.getKey());
    expect(hamiltonianCycleSet[2][1].getKey()).toBe(vertexE.getKey());
    expect(hamiltonianCycleSet[2][2].getKey()).toBe(vertexB.getKey());
    expect(hamiltonianCycleSet[2][3].getKey()).toBe(vertexD.getKey());
    expect(hamiltonianCycleSet[2][4].getKey()).toBe(vertexC.getKey());

    expect(hamiltonianCycleSet[3][0].getKey()).toBe(vertexA.getKey());
    expect(hamiltonianCycleSet[3][1].getKey()).toBe(vertexE.getKey());
    expect(hamiltonianCycleSet[3][2].getKey()).toBe(vertexD.getKey());
    expect(hamiltonianCycleSet[3][3].getKey()).toBe(vertexB.getKey());
    expect(hamiltonianCycleSet[3][4].getKey()).toBe(vertexC.getKey());
  });

  it('should return false for graph without Hamiltonian path', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeAE = new GraphEdge(vertexA, vertexE);
    const edgeBE = new GraphEdge(vertexB, vertexE);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeBD = new GraphEdge(vertexB, vertexD);
    const edgeCD = new GraphEdge(vertexC, vertexD);

    const graph = new Graph();
    graph
      .addEdge(edgeAB)
```

```
      .addEdge(edgeAE)
      .addEdge(edgeBE)
      .addEdge(edgeBC)
      .addEdge(edgeBD)
      .addEdge(edgeCD);

    const hamiltonianCycleSet = hamiltonianCycle(graph);

    expect(hamiltonianCycleSet.length).toBe(0);
  });
});
```

Listing 29: hamiltonianCycle.js

```
import GraphVertex from '../../../data-structures/graph/GraphVertex';

/**
 * @param {number[][]} adjacencyMatrix
 * @param {object} verticesIndices
 * @param {GraphVertex[]} cycle
 * @param {GraphVertex} vertexCandidate
 * @return {boolean}
 */
function isSafe(adjacencyMatrix, verticesIndices, cycle, vertexCandidate) {
  const endVertex = cycle[cycle.length - 1];

  // Get end and candidate vertices indices in adjacency matrix.
  const candidateVertexAdjacencyIndex = verticesIndices[vertexCandidate.getKey()];
  const endVertexAdjacencyIndex = verticesIndices[endVertex.getKey()];

  // Check if last vertex in the path and candidate vertex are adjacent.
  if (adjacencyMatrix[endVertexAdjacencyIndex][candidateVertexAdjacencyIndex] === Infinity) {
    return false;
  }

  // Check if vertexCandidate is being added to the path for the first time.
  const candidateDuplicate = cycle.find(vertex => vertex.getKey() === vertexCandidate.getKey());

  return !candidateDuplicate;
}

/**
 * @param {number[][]} adjacencyMatrix
 * @param {object} verticesIndices
 * @param {GraphVertex[]} cycle
 * @return {boolean}
 */
function isCycle(adjacencyMatrix, verticesIndices, cycle) {
  // Check if first and last vertices in hamiltonian path are adjacent.

  // Get start and end vertices from the path.
  const startVertex = cycle[0];
  const endVertex = cycle[cycle.length - 1];

  // Get start/end vertices indices in adjacency matrix.
  const startVertexAdjacencyIndex = verticesIndices[startVertex.getKey()];
  const endVertexAdjacencyIndex = verticesIndices[endVertex.getKey()];

  // Check if we can go from end vertex to the start one.
  return adjacencyMatrix[endVertexAdjacencyIndex][startVertexAdjacencyIndex] !== Infinity;
}

/**
 * @param {number[][]} adjacencyMatrix
 * @param {GraphVertex[]} vertices
 * @param {object} verticesIndices
 * @param {GraphVertex[][]} cycles
 * @param {GraphVertex[]} cycle
 */
function hamiltonianCycleRecursive({
  adjacencyMatrix,
  vertices,
  verticesIndices,
  cycles,
  cycle,
}) {
  // Clone cycle in order to prevent it from modification by other DFS branches.
  const currentCycle = [...cycle].map(vertex => new GraphVertex(vertex.value));

  if (vertices.length === currentCycle.length) {
    // Hamiltonian path is found.
    // Now we need to check if it is cycle or not.
    if (isCycle(adjacencyMatrix, verticesIndices, currentCycle)) {
      // Another solution has been found. Save it.
      cycles.push(currentCycle);
    }
    return;
  }

  for (let vertexIndex = 0; vertexIndex < vertices.length; vertexIndex += 1) {
    // Get vertex candidate that we will try to put into next path step and see if it fits.
    const vertexCandidate = vertices[vertexIndex];
```

```
    // Check if it is safe to put vertex candidate to cycle.
    if (isSafe(adjacencyMatrix, verticesIndices, currentCycle, vertexCandidate)) {
      // Add candidate vertex to cycle path.
      currentCycle.push(vertexCandidate);

      // Try to find other vertices in cycle.
      hamiltonianCycleRecursive({
        adjacencyMatrix,
        vertices,
        verticesIndices,
        cycles,
        cycle: currentCycle,
      });

      // BACKTRACKING.
      // Remove candidate vertex from cycle path in order to try another one.
      currentCycle.pop();
    }
  }
}

/**
 * @param {Graph} graph
 * @return {GraphVertex[][]}
 */
export default function hamiltonianCycle(graph) {
  // Gather some information about the graph that we will need to during
  // the problem solving.
  const verticesIndices = graph.getVerticesIndices();
  const adjacencyMatrix = graph.getAdjacencyMatrix();
  const vertices = graph.getAllVertices();

  // Define start vertex. We will always pick the first one
  // this it doesn't matter which vertex to pick in a cycle.
  // Every vertex is in a cycle so we can start from any of them.
  const startVertex = vertices[0];

  // Init cycles array that will hold all solutions.
  const cycles = [];

  // Init cycle array that will hold current cycle path.
  const cycle = [startVertex];

  // Try to find cycles recursively in Depth First Search order.
  hamiltonianCycleRecursive({
    adjacencyMatrix,
    vertices,
    verticesIndices,
    cycles,
    cycle,
  });

  // Return found cycles.
  return cycles;
}
```

Listing 30: kruskal.test.js

```
 import GraphVertex from '../../../../data-structures/graph/GraphVertex';
import GraphEdge from '../../../../data-structures/graph/GraphEdge';
import Graph from '../../../../data-structures/graph/Graph';
import kruskal from '../kruskal';

describe('kruskal', () => {
  it('should fire an error for directed graph', () => {
    function applyPrimToDirectedGraph() {
      const graph = new Graph(true);

      kruskal(graph);
    }

    expect(applyPrimToDirectedGraph).toThrowError();
  });

  it('should find minimum spanning tree', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');
    const vertexF = new GraphVertex('F');
    const vertexG = new GraphVertex('G');

    const edgeAB = new GraphEdge(vertexA, vertexB, 2);
    const edgeAD = new GraphEdge(vertexA, vertexD, 3);
    const edgeAC = new GraphEdge(vertexA, vertexC, 3);
    const edgeBC = new GraphEdge(vertexB, vertexC, 4);
    const edgeBE = new GraphEdge(vertexB, vertexE, 3);
    const edgeDF = new GraphEdge(vertexD, vertexF, 7);
    const edgeEC = new GraphEdge(vertexE, vertexC, 1);
    const edgeEF = new GraphEdge(vertexE, vertexF, 8);
    const edgeFG = new GraphEdge(vertexF, vertexG, 9);
    const edgeFC = new GraphEdge(vertexF, vertexC, 6);

    const graph = new Graph();

    graph
      .addEdge(edgeAB)
      .addEdge(edgeAD)
      .addEdge(edgeAC)
      .addEdge(edgeBC)
      .addEdge(edgeBE)
      .addEdge(edgeDF)
      .addEdge(edgeEC)
      .addEdge(edgeEF)
      .addEdge(edgeFC)
      .addEdge(edgeFG);

    expect(graph.getWeight()).toEqual(46);

    const minimumSpanningTree = kruskal(graph);

    expect(minimumSpanningTree.getWeight()).toBe(24);
    expect(minimumSpanningTree.getAllVertices().length).toBe(graph.getAllVertices().length);
    expect(minimumSpanningTree.getAllEdges().length).toBe(graph.getAllVertices().length - 1);
    expect(minimumSpanningTree.toString()).toBe('E,C,A,B,D,F,G');
  });

  it('should find minimum spanning tree for simple graph', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');

    const edgeAB = new GraphEdge(vertexA, vertexB, 1);
    const edgeAD = new GraphEdge(vertexA, vertexD, 3);
    const edgeBC = new GraphEdge(vertexB, vertexC, 1);
    const edgeBD = new GraphEdge(vertexB, vertexD, 3);
    const edgeCD = new GraphEdge(vertexC, vertexD, 1);

    const graph = new Graph();

    graph
      .addEdge(edgeAB)
      .addEdge(edgeAD)
      .addEdge(edgeBC)
      .addEdge(edgeBD)
```

```
      .addEdge(edgeCD);

    expect(graph.getWeight()).toEqual(9);

    const minimumSpanningTree = kruskal(graph);

    expect(minimumSpanningTree.getWeight()).toBe(3);
    expect(minimumSpanningTree.getAllVertices().length).toBe(graph.getAllVertices().length);
    expect(minimumSpanningTree.getAllEdges().length).toBe(graph.getAllVertices().length - 1);
    expect(minimumSpanningTree.toString()).toBe('A,B,C,D');
  });
});
```

Listing 31: kruskal.js

```
 import Graph from '../../../data-structures/graph/Graph';
import QuickSort from '../../sorting/quick-sort/QuickSort';
import DisjointSet from '../../../data-structures/disjoint-set/DisjointSet';

/**
 * @param {Graph} graph
 * @return {Graph}
 */
export default function kruskal(graph) {
  // It should fire error if graph is directed since the algorithm works only
  // for undirected graphs.
  if (graph.isDirected) {
    throw new Error('Kruskal\'s algorithms works only for undirected graphs');
  }

  // Init new graph that will contain minimum spanning tree of original graph.
  const minimumSpanningTree = new Graph();

  // Sort all graph edges in increasing order.
  const sortingCallbacks = {
    /**
     * @param {GraphEdge} graphEdgeA
     * @param {GraphEdge} graphEdgeB
     */
    compareCallback: (graphEdgeA, graphEdgeB) => {
      if (graphEdgeA.weight === graphEdgeB.weight) {
        return 1;
      }

      return graphEdgeA.weight <= graphEdgeB.weight ? -1 : 1;
    },
  };
  const sortedEdges = new QuickSort(sortingCallbacks).sort(graph.getAllEdges());

  // Create disjoint sets for all graph vertices.
  const keyCallback = graphVertex => graphVertex.getKey();
  const disjointSet = new DisjointSet(keyCallback);

  graph.getAllVertices().forEach((graphVertex) => {
    disjointSet.makeSet(graphVertex);
  });

  // Go through all edges started from the minimum one and try to add them
  // to minimum spanning tree. The criteria of adding the edge would be whether
  // it is forms the cycle or not (if it connects two vertices from one disjoint
  // set or not).
  for (let edgeIndex = 0; edgeIndex < sortedEdges.length; edgeIndex += 1) {
    /** @var {GraphEdge} currentEdge */
    const currentEdge = sortedEdges[edgeIndex];

    // Check if edge forms the cycle. If it does then skip it.
    if (!disjointSet.inSameSet(currentEdge.startVertex, currentEdge.endVertex)) {
      // Unite two subsets into one.
      disjointSet.union(currentEdge.startVertex, currentEdge.endVertex);

      // Add this edge to spanning tree.
      minimumSpanningTree.addEdge(currentEdge);
    }
  }

  return minimumSpanningTree;
}
```

Listing 32: prim.test.js

```
import GraphVertex from '../../../../data-structures/graph/GraphVertex';
import GraphEdge from '../../../../data-structures/graph/GraphEdge';
import Graph from '../../../../data-structures/graph/Graph';
import prim from '../prim';

describe('prim', () => {
  it('should fire an error for directed graph', () => {
    function applyPrimToDirectedGraph() {
      const graph = new Graph(true);

      prim(graph);
    }

    expect(applyPrimToDirectedGraph).toThrowError();
  });

  it('should find minimum spanning tree', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');
    const vertexF = new GraphVertex('F');
    const vertexG = new GraphVertex('G');

    const edgeAB = new GraphEdge(vertexA, vertexB, 2);
    const edgeAD = new GraphEdge(vertexA, vertexD, 3);
    const edgeAC = new GraphEdge(vertexA, vertexC, 3);
    const edgeBC = new GraphEdge(vertexB, vertexC, 4);
    const edgeBE = new GraphEdge(vertexB, vertexE, 3);
    const edgeDF = new GraphEdge(vertexD, vertexF, 7);
    const edgeEC = new GraphEdge(vertexE, vertexC, 1);
    const edgeEF = new GraphEdge(vertexE, vertexF, 8);
    const edgeFG = new GraphEdge(vertexF, vertexG, 9);
    const edgeFC = new GraphEdge(vertexF, vertexC, 6);

    const graph = new Graph();

    graph
      .addEdge(edgeAB)
      .addEdge(edgeAD)
      .addEdge(edgeAC)
      .addEdge(edgeBC)
      .addEdge(edgeBE)
      .addEdge(edgeDF)
      .addEdge(edgeEC)
      .addEdge(edgeEF)
      .addEdge(edgeFC)
      .addEdge(edgeFG);

    expect(graph.getWeight()).toEqual(46);

    const minimumSpanningTree = prim(graph);

    expect(minimumSpanningTree.getWeight()).toBe(24);
    expect(minimumSpanningTree.getAllVertices().length).toBe(graph.getAllVertices().length);
    expect(minimumSpanningTree.getAllEdges().length).toBe(graph.getAllVertices().length - 1);
    expect(minimumSpanningTree.toString()).toBe('A,B,C,E,D,F,G');
  });

  it('should find minimum spanning tree for simple graph', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');

    const edgeAB = new GraphEdge(vertexA, vertexB, 1);
    const edgeAD = new GraphEdge(vertexA, vertexD, 3);
    const edgeBC = new GraphEdge(vertexB, vertexC, 1);
    const edgeBD = new GraphEdge(vertexB, vertexD, 3);
    const edgeCD = new GraphEdge(vertexC, vertexD, 1);

    const graph = new Graph();

    graph
      .addEdge(edgeAB)
      .addEdge(edgeAD)
      .addEdge(edgeBC)
      .addEdge(edgeBD)
```

```
      .addEdge(edgeCD);

    expect(graph.getWeight()).toEqual(9);

    const minimumSpanningTree = prim(graph);

    expect(minimumSpanningTree.getWeight()).toBe(3);
    expect(minimumSpanningTree.getAllVertices().length).toBe(graph.getAllVertices().length);
    expect(minimumSpanningTree.getAllEdges().length).toBe(graph.getAllVertices().length - 1);
    expect(minimumSpanningTree.toString()).toBe('A,B,C,D');
  });
});
```

Listing 33: prim.js

```
 import Graph from '../../../data-structures/graph/Graph';
import PriorityQueue from '../../../data-structures/priority-queue/PriorityQueue';

/**
 * @param {Graph} graph
 * @return {Graph}
 */
export default function prim(graph) {
  // It should fire error if graph is directed since the algorithm works only
  // for undirected graphs.
  if (graph.isDirected) {
    throw new Error('Prim\'s algorithms works only for undirected graphs');
  }

  // Init new graph that will contain minimum spanning tree of original graph.
  const minimumSpanningTree = new Graph();

  // This priority queue will contain all the edges that are starting from
  // visited nodes and they will be ranked by edge weight - so that on each step
  // we would always pick the edge with minimal edge weight.
  const edgesQueue = new PriorityQueue();

  // Set of vertices that has been already visited.
  const visitedVertices = {};

  // Vertex from which we will start graph traversal.
  const startVertex = graph.getAllVertices()[0];

  // Add start vertex to the set of visited ones.
  visitedVertices[startVertex.getKey()] = startVertex;

  // Add all edges of start vertex to the queue.
  startVertex.getEdges().forEach((graphEdge) => {
    edgesQueue.add(graphEdge, graphEdge.weight);
  });

  // Now let's explore all queued edges.
  while (!edgesQueue.isEmpty()) {
    // Fetch next queued edge with minimal weight.
    /** @var {GraphEdge} currentEdge */
    const currentMinEdge = edgesQueue.poll();

    // Find out the next unvisited minimal vertex to traverse.
    let nextMinVertex = null;
    if (!visitedVertices[currentMinEdge.startVertex.getKey()]) {
      nextMinVertex = currentMinEdge.startVertex;
    } else if (!visitedVertices[currentMinEdge.endVertex.getKey()]) {
      nextMinVertex = currentMinEdge.endVertex;
    }

    // If all vertices of current edge has been already visited then skip this round.
    if (nextMinVertex) {
      // Add current min edge to MST.
      minimumSpanningTree.addEdge(currentMinEdge);

      // Add vertex to the set of visited ones.
      visitedVertices[nextMinVertex.getKey()] = nextMinVertex;

      // Add all current vertex's edges to the queue.
      nextMinVertex.getEdges().forEach((graphEdge) => {
        // Add only vertices that link to unvisited nodes.
        if (
          !visitedVertices[graphEdge.startVertex.getKey()]
          || !visitedVertices[graphEdge.endVertex.getKey()]
        ) {
          edgesQueue.add(graphEdge, graphEdge.weight);
        }
      });
    }
  }

  return minimumSpanningTree;
}
```

Listing 34: stronglyConnectedComponents.test.js

```
import GraphVertex from '../../../../data-structures/graph/GraphVertex';
import GraphEdge from '../../../../data-structures/graph/GraphEdge';
import Graph from '../../../../data-structures/graph/Graph';
import stronglyConnectedComponents from '../stronglyConnectedComponents';

describe('stronglyConnectedComponents', () => {
  it('should detect strongly connected components in simple graph', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeCA = new GraphEdge(vertexC, vertexA);
    const edgeCD = new GraphEdge(vertexC, vertexD);

    const graph = new Graph(true);

    graph
      .addEdge(edgeAB)
      .addEdge(edgeBC)
      .addEdge(edgeCA)
      .addEdge(edgeCD);

    const components = stronglyConnectedComponents(graph);

    expect(components).toBeDefined();
    expect(components.length).toBe(2);

    expect(components[0][0].getKey()).toBe(vertexA.getKey());
    expect(components[0][1].getKey()).toBe(vertexC.getKey());
    expect(components[0][2].getKey()).toBe(vertexB.getKey());

    expect(components[1][0].getKey()).toBe(vertexD.getKey());
  });

  it('should detect strongly connected components in graph', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');
    const vertexF = new GraphVertex('F');
    const vertexG = new GraphVertex('G');
    const vertexH = new GraphVertex('H');
    const vertexI = new GraphVertex('I');
    const vertexJ = new GraphVertex('J');
    const vertexK = new GraphVertex('K');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeCA = new GraphEdge(vertexC, vertexA);
    const edgeBD = new GraphEdge(vertexB, vertexD);
    const edgeDE = new GraphEdge(vertexD, vertexE);
    const edgeEF = new GraphEdge(vertexE, vertexF);
    const edgeFD = new GraphEdge(vertexF, vertexD);
    const edgeGF = new GraphEdge(vertexG, vertexF);
    const edgeGH = new GraphEdge(vertexG, vertexH);
    const edgeHI = new GraphEdge(vertexH, vertexI);
    const edgeIJ = new GraphEdge(vertexI, vertexJ);
    const edgeJG = new GraphEdge(vertexJ, vertexG);
    const edgeJK = new GraphEdge(vertexJ, vertexK);

    const graph = new Graph(true);

    graph
      .addEdge(edgeAB)
      .addEdge(edgeBC)
      .addEdge(edgeCA)
      .addEdge(edgeBD)
      .addEdge(edgeDE)
      .addEdge(edgeEF)
      .addEdge(edgeFD)
      .addEdge(edgeGF)
      .addEdge(edgeGH)
      .addEdge(edgeHI)
      .addEdge(edgeIJ)
      .addEdge(edgeJG)
```

```
      .addEdge(edgeJK);

    const components = stronglyConnectedComponents(graph);

    expect(components).toBeDefined();
    expect(components.length).toBe(4);

    expect(components[0][0].getKey()).toBe(vertexG.getKey());
    expect(components[0][1].getKey()).toBe(vertexJ.getKey());
    expect(components[0][2].getKey()).toBe(vertexI.getKey());
    expect(components[0][3].getKey()).toBe(vertexH.getKey());

    expect(components[1][0].getKey()).toBe(vertexK.getKey());

    expect(components[2][0].getKey()).toBe(vertexA.getKey());
    expect(components[2][1].getKey()).toBe(vertexC.getKey());
    expect(components[2][2].getKey()).toBe(vertexB.getKey());

    expect(components[3][0].getKey()).toBe(vertexD.getKey());
    expect(components[3][1].getKey()).toBe(vertexF.getKey());
    expect(components[3][2].getKey()).toBe(vertexE.getKey());
  });
});
```

Listing 35: stronglyConnectedComponents.js

```
import Stack from '../../../data-structures/stack/Stack';
import depthFirstSearch from '../depth-first-search/depthFirstSearch';

/**
 * @param {Graph} graph
 * @return {Stack}
 */
function getVerticesSortedByDfsFinishTime(graph) {
  // Set of all visited vertices during DFS pass.
  const visitedVerticesSet = {};

  // Stack of vertices by finish time.
  // All vertices in this stack are ordered by finished time in decreasing order.
  // Vertex that has been finished first will be at the bottom of the stack and
  // vertex that has been finished last will be at the top of the stack.
  const verticesByDfsFinishTime = new Stack();

  // Set of all vertices we're going to visit.
  const notVisitedVerticesSet = {};
  graph.getAllVertices().forEach((vertex) => {
    notVisitedVerticesSet[vertex.getKey()] = vertex;
  });

  // Specify DFS traversal callbacks.
  const dfsCallbacks = {
    enterVertex: ({ currentVertex }) => {
      // Add current vertex to visited set.
      visitedVerticesSet[currentVertex.getKey()] = currentVertex;

      // Delete current vertex from not visited set.
      delete notVisitedVerticesSet[currentVertex.getKey()];
    },
    leaveVertex: ({ currentVertex }) => {
      // Push vertex to the stack when leaving it.
      // This will make stack to be ordered by finish time in decreasing order.
      verticesByDfsFinishTime.push(currentVertex);
    },
    allowTraversal: ({ nextVertex }) => {
      // Don't allow to traverse the nodes that have been already visited.
      return !visitedVerticesSet[nextVertex.getKey()];
    },
  };

  // Do FIRST DFS PASS traversal for all graph vertices to fill the verticesByFinishTime stack.
  while (Object.values(notVisitedVerticesSet).length) {
    // Peek any vertex to start DFS traversal from.
    const startVertexKey = Object.keys(notVisitedVerticesSet)[0];
    const startVertex = notVisitedVerticesSet[startVertexKey];
    delete notVisitedVerticesSet[startVertexKey];

    depthFirstSearch(graph, startVertex, dfsCallbacks);
  }

  return verticesByDfsFinishTime;
}

/**
 * @param {Graph} graph
 * @param {Stack} verticesByFinishTime
 * @return {*[]}
 */
function getSCCSets(graph, verticesByFinishTime) {
  // Array of arrays of strongly connected vertices.
  const stronglyConnectedComponentsSets = [];

  // Array that will hold all vertices that are being visited during one DFS run.
  let stronglyConnectedComponentsSet = [];

  // Visited vertices set.
  const visitedVerticesSet = {};

  // Callbacks for DFS traversal.
  const dfsCallbacks = {
    enterVertex: ({ currentVertex }) => {
      // Add current vertex to SCC set of current DFS round.
      stronglyConnectedComponentsSet.push(currentVertex);

      // Add current vertex to visited set.
      visitedVerticesSet[currentVertex.getKey()] = currentVertex;
```

```
    },
    leaveVertex: ({ previousVertex }) => {
      // Once DFS traversal is finished push the set of found strongly connected
      // components during current DFS round to overall strongly connected components set.
      // The sign that traversal is about to be finished is that we came back to start vertex
      // which doesn't have parent.
      if (previousVertex === null) {
        stronglyConnectedComponentsSets.push([...stronglyConnectedComponentsSet]);
      }
    },
    allowTraversal: ({ nextVertex }) => {
      // Don't allow traversal of already visited vertices.
      return !visitedVerticesSet[nextVertex.getKey()];
    },
  };

  while (!verticesByFinishTime.isEmpty()) {
    /** @var {GraphVertex} startVertex */
    const startVertex = verticesByFinishTime.pop();

    // Reset the set of strongly connected vertices.
    stronglyConnectedComponentsSet = [];

    // Don't do DFS on already visited vertices.
    if (!visitedVerticesSet[startVertex.getKey()]) {
      // Do DFS traversal.
      depthFirstSearch(graph, startVertex, dfsCallbacks);
    }
  }

  return stronglyConnectedComponentsSets;
}

/**
 * Kosaraju's algorithm.
 *
 * @param {Graph} graph
 * @return {*[]}
 */
export default function stronglyConnectedComponents(graph) {
  // In this algorithm we will need to do TWO DFS PASSES overt the graph.

  // Get stack of vertices ordered by DFS finish time.
  // All vertices in this stack are ordered by finished time in decreasing order:
  // Vertex that has been finished first will be at the bottom of the stack and
  // vertex that has been finished last will be at the top of the stack.
  const verticesByFinishTime = getVerticesSortedByDfsFinishTime(graph);

  // Reverse the graph.
  graph.reverse();

  // Do DFS once again on reversed graph.
  return getSCCSets(graph, verticesByFinishTime);
}
```

Listing 36: topologicalSort.test.js

```
import GraphVertex from '../../../../data-structures/graph/GraphVertex';
import GraphEdge from '../../../../data-structures/graph/GraphEdge';
import Graph from '../../../../data-structures/graph/Graph';
import topologicalSort from '../topologicalSort';

describe('topologicalSort', () => {
  it('should do topological sorting on graph', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');
    const vertexE = new GraphVertex('E');
    const vertexF = new GraphVertex('F');
    const vertexG = new GraphVertex('G');
    const vertexH = new GraphVertex('H');

    const edgeAC = new GraphEdge(vertexA, vertexC);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeBD = new GraphEdge(vertexB, vertexD);
    const edgeCE = new GraphEdge(vertexC, vertexE);
    const edgeDF = new GraphEdge(vertexD, vertexF);
    const edgeEF = new GraphEdge(vertexE, vertexF);
    const edgeEH = new GraphEdge(vertexE, vertexH);
    const edgeFG = new GraphEdge(vertexF, vertexG);

    const graph = new Graph(true);

    graph
      .addEdge(edgeAC)
      .addEdge(edgeBC)
      .addEdge(edgeBD)
      .addEdge(edgeCE)
      .addEdge(edgeDF)
      .addEdge(edgeEF)
      .addEdge(edgeEH)
      .addEdge(edgeFG);

    const sortedVertices = topologicalSort(graph);

    expect(sortedVertices).toBeDefined();
    expect(sortedVertices.length).toBe(graph.getAllVertices().length);
    expect(sortedVertices).toEqual([
      vertexB,
      vertexD,
      vertexA,
      vertexC,
      vertexE,
      vertexH,
      vertexF,
      vertexG,
    ]);
  });
});
```

Listing 37: topologicalSort.js

```
import Stack from '../../../data-structures/stack/Stack';
import depthFirstSearch from '../depth-first-search/depthFirstSearch';

/**
 * @param {Graph} graph
 */
export default function topologicalSort(graph) {
  // Create a set of all vertices we want to visit.
  const unvisitedSet = {};
  graph.getAllVertices().forEach((vertex) => {
    unvisitedSet[vertex.getKey()] = vertex;
  });

  // Create a set for all vertices that we've already visited.
  const visitedSet = {};

  // Create a stack of already ordered vertices.
  const sortedStack = new Stack();

  const dfsCallbacks = {
    enterVertex: ({ currentVertex }) => {
      // Add vertex to visited set in case if all its children has been explored.
      visitedSet[currentVertex.getKey()] = currentVertex;

      // Remove this vertex from unvisited set.
      delete unvisitedSet[currentVertex.getKey()];
    },
    leaveVertex: ({ currentVertex }) => {
      // If the vertex has been totally explored then we may push it to stack.
      sortedStack.push(currentVertex);
    },
    allowTraversal: ({ nextVertex }) => {
      return !visitedSet[nextVertex.getKey()];
    },
  };

  // Let's go and do DFS for all unvisited nodes.
  while (Object.keys(unvisitedSet).length) {
    const currentVertexKey = Object.keys(unvisitedSet)[0];
    const currentVertex = unvisitedSet[currentVertexKey];

    // Do DFS for current node.
    depthFirstSearch(graph, currentVertex, dfsCallbacks);
  }

  return sortedStack.toArray();
}
```

Listing 38: bfTravellingSalesman.test.js

```
import GraphVertex from '../../../../data-structures/graph/GraphVertex';
import GraphEdge from '../../../../data-structures/graph/GraphEdge';
import Graph from '../../../../data-structures/graph/Graph';
import bfTravellingSalesman from '../bfTravellingSalesman';

describe('bfTravellingSalesman', () => {
  it('should solve problem for simple graph', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');

    const edgeAB = new GraphEdge(vertexA, vertexB, 1);
    const edgeBD = new GraphEdge(vertexB, vertexD, 1);
    const edgeDC = new GraphEdge(vertexD, vertexC, 1);
    const edgeCA = new GraphEdge(vertexC, vertexA, 1);

    const edgeBA = new GraphEdge(vertexB, vertexA, 5);
    const edgeDB = new GraphEdge(vertexD, vertexB, 8);
    const edgeCD = new GraphEdge(vertexC, vertexD, 7);
    const edgeAC = new GraphEdge(vertexA, vertexC, 4);
    const edgeAD = new GraphEdge(vertexA, vertexD, 2);
    const edgeDA = new GraphEdge(vertexD, vertexA, 3);
    const edgeBC = new GraphEdge(vertexB, vertexC, 3);
    const edgeCB = new GraphEdge(vertexC, vertexB, 9);

    const graph = new Graph(true);
    graph
      .addEdge(edgeAB)
      .addEdge(edgeBD)
      .addEdge(edgeDC)
      .addEdge(edgeCA)
      .addEdge(edgeBA)
      .addEdge(edgeDB)
      .addEdge(edgeCD)
      .addEdge(edgeAC)
      .addEdge(edgeAD)
      .addEdge(edgeDA)
      .addEdge(edgeBC)
      .addEdge(edgeCB);

    const salesmanPath = bfTravellingSalesman(graph);

    expect(salesmanPath.length).toBe(4);

    expect(salesmanPath[0].getKey()).toEqual(vertexA.getKey());
    expect(salesmanPath[1].getKey()).toEqual(vertexB.getKey());
    expect(salesmanPath[2].getKey()).toEqual(vertexD.getKey());
    expect(salesmanPath[3].getKey()).toEqual(vertexC.getKey());
  });
});
```

Listing 39: bfTravellingSalesman.js

```
/**
 * Get all possible paths
 * @param {GraphVertex} startVertex
 * @param {GraphVertex[][]} [paths]
 * @param {GraphVertex[]} [path]
 */
function findAllPaths(startVertex, paths = [], path = []) {
  // Clone path.
  const currentPath = [...path];

  // Add startVertex to the path.
  currentPath.push(startVertex);

  // Generate visited set from path.
  const visitedSet = currentPath.reduce((accumulator, vertex) => {
    const updatedAccumulator = { ...accumulator };
    updatedAccumulator[vertex.getKey()] = vertex;

    return updatedAccumulator;
  }, {});

  // Get all unvisited neighbors of startVertex.
  const unvisitedNeighbors = startVertex.getNeighbors().filter((neighbor) => {
    return !visitedSet[neighbor.getKey()];
  });

  // If there no unvisited neighbors then treat current path as complete and save it.
  if (!unvisitedNeighbors.length) {
    paths.push(currentPath);

    return paths;
  }

  // Go through all the neighbors.
  for (let neighborIndex = 0; neighborIndex < unvisitedNeighbors.length; neighborIndex += 1) {
    const currentUnvisitedNeighbor = unvisitedNeighbors[neighborIndex];
    findAllPaths(currentUnvisitedNeighbor, paths, currentPath);
  }

  return paths;
}

/**
 * @param {number[][]} adjacencyMatrix
 * @param {object} verticesIndices
 * @param {GraphVertex[]} cycle
 * @return {number}
 */
function getCycleWeight(adjacencyMatrix, verticesIndices, cycle) {
  let weight = 0;

  for (let cycleIndex = 1; cycleIndex < cycle.length; cycleIndex += 1) {
    const fromVertex = cycle[cycleIndex - 1];
    const toVertex = cycle[cycleIndex];
    const fromVertexIndex = verticesIndices[fromVertex.getKey()];
    const toVertexIndex = verticesIndices[toVertex.getKey()];
    weight += adjacencyMatrix[fromVertexIndex][toVertexIndex];
  }

  return weight;
}

/**
 * BRUTE FORCE approach to solve Traveling Salesman Problem.
 *
 * @param {Graph} graph
 * @return {GraphVertex[]}
 */
export default function bfTravellingSalesman(graph) {
  // Pick starting point from where we will traverse the graph.
  const startVertex = graph.getAllVertices()[0];

  // BRUTE FORCE.
  // Generate all possible paths from startVertex.
  const allPossiblePaths = findAllPaths(startVertex);

  // Filter out paths that are not cycles.
  const allPossibleCycles = allPossiblePaths.filter((path) => {
    /** @var {GraphVertex} */
```

```
    const lastVertex = path[path.length - 1];
    const lastVertexNeighbors = lastVertex.getNeighbors();

    return lastVertexNeighbors.includes(startVertex);
  });

  // Go through all possible cycles and pick the one with minimum overall tour weight.
  const adjacencyMatrix = graph.getAdjacencyMatrix();
  const verticesIndices = graph.getVerticesIndices();
  let salesmanPath = [];
  let salesmanPathWeight = null;
  for (let cycleIndex = 0; cycleIndex < allPossibleCycles.length; cycleIndex += 1) {
    const currentCycle = allPossibleCycles[cycleIndex];
    const currentCycleWeight = getCycleWeight(adjacencyMatrix, verticesIndices, currentCycle);

    // If current cycle weight is smaller then previous ones treat current cycle as most optimal.
    if (salesmanPathWeight === null || currentCycleWeight < salesmanPathWeight) {
      salesmanPath = currentCycle;
      salesmanPathWeight = currentCycleWeight;
    }
  }

  // Return the solution.
  return salesmanPath;
}
```

Listing 40: reverseTraversal.test.js

```
import LinkedList from '../../../../data-structures/linked-list/LinkedList';
import reverseTraversal from '../reverseTraversal';

describe('reverseTraversal', () => {
  it('should traverse linked list in reverse order', () => {
    const linkedList = new LinkedList();

    linkedList
      .append(1)
      .append(2)
      .append(3);

    const traversedNodeValues = [];
    const traversalCallback = (nodeValue) => {
      traversedNodeValues.push(nodeValue);
    };

    reverseTraversal(linkedList, traversalCallback);

    expect(traversedNodeValues).toEqual([3, 2, 1]);
  });
});


// it('should reverse traversal the linked list with callback', () => {
//   const linkedList = new LinkedList();
//
//   linkedList
//     .append(1)
//     .append(2)
//     .append(3);
//
//   expect(linkedList.toString()).toBe('1,2,3');
//   expect(linkedList.reverseTraversal(linkedList.head, value => value * 2)).toEqual([6, 4, 2]);
//   expect(() => linkedList.reverseTraversal(linkedList.head)).toThrow();
// });
```

Listing 41: reverseTraversal.js

```
/**
 * Traversal callback function.
 * @callback traversalCallback
 * @param {*} nodeValue
 */

/**
 * @param {LinkedListNode} node
 * @param {traversalCallback} callback
 */
function reverseTraversalRecursive(node, callback) {
  if (node) {
    reverseTraversalRecursive(node.next, callback);
    callback(node.value);
  }
}

/**
 * @param {LinkedList} linkedList
 * @param {traversalCallback} callback
 */
export default function reverseTraversal(linkedList, callback) {
  reverseTraversalRecursive(linkedList.head, callback);
}
```

Listing 42: traversal.test.js

```
import LinkedList from '../../../../data-structures/linked-list/LinkedList';
import traversal from '../traversal';

describe('traversal', () => {
  it('should traverse linked list', () => {
    const linkedList = new LinkedList();

    linkedList
      .append(1)
      .append(2)
      .append(3);

    const traversedNodeValues = [];
    const traversalCallback = (nodeValue) => {
      traversedNodeValues.push(nodeValue);
    };

    traversal(linkedList, traversalCallback);

    expect(traversedNodeValues).toEqual([1, 2, 3]);
  });
});
```

Listing 43: traversal.js

```
 /**
 * Traversal callback function.
 * @callback traversalCallback
 * @param {*} nodeValue
 */

/**
 * @param {LinkedList} linkedList
 * @param {traversalCallback} callback
 */
export default function traversal(linkedList, callback) {
  let currentNode = linkedList.head;

  while (currentNode) {
    callback(currentNode.value);
    currentNode = currentNode.next;
  }
}
```

Listing 44: bitLength.test.js

```
import bitLength from '../bitLength';

describe('bitLength', () => {
  it('should calculate number of bits that the number is consists of', () => {
    expect(bitLength(0b0)).toBe(0);
    expect(bitLength(0b1)).toBe(1);
    expect(bitLength(0b01)).toBe(1);
    expect(bitLength(0b101)).toBe(3);
    expect(bitLength(0b0101)).toBe(3);
    expect(bitLength(0b10101)).toBe(5);
    expect(bitLength(0b11110101)).toBe(8);
    expect(bitLength(0b00011110101)).toBe(8);
  });
});
```

Listing 45: bitsDiff.test.js

```
import bitsDiff from '../bitsDiff';

describe('bitsDiff', () => {
  it('should calculate bits difference between two numbers', () => {
    expect(bitsDiff(0, 0)).toBe(0);
    expect(bitsDiff(1, 1)).toBe(0);
    expect(bitsDiff(124, 124)).toBe(0);
    expect(bitsDiff(0, 1)).toBe(1);
    expect(bitsDiff(1, 0)).toBe(1);
    expect(bitsDiff(1, 2)).toBe(2);
    expect(bitsDiff(1, 3)).toBe(1);
  });
});
```

Listing 46: clearBit.test.js

```
import clearBit from '../clearBit';

describe('clearBit', () => {
  it('should clear bit at specific position', () => {
    // 1 = 0b0001
    expect(clearBit(1, 0)).toBe(0);
    expect(clearBit(1, 1)).toBe(1);
    expect(clearBit(1, 2)).toBe(1);

    // 10 = 0b1010
    expect(clearBit(10, 0)).toBe(10);
    expect(clearBit(10, 1)).toBe(8);
    expect(clearBit(10, 3)).toBe(2);
  });
});
```

Listing 47: countSetBits.test.js

```
import countSetBits from '../countSetBits';

describe('countSetBits', () => {
  it('should return number of set bits', () => {
    expect(countSetBits(0)).toBe(0);
    expect(countSetBits(1)).toBe(1);
    expect(countSetBits(2)).toBe(1);
    expect(countSetBits(3)).toBe(2);
    expect(countSetBits(4)).toBe(1);
    expect(countSetBits(5)).toBe(2);
    expect(countSetBits(21)).toBe(3);
    expect(countSetBits(255)).toBe(8);
    expect(countSetBits(1023)).toBe(10);
  });
});
```

Listing 48: divideByTwo.test.js

```
import divideByTwo from '../divideByTwo';

describe('divideByTwo', () => {
  it('should divide numbers by two using bitwise operations', () => {
    expect(divideByTwo(0)).toBe(0);
    expect(divideByTwo(1)).toBe(0);
    expect(divideByTwo(3)).toBe(1);
    expect(divideByTwo(10)).toBe(5);
    expect(divideByTwo(17)).toBe(8);
    expect(divideByTwo(125)).toBe(62);
  });
});
```

Listing 49: fullAdder.test.js

```
import fullAdder from '../fullAdder';

describe('fullAdder', () => {
  it('should add up two numbers', () => {
    expect(fullAdder(0, 0)).toBe(0);
    expect(fullAdder(2, 0)).toBe(2);
    expect(fullAdder(0, 2)).toBe(2);
    expect(fullAdder(1, 2)).toBe(3);
    expect(fullAdder(2, 1)).toBe(3);
    expect(fullAdder(6, 6)).toBe(12);
    expect(fullAdder(-2, 4)).toBe(2);
    expect(fullAdder(4, -2)).toBe(2);
    expect(fullAdder(-4, -4)).toBe(-8);
    expect(fullAdder(4, -5)).toBe(-1);
    expect(fullAdder(2, 121)).toBe(123);
    expect(fullAdder(121, 2)).toBe(123);
  });
});
```

Listing 50: getBit.test.js

```
import getBit from '../getBit';

describe('getBit', () => {
  it('should get bit at specific position', () => {
    // 1 = 0b0001
    expect(getBit(1, 0)).toBe(1);
    expect(getBit(1, 1)).toBe(0);

    // 2 = 0b0010
    expect(getBit(2, 0)).toBe(0);
    expect(getBit(2, 1)).toBe(1);

    // 3 = 0b0011
    expect(getBit(3, 0)).toBe(1);
    expect(getBit(3, 1)).toBe(1);

    // 10 = 0b1010
    expect(getBit(10, 0)).toBe(0);
    expect(getBit(10, 1)).toBe(1);
    expect(getBit(10, 2)).toBe(0);
    expect(getBit(10, 3)).toBe(1);
  });
});
```

Listing 51: isEven.test.js

```
import isEven from '../isEven';

describe('isEven', () => {
  it('should detect if a number is even', () => {
    expect(isEven(0)).toBe(true);
    expect(isEven(2)).toBe(true);
    expect(isEven(-2)).toBe(true);
    expect(isEven(1)).toBe(false);
    expect(isEven(-1)).toBe(false);
    expect(isEven(-3)).toBe(false);
    expect(isEven(3)).toBe(false);
    expect(isEven(8)).toBe(true);
    expect(isEven(9)).toBe(false);
    expect(isEven(121)).toBe(false);
    expect(isEven(122)).toBe(true);
    expect(isEven(1201)).toBe(false);
    expect(isEven(1202)).toBe(true);
  });
});
```

Listing 52: isPositive.test.js

```
import isPositive from '../isPositive';

describe('isPositive', () => {
  it('should detect if a number is positive', () => {
    expect(isPositive(1)).toBe(true);
    expect(isPositive(2)).toBe(true);
    expect(isPositive(3)).toBe(true);
    expect(isPositive(5665)).toBe(true);
    expect(isPositive(56644325)).toBe(true);

    expect(isPositive(0)).toBe(false);
    expect(isPositive(-0)).toBe(false);
    expect(isPositive(-1)).toBe(false);
    expect(isPositive(-2)).toBe(false);
    expect(isPositive(-126)).toBe(false);
    expect(isPositive(-5665)).toBe(false);
    expect(isPositive(-56644325)).toBe(false);
  });
});
```

Listing 53: isPowerOfTwo.test.js

```
import isPowerOfTwo from '../isPowerOfTwo';

describe('isPowerOfTwo', () => {
  it('should detect if the number is power of two', () => {
    expect(isPowerOfTwo(1)).toBe(true);
    expect(isPowerOfTwo(2)).toBe(true);
    expect(isPowerOfTwo(3)).toBe(false);
    expect(isPowerOfTwo(4)).toBe(true);
    expect(isPowerOfTwo(5)).toBe(false);
    expect(isPowerOfTwo(6)).toBe(false);
    expect(isPowerOfTwo(7)).toBe(false);
    expect(isPowerOfTwo(8)).toBe(true);
    expect(isPowerOfTwo(9)).toBe(false);
    expect(isPowerOfTwo(16)).toBe(true);
    expect(isPowerOfTwo(23)).toBe(false);
    expect(isPowerOfTwo(32)).toBe(true);
    expect(isPowerOfTwo(127)).toBe(false);
    expect(isPowerOfTwo(128)).toBe(true);
  });
});
```

Listing 54: multiply.test.js

```
import multiply from '../multiply';

describe('multiply', () => {
  it('should multiply two numbers', () => {
    expect(multiply(0, 0)).toBe(0);
    expect(multiply(2, 0)).toBe(0);
    expect(multiply(0, 2)).toBe(0);
    expect(multiply(1, 2)).toBe(2);
    expect(multiply(2, 1)).toBe(2);
    expect(multiply(6, 6)).toBe(36);
    expect(multiply(-2, 4)).toBe(-8);
    expect(multiply(4, -2)).toBe(-8);
    expect(multiply(-4, -4)).toBe(16);
    expect(multiply(4, -5)).toBe(-20);
    expect(multiply(2, 121)).toBe(242);
    expect(multiply(121, 2)).toBe(242);
  });
});
```

Listing 55: multiplyByTwo.test.js

```
import multiplyByTwo from '../multiplyByTwo';

describe('multiplyByTwo', () => {
  it('should multiply numbers by two using bitwise operations', () => {
    expect(multiplyByTwo(0)).toBe(0);
    expect(multiplyByTwo(1)).toBe(2);
    expect(multiplyByTwo(3)).toBe(6);
    expect(multiplyByTwo(10)).toBe(20);
    expect(multiplyByTwo(17)).toBe(34);
    expect(multiplyByTwo(125)).toBe(250);
  });
});
```

Listing 56: multiplyUnsigned.test.js

```
import multiplyUnsigned from '../multiplyUnsigned';

describe('multiplyUnsigned', () => {
  it('should multiply two unsigned numbers', () => {
    expect(multiplyUnsigned(0, 2)).toBe(0);
    expect(multiplyUnsigned(2, 0)).toBe(0);
    expect(multiplyUnsigned(1, 1)).toBe(1);
    expect(multiplyUnsigned(1, 2)).toBe(2);
    expect(multiplyUnsigned(2, 7)).toBe(14);
    expect(multiplyUnsigned(7, 2)).toBe(14);
    expect(multiplyUnsigned(30, 2)).toBe(60);
    expect(multiplyUnsigned(17, 34)).toBe(578);
    expect(multiplyUnsigned(170, 2340)).toBe(397800);
  });
});
```

Listing 57: setBit.test.js

```
import setBit from '../setBit';

describe('setBit', () => {
  it('should set bit at specific position', () => {
    // 1 = 0b0001
    expect(setBit(1, 0)).toBe(1);
    expect(setBit(1, 1)).toBe(3);
    expect(setBit(1, 2)).toBe(5);

    // 10 = 0b1010
    expect(setBit(10, 0)).toBe(11);
    expect(setBit(10, 1)).toBe(10);
    expect(setBit(10, 2)).toBe(14);
  });
});
```

Listing 58: switchSign.test.js

```
import switchSign from '../switchSign';

describe('switchSign', () => {
  it('should switch the sign of the number using twos complement approach', () => {
    expect(switchSign(0)).toBe(0);
    expect(switchSign(1)).toBe(-1);
    expect(switchSign(-1)).toBe(1);
    expect(switchSign(32)).toBe(-32);
    expect(switchSign(-32)).toBe(32);
    expect(switchSign(23)).toBe(-23);
    expect(switchSign(-23)).toBe(23);
  });
});
```

Listing 59: updateBit.test.js

```
import updateBit from '../updateBit';

describe('updateBit', () => {
  it('should update bit at specific position', () => {
    // 1 = 0b0001
    expect(updateBit(1, 0, 1)).toBe(1);
    expect(updateBit(1, 0, 0)).toBe(0);
    expect(updateBit(1, 1, 1)).toBe(3);
    expect(updateBit(1, 2, 1)).toBe(5);

    // 10 = 0b1010
    expect(updateBit(10, 0, 1)).toBe(11);
    expect(updateBit(10, 0, 0)).toBe(10);
    expect(updateBit(10, 1, 1)).toBe(10);
    expect(updateBit(10, 1, 0)).toBe(8);
    expect(updateBit(10, 2, 1)).toBe(14);
    expect(updateBit(10, 2, 0)).toBe(10);
  });
});
```

Listing 60: bitLength.js

```
/**
 * Return the number of bits used in the binary representation of the number.
 *
 * @param {number} number
 * @return {number}
 */
export default function bitLength(number) {
  let bitsCounter = 0;

  while ((1 << bitsCounter) <= number) {
    bitsCounter += 1;
  }

  return bitsCounter;
}
```

Listing 61: bitsDiff.js

```
import countSetBits from './countSetBits';

/**
 * Counts the number of bits that need to be change in order
 * to convert numberA to numberB.
 *
 * @param {number} numberA
 * @param {number} numberB
 * @return {number}
 */
export default function bitsDiff(numberA, numberB) {
  return countSetBits(numberA ^ numberB);
}
```

Listing 62: clearBit.js

```
/**
 * @param {number} number
 * @param {number} bitPosition - zero based.
 * @return {number}
 */
export default function clearBit(number, bitPosition) {
  const mask = ~(1 << bitPosition);

  return number & mask;
}
```

Listing 63: countSetBits.js

```
/**
 * @param {number} originalNumber
 * @return {number}
 */
export default function countSetBits(originalNumber) {
  let setBitsCount = 0;
  let number = originalNumber;

  while (number) {
    // Add last bit of the number to the sum of set bits.
    setBitsCount += number & 1;

    // Shift number right by one bit to investigate other bits.
    number >>= 1;
  }

  return setBitsCount;
}
```

Listing 64: divideByTwo.js

```
/**
 * @param {number} number
 * @return {number}
 */
export default function divideByTwo(number) {
  return number >> 1;
}
```

Listing 65: fullAdder.js

```
import getBit from './getBit';

/**
 * Add two numbers using only binary operators.
 *
 * This is an implementation of full adders logic circuit.
 * https://en.wikipedia.org/wiki/Adder_(electronics)
 * Inspired by: https://www.youtube.com/watch?v=wvJc9CZcvBc
 *
 * Table(1)
 *  INPUT  | OUT
 *  C Ai Bi | C Si | Row
 * -------- | -----| ---
 *  0  0  0 | 0  0 | 1
 *  0  0  1 | 0  1 | 2
 *  0  1  0 | 0  1 | 3
 *  0  1  1 | 1  0 | 4
 * -------- | ---- | --
 *  1  0  0 | 0  1 | 5
 *  1  0  1 | 1  0 | 6
 *  1  1  0 | 1  0 | 7
 *  1  1  1 | 1  1 | 8
 * ---------------------
 *
 * Legend:
 * INPUT C = Carry in, from the previous less-significant stage
 * INPUT Ai = ith bit of Number A
 * INPUT Bi = ith bit of Number B
 * OUT C = Carry out to the next most-significant stage
 * OUT Si = Bit Sum, ith least significant bit of the result
 *
 *
 * @param {number} a
 * @param {number} b
 * @return {number}
 */
export default function fullAdder(a, b) {
  let result = 0;
  let carry = 0;

  // The operands of all bitwise operators are converted to signed
  // 32-bit integers in two's complement format.
  // https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators#Signed_32
      -bit_integers
  for (let i = 0; i < 32; i += 1) {
    const ai = getBit(a, i);
    const bi = getBit(b, i);
    const carryIn = carry;

    // Calculate binary Ai + Bi without carry (half adder)
    // See Table(1) rows 1 - 4: Si = Ai ^ Bi
    const aiPlusBi = ai ^ bi;

    // Calculate ith bit of the result by adding the carry bit to Ai + Bi
    // For Table(1) rows 5 - 8 carryIn = 1: Si = Ai ^ Bi ^ 1, flip the bit
    // Fpr Table(1) rows 1 - 4 carryIn = 0: Si = Ai ^ Bi ^ 0, a no-op.
    const bitSum = aiPlusBi ^ carryIn;

    // Carry out one to the next most-significant stage
    // when at least one of these is true:
    // 1) Table(1) rows 6, 7: one of Ai OR Bi is 1 AND carryIn = 1
    // 2) Table(1) rows 4, 8: Both Ai AND Bi are 1
    const carryOut = (aiPlusBi & carryIn) | (ai & bi);
    carry = carryOut;

    // Set ith least significant bit of the result to bitSum.
    result |= bitSum << i;
  }

  return result;
}
```

Listing 66: getBit.js

```
/**
 * @param {number} number
 * @param {number} bitPosition - zero based.
 * @return {number}
 */
export default function getBit(number, bitPosition) {
  return (number >> bitPosition) & 1;
}
```

Listing 67: isEven.js

```
/**
 * @param {number} number
 * @return {boolean}
 */
export default function isEven(number) {
  return (number & 1) === 0;
}
```

Listing 68: isPositive.js

```
/**
 * @param {number} number - 32-bit integer.
 * @return {boolean}
 */
export default function isPositive(number) {
  // Zero is neither a positive nor a negative number.
  if (number === 0) {
    return false;
  }

  // The most significant 32nd bit can be used to determine whether the number is positive.
  return ((number >> 31) & 1) === 0;
}
```

Listing 69: isPowerOfTwo.js

```
/**
 * @param {number} number
 * @return bool
 */
export default function isPowerOfTwo(number) {
  return (number & (number - 1)) === 0;
}
```

Listing 70: multiply.js

```
import multiplyByTwo from './multiplyByTwo';
import divideByTwo from './divideByTwo';
import isEven from './isEven';
import isPositive from './isPositive';

/**
 * Multiply two signed numbers using bitwise operations.
 *
 * If a is zero or b is zero or if both a and b are zeros:
 * multiply(a, b) = 0
 *
 * If b is even:
 * multiply(a, b) = multiply(2a, b/2)
 *
 * If b is odd and b is positive:
 * multiply(a, b) = multiply(2a, (b-1)/2) + a
 *
 * If b is odd and b is negative:
 * multiply(a, b) = multiply(2a, (b+1)/2) - a
 *
 * Time complexity: O(log b)
 *
 * @param {number} a
 * @param {number} b
 * @return {number}
 */
export default function multiply(a, b) {
  // If a is zero or b is zero or if both a and b are zeros then the production is also zero.
  if (b === 0 || a === 0) {
    return 0;
  }

  // Otherwise we will have four different cases that are described above.
  const multiplyByOddPositive = () => multiply(multiplyByTwo(a), divideByTwo(b - 1)) + a;
  const multiplyByOddNegative = () => multiply(multiplyByTwo(a), divideByTwo(b + 1)) - a;

  const multiplyByEven = () => multiply(multiplyByTwo(a), divideByTwo(b));
  const multiplyByOdd = () => (isPositive(b) ? multiplyByOddPositive() : multiplyByOddNegative());

  return isEven(b) ? multiplyByEven() : multiplyByOdd();
}
```

Listing 71: multiplyByTwo.js

```
/**
 * @param {number} number
 * @return {number}
 */
export default function multiplyByTwo(number) {
  return number << 1;
}
```

Listing 72: multiplyUnsigned.js

```
/**
 * Multiply to unsigned numbers using bitwise operator.
 *
 * The main idea of bitwise multiplication is that every number may be split
 * to the sum of powers of two:
 *
 * I.e. 19 = 2^4 + 2^1 + 2^0
 *
 * Then multiplying number x by 19 is equivalent of:
 *
 * x * 19 = x * 2^4 + x * 2^1 + x * 2^0
 *
 * Now we need to remember that (x * 2^4) is equivalent of shifting x left by 4 bits (x << 4).
 *
 * @param {number} number1
 * @param {number} number2
 * @return {number}
 */
export default function multiplyUnsigned(number1, number2) {
  let result = 0;

  // Let's treat number2 as a multiplier for the number1.
  let multiplier = number2;

  // Multiplier current bit index.
  let bitIndex = 0;

  // Go through all bits of number2.
  while (multiplier !== 0) {
    // Check if current multiplier bit is set.
    if (multiplier & 1) {
      // In case if multiplier's bit at position bitIndex is set
      // it would mean that we need to multiply number1 by the power
      // of bit with index bitIndex and then add it to the result.
      result += (number1 << bitIndex);
    }

    bitIndex += 1;
    multiplier >>= 1;
  }

  return result;
}
```

Listing 73: setBit.js

```
/**
 * @param {number} number
 * @param {number} bitPosition - zero based.
 * @return {number}
 */
export default function setBit(number, bitPosition) {
  return number | (1 << bitPosition);
}
```

Listing 74: switchSign.js

```
/**
 * Switch the sign of the number using "Twos Complement" approach.
 * @param {number} number
 * @return {number}
 */
export default function switchSign(number) {
  return ~number + 1;
}
```

Listing 75: updateBit.js

```
/**
 * @param {number} number
 * @param {number} bitPosition - zero based.
 * @param {number} bitValue - 0 or 1.
 * @return {number}
 */
export default function updateBit(number, bitPosition, bitValue) {
  // Normalized bit value.
  const bitValueNormalized = bitValue ? 1 : 0;

  // Init clear mask.
  const clearMask = ~(1 << bitPosition);

  // Clear bit value and then set it up to required value.
  return (number & clearMask) | (bitValueNormalized << bitPosition);
}
```

Listing 76: ComplexNumber.test.js

```
import ComplexNumber from '../ComplexNumber';

describe('ComplexNumber', () => {
  it('should create complex numbers', () => {
    const complexNumber = new ComplexNumber({ re: 1, im: 2 });

    expect(complexNumber).toBeDefined();
    expect(complexNumber.re).toBe(1);
    expect(complexNumber.im).toBe(2);

    const defaultComplexNumber = new ComplexNumber();
    expect(defaultComplexNumber.re).toBe(0);
    expect(defaultComplexNumber.im).toBe(0);
  });

  it('should add complex numbers', () => {
    const complexNumber1 = new ComplexNumber({ re: 1, im: 2 });
    const complexNumber2 = new ComplexNumber({ re: 3, im: 8 });

    const complexNumber3 = complexNumber1.add(complexNumber2);
    const complexNumber4 = complexNumber2.add(complexNumber1);

    expect(complexNumber3.re).toBe(1 + 3);
    expect(complexNumber3.im).toBe(2 + 8);

    expect(complexNumber4.re).toBe(1 + 3);
    expect(complexNumber4.im).toBe(2 + 8);
  });

  it('should add complex and natural numbers', () => {
    const complexNumber = new ComplexNumber({ re: 1, im: 2 });
    const realNumber = new ComplexNumber({ re: 3 });

    const complexNumber3 = complexNumber.add(realNumber);
    const complexNumber4 = realNumber.add(complexNumber);
    const complexNumber5 = complexNumber.add(3);

    expect(complexNumber3.re).toBe(1 + 3);
    expect(complexNumber3.im).toBe(2);

    expect(complexNumber4.re).toBe(1 + 3);
    expect(complexNumber4.im).toBe(2);

    expect(complexNumber5.re).toBe(1 + 3);
    expect(complexNumber5.im).toBe(2);
  });

  it('should subtract complex numbers', () => {
    const complexNumber1 = new ComplexNumber({ re: 1, im: 2 });
    const complexNumber2 = new ComplexNumber({ re: 3, im: 8 });

    const complexNumber3 = complexNumber1.subtract(complexNumber2);
    const complexNumber4 = complexNumber2.subtract(complexNumber1);

    expect(complexNumber3.re).toBe(1 - 3);
    expect(complexNumber3.im).toBe(2 - 8);

    expect(complexNumber4.re).toBe(3 - 1);
    expect(complexNumber4.im).toBe(8 - 2);
  });

  it('should subtract complex and natural numbers', () => {
    const complexNumber = new ComplexNumber({ re: 1, im: 2 });
    const realNumber = new ComplexNumber({ re: 3 });

    const complexNumber3 = complexNumber.subtract(realNumber);
    const complexNumber4 = realNumber.subtract(complexNumber);
    const complexNumber5 = complexNumber.subtract(3);

    expect(complexNumber3.re).toBe(1 - 3);
    expect(complexNumber3.im).toBe(2);

    expect(complexNumber4.re).toBe(3 - 1);
    expect(complexNumber4.im).toBe(-2);

    expect(complexNumber5.re).toBe(1 - 3);
    expect(complexNumber5.im).toBe(2);
  });
```

```
it('should multiply complex numbers', () => {
  const complexNumber1 = new ComplexNumber({ re: 3, im: 2 });
  const complexNumber2 = new ComplexNumber({ re: 1, im: 7 });

  const complexNumber3 = complexNumber1.multiply(complexNumber2);
  const complexNumber4 = complexNumber2.multiply(complexNumber1);
  const complexNumber5 = complexNumber1.multiply(5);

  expect(complexNumber3.re).toBe(-11);
  expect(complexNumber3.im).toBe(23);

  expect(complexNumber4.re).toBe(-11);
  expect(complexNumber4.im).toBe(23);

  expect(complexNumber5.re).toBe(15);
  expect(complexNumber5.im).toBe(10);
});

it('should multiply complex numbers by themselves', () => {
  const complexNumber = new ComplexNumber({ re: 1, im: 1 });

  const result = complexNumber.multiply(complexNumber);

  expect(result.re).toBe(0);
  expect(result.im).toBe(2);
});

it('should calculate i in power of two', () => {
  const complexNumber = new ComplexNumber({ re: 0, im: 1 });

  const result = complexNumber.multiply(complexNumber);

  expect(result.re).toBe(-1);
  expect(result.im).toBe(0);
});

it('should divide complex numbers', () => {
  const complexNumber1 = new ComplexNumber({ re: 2, im: 3 });
  const complexNumber2 = new ComplexNumber({ re: 4, im: -5 });

  const complexNumber3 = complexNumber1.divide(complexNumber2);
  const complexNumber4 = complexNumber1.divide(2);

  expect(complexNumber3.re).toBe(-7 / 41);
  expect(complexNumber3.im).toBe(22 / 41);

  expect(complexNumber4.re).toBe(1);
  expect(complexNumber4.im).toBe(1.5);
});

it('should return complex number in polar form', () => {
  const complexNumber1 = new ComplexNumber({ re: 3, im: 3 });
  expect(complexNumber1.getPolarForm().radius).toBe(Math.sqrt((3 ** 2) + (3 ** 2)));
  expect(complexNumber1.getPolarForm().phase).toBe(Math.PI / 4);
  expect(complexNumber1.getPolarForm(false).phase).toBe(45);

  const complexNumber2 = new ComplexNumber({ re: -3, im: 3 });
  expect(complexNumber2.getPolarForm().radius).toBe(Math.sqrt((3 ** 2) + (3 ** 2)));
  expect(complexNumber2.getPolarForm().phase).toBe(3 * (Math.PI / 4));
  expect(complexNumber2.getPolarForm(false).phase).toBe(135);

  const complexNumber3 = new ComplexNumber({ re: -3, im: -3 });
  expect(complexNumber3.getPolarForm().radius).toBe(Math.sqrt((3 ** 2) + (3 ** 2)));
  expect(complexNumber3.getPolarForm().phase).toBe(-3 * (Math.PI / 4));
  expect(complexNumber3.getPolarForm(false).phase).toBe(-135);

  const complexNumber4 = new ComplexNumber({ re: 3, im: -3 });
  expect(complexNumber4.getPolarForm().radius).toBe(Math.sqrt((3 ** 2) + (3 ** 2)));
  expect(complexNumber4.getPolarForm().phase).toBe(-1 * (Math.PI / 4));
  expect(complexNumber4.getPolarForm(false).phase).toBe(-45);

  const complexNumber5 = new ComplexNumber({ re: 5, im: 7 });
  expect(complexNumber5.getPolarForm().radius).toBeCloseTo(8.60);
  expect(complexNumber5.getPolarForm().phase).toBeCloseTo(0.95);
  expect(complexNumber5.getPolarForm(false).phase).toBeCloseTo(54.46);

  const complexNumber6 = new ComplexNumber({ re: 0, im: 0.25 });
  expect(complexNumber6.getPolarForm().radius).toBeCloseTo(0.25);
  expect(complexNumber6.getPolarForm().phase).toBeCloseTo(1.57);
  expect(complexNumber6.getPolarForm(false).phase).toBeCloseTo(90);

  const complexNumber7 = new ComplexNumber({ re: 0, im: -0.25 });
```

```
    expect(complexNumber7.getPolarForm().radius).toBeCloseTo(0.25);
    expect(complexNumber7.getPolarForm().phase).toBeCloseTo(-1.57);
    expect(complexNumber7.getPolarForm(false).phase).toBeCloseTo(-90);

    const complexNumber8 = new ComplexNumber();
    expect(complexNumber8.getPolarForm().radius).toBeCloseTo(0);
    expect(complexNumber8.getPolarForm().phase).toBeCloseTo(0);
    expect(complexNumber8.getPolarForm(false).phase).toBeCloseTo(0);

    const complexNumber9 = new ComplexNumber({ re: -0.25, im: 0 });
    expect(complexNumber9.getPolarForm().radius).toBeCloseTo(0.25);
    expect(complexNumber9.getPolarForm().phase).toBeCloseTo(Math.PI);
    expect(complexNumber9.getPolarForm(false).phase).toBeCloseTo(180);

    const complexNumber10 = new ComplexNumber({ re: 0.25, im: 0 });
    expect(complexNumber10.getPolarForm().radius).toBeCloseTo(0.25);
    expect(complexNumber10.getPolarForm().phase).toBeCloseTo(0);
    expect(complexNumber10.getPolarForm(false).phase).toBeCloseTo(0);
  });
});
```

Listing 77: ComplexNumber.js

```
import radianToDegree from '../radian/radianToDegree';

export default class ComplexNumber {
  /**
   * z = re + im * i
   * z = radius * e^(i * phase)
   *
   * @param {number} [re]
   * @param {number} [im]
   */
  constructor({ re = 0, im = 0 } = {}) {
    this.re = re;
    this.im = im;
  }

  /**
   * @param {ComplexNumber|number} addend
   * @return {ComplexNumber}
   */
  add(addend) {
    // Make sure we're dealing with complex number.
    const complexAddend = this.toComplexNumber(addend);

    return new ComplexNumber({
      re: this.re + complexAddend.re,
      im: this.im + complexAddend.im,
    });
  }

  /**
   * @param {ComplexNumber|number} subtrahend
   * @return {ComplexNumber}
   */
  subtract(subtrahend) {
    // Make sure we're dealing with complex number.
    const complexSubtrahend = this.toComplexNumber(subtrahend);

    return new ComplexNumber({
      re: this.re - complexSubtrahend.re,
      im: this.im - complexSubtrahend.im,
    });
  }

  /**
   * @param {ComplexNumber|number} multiplicand
   * @return {ComplexNumber}
   */
  multiply(multiplicand) {
    // Make sure we're dealing with complex number.
    const complexMultiplicand = this.toComplexNumber(multiplicand);

    return new ComplexNumber({
      re: this.re * complexMultiplicand.re - this.im * complexMultiplicand.im,
      im: this.re * complexMultiplicand.im + this.im * complexMultiplicand.re,
    });
  }

  /**
   * @param {ComplexNumber|number} divider
   * @return {ComplexNumber}
   */
  divide(divider) {
    // Make sure we're dealing with complex number.
    const complexDivider = this.toComplexNumber(divider);

    // Get divider conjugate.
    const dividerConjugate = this.conjugate(complexDivider);

    // Multiply dividend by divider's conjugate.
    const finalDivident = this.multiply(dividerConjugate);

    // Calculating final divider using formula (a + bi)(a  bi) = a^2 + b^2
    const finalDivider = (complexDivider.re ** 2) + (complexDivider.im ** 2);

    return new ComplexNumber({
      re: finalDivident.re / finalDivider,
      im: finalDivident.im / finalDivider,
    });
  }
```

```
  /**
   * @param {ComplexNumber|number} number
   */
  conjugate(number) {
    // Make sure we're dealing with complex number.
    const complexNumber = this.toComplexNumber(number);

    return new ComplexNumber({
      re: complexNumber.re,
      im: -1 * complexNumber.im,
    });
  }

  /**
   * @return {number}
   */
  getRadius() {
    return Math.sqrt((this.re ** 2) + (this.im ** 2));
  }

  /**
   * @param {boolean} [inRadians]
   * @return {number}
   */
  getPhase(inRadians = true) {
    let phase = Math.atan(Math.abs(this.im) / Math.abs(this.re));

    if (this.re < 0 && this.im > 0) {
      phase = Math.PI - phase;
    } else if (this.re < 0 && this.im < 0) {
      phase = -(Math.PI - phase);
    } else if (this.re > 0 && this.im < 0) {
      phase = -phase;
    } else if (this.re === 0 && this.im > 0) {
      phase = Math.PI / 2;
    } else if (this.re === 0 && this.im < 0) {
      phase = -Math.PI / 2;
    } else if (this.re < 0 && this.im === 0) {
      phase = Math.PI;
    } else if (this.re > 0 && this.im === 0) {
      phase = 0;
    } else if (this.re === 0 && this.im === 0) {
      // More correctly would be to set 'indeterminate'.
      // But just for simplicity reasons let's set zero.
      phase = 0;
    }

    if (!inRadians) {
      phase = radianToDegree(phase);
    }

    return phase;
  }

  /**
   * @param {boolean} [inRadians]
   * @return {{radius: number, phase: number}}
   */
  getPolarForm(inRadians = true) {
    return {
      radius: this.getRadius(),
      phase: this.getPhase(inRadians),
    };
  }

  /**
   * Convert real numbers to complex number.
   * In case if complex number is provided then lefts it as is.
   *
   * @param {ComplexNumber|number} number
   * @return {ComplexNumber}
   */
  toComplexNumber(number) {
    if (number instanceof ComplexNumber) {
      return number;
    }

    return new ComplexNumber({ re: number });
  }
}
```

Listing 78: euclideanAlgorithm.test.js

```
import euclideanAlgorithm from '../euclideanAlgorithm';

describe('euclideanAlgorithm', () => {
  it('should calculate GCD recursively', () => {
    expect(euclideanAlgorithm(0, 0)).toBe(0);
    expect(euclideanAlgorithm(2, 0)).toBe(2);
    expect(euclideanAlgorithm(0, 2)).toBe(2);
    expect(euclideanAlgorithm(1, 2)).toBe(1);
    expect(euclideanAlgorithm(2, 1)).toBe(1);
    expect(euclideanAlgorithm(6, 6)).toBe(6);
    expect(euclideanAlgorithm(2, 4)).toBe(2);
    expect(euclideanAlgorithm(4, 2)).toBe(2);
    expect(euclideanAlgorithm(12, 4)).toBe(4);
    expect(euclideanAlgorithm(4, 12)).toBe(4);
    expect(euclideanAlgorithm(5, 13)).toBe(1);
    expect(euclideanAlgorithm(27, 13)).toBe(1);
    expect(euclideanAlgorithm(24, 60)).toBe(12);
    expect(euclideanAlgorithm(60, 24)).toBe(12);
    expect(euclideanAlgorithm(252, 105)).toBe(21);
    expect(euclideanAlgorithm(105, 252)).toBe(21);
    expect(euclideanAlgorithm(1071, 462)).toBe(21);
    expect(euclideanAlgorithm(462, 1071)).toBe(21);
    expect(euclideanAlgorithm(462, -1071)).toBe(21);
    expect(euclideanAlgorithm(-462, -1071)).toBe(21);
  });
});
```

Listing 79: euclideanAlgorithmIterative.test.js

```
import euclideanAlgorithmIterative from '../euclideanAlgorithmIterative';

describe('euclideanAlgorithmIterative', () => {
  it('should calculate GCD iteratively', () => {
    expect(euclideanAlgorithmIterative(0, 0)).toBe(0);
    expect(euclideanAlgorithmIterative(2, 0)).toBe(2);
    expect(euclideanAlgorithmIterative(0, 2)).toBe(2);
    expect(euclideanAlgorithmIterative(1, 2)).toBe(1);
    expect(euclideanAlgorithmIterative(2, 1)).toBe(1);
    expect(euclideanAlgorithmIterative(6, 6)).toBe(6);
    expect(euclideanAlgorithmIterative(2, 4)).toBe(2);
    expect(euclideanAlgorithmIterative(4, 2)).toBe(2);
    expect(euclideanAlgorithmIterative(12, 4)).toBe(4);
    expect(euclideanAlgorithmIterative(4, 12)).toBe(4);
    expect(euclideanAlgorithmIterative(5, 13)).toBe(1);
    expect(euclideanAlgorithmIterative(27, 13)).toBe(1);
    expect(euclideanAlgorithmIterative(24, 60)).toBe(12);
    expect(euclideanAlgorithmIterative(60, 24)).toBe(12);
    expect(euclideanAlgorithmIterative(252, 105)).toBe(21);
    expect(euclideanAlgorithmIterative(105, 252)).toBe(21);
    expect(euclideanAlgorithmIterative(1071, 462)).toBe(21);
    expect(euclideanAlgorithmIterative(462, 1071)).toBe(21);
    expect(euclideanAlgorithmIterative(462, -1071)).toBe(21);
    expect(euclideanAlgorithmIterative(-462, -1071)).toBe(21);
  });
});
```

Listing 80: euclideanAlgorithm.js

```
/**
 * Recursive version of Euclidean Algorithm of finding greatest common divisor (GCD).
 * @param {number} originalA
 * @param {number} originalB
 * @return {number}
 */
export default function euclideanAlgorithm(originalA, originalB) {
  // Make input numbers positive.
  const a = Math.abs(originalA);
  const b = Math.abs(originalB);

  // To make algorithm work faster instead of subtracting one number from the other
  // we may use modulo operation.
  return (b === 0) ? a : euclideanAlgorithm(b, a % b);
}
```

Listing 81: euclideanAlgorithmIterative.js

```
/**
 * Iterative version of Euclidean Algorithm of finding greatest common divisor (GCD).
 * @param {number} originalA
 * @param {number} originalB
 * @return {number}
 */
export default function euclideanAlgorithmIterative(originalA, originalB) {
  // Make input numbers positive.
  let a = Math.abs(originalA);
  let b = Math.abs(originalB);

  // Subtract one number from another until both numbers would become the same.
  // This will be out GCD. Also quit the loop if one of the numbers is zero.
  while (a && b && a !== b) {
    [a, b] = a > b ? [a - b, b] : [a, b - a];
  }

  // Return the number that is not equal to zero since the last subtraction (it will be a GCD).
  return a || b;
}
```

Listing 82: factorial.test.js

```
import factorial from '../factorial';

describe('factorial', () => {
  it('should calculate factorial', () => {
    expect(factorial(0)).toBe(1);
    expect(factorial(1)).toBe(1);
    expect(factorial(5)).toBe(120);
    expect(factorial(8)).toBe(40320);
    expect(factorial(10)).toBe(3628800);
  });
});
```

Listing 83: factorialRecursive.test.js

```
import factorialRecursive from '../factorialRecursive';

describe('factorialRecursive', () => {
  it('should calculate factorial', () => {
    expect(factorialRecursive(0)).toBe(1);
    expect(factorialRecursive(1)).toBe(1);
    expect(factorialRecursive(5)).toBe(120);
    expect(factorialRecursive(8)).toBe(40320);
    expect(factorialRecursive(10)).toBe(3628800);
  });
});
```

Listing 84: factorial.js

```
/**
 * @param {number} number
 * @return {number}
 */
export default function factorial(number) {
  let result = 1;

  for (let i = 2; i <= number; i += 1) {
    result *= i;
  }

  return result;
}
```

Listing 85: factorialRecursive.js

```
/**
 * @param {number} number
 * @return {number}
 */
export default function factorialRecursive(number) {
  return number > 1 ? number * factorialRecursive(number - 1) : 1;
}
```

Listing 86: fastPowering.test.js

```
import fastPowering from '../fastPowering';

describe('fastPowering', () => {
  it('should compute power in log(n) time', () => {
    expect(fastPowering(1, 1)).toBe(1);
    expect(fastPowering(2, 0)).toBe(1);
    expect(fastPowering(2, 2)).toBe(4);
    expect(fastPowering(2, 3)).toBe(8);
    expect(fastPowering(2, 4)).toBe(16);
    expect(fastPowering(2, 5)).toBe(32);
    expect(fastPowering(2, 6)).toBe(64);
    expect(fastPowering(2, 7)).toBe(128);
    expect(fastPowering(2, 8)).toBe(256);
    expect(fastPowering(3, 4)).toBe(81);
    expect(fastPowering(190, 2)).toBe(36100);
    expect(fastPowering(11, 5)).toBe(161051);
    expect(fastPowering(13, 11)).toBe(1792160394037);
    expect(fastPowering(9, 16)).toBe(1853020188851841);
    expect(fastPowering(16, 16)).toBe(18446744073709552000);
    expect(fastPowering(7, 21)).toBe(558545864083284000);
    expect(fastPowering(100, 9)).toBe(1000000000000000000);
  });
});
```

Listing 87: fastPowering.js

```
/**
 * Fast Powering Algorithm.
 * Recursive implementation to compute power.
 *
 * Complexity: log(n)
 *
 * @param {number} base - Number that will be raised to the power.
 * @param {number} power - The power that number will be raised to.
 * @return {number}
 */
export default function fastPowering(base, power) {
  if (power === 0) {
    // Anything that is raised to the power of zero is 1.
    return 1;
  }

  if (power % 2 === 0) {
    // If the power is even...
    // we may recursively redefine the result via twice smaller powers:
    // x^8 = x^4 * x^4.
    const multiplier = fastPowering(base, power / 2);
    return multiplier * multiplier;
  }

  // If the power is odd...
  // we may recursively redefine the result via twice smaller powers:
  // x^9 = x^4 * x^4 * x.
  const multiplier = fastPowering(base, Math.floor(power / 2));
  return multiplier * multiplier * base;
}
```

Listing 88: fibonacci.test.js

```
import fibonacci from '../fibonacci';

describe('fibonacci', () => {
  it('should calculate fibonacci correctly', () => {
    expect(fibonacci(1)).toEqual([1]);
    expect(fibonacci(2)).toEqual([1, 1]);
    expect(fibonacci(3)).toEqual([1, 1, 2]);
    expect(fibonacci(4)).toEqual([1, 1, 2, 3]);
    expect(fibonacci(5)).toEqual([1, 1, 2, 3, 5]);
    expect(fibonacci(6)).toEqual([1, 1, 2, 3, 5, 8]);
    expect(fibonacci(7)).toEqual([1, 1, 2, 3, 5, 8, 13]);
    expect(fibonacci(8)).toEqual([1, 1, 2, 3, 5, 8, 13, 21]);
    expect(fibonacci(9)).toEqual([1, 1, 2, 3, 5, 8, 13, 21, 34]);
    expect(fibonacci(10)).toEqual([1, 1, 2, 3, 5, 8, 13, 21, 34, 55]);
  });
});
```

Listing 89: fibonacciNth.test.js

```
import fibonacciNth from '../fibonacciNth';

describe('fibonacciNth', () => {
  it('should calculate fibonacci correctly', () => {
    expect(fibonacciNth(1)).toBe(1);
    expect(fibonacciNth(2)).toBe(1);
    expect(fibonacciNth(3)).toBe(2);
    expect(fibonacciNth(4)).toBe(3);
    expect(fibonacciNth(5)).toBe(5);
    expect(fibonacciNth(6)).toBe(8);
    expect(fibonacciNth(7)).toBe(13);
    expect(fibonacciNth(8)).toBe(21);
    expect(fibonacciNth(20)).toBe(6765);
    expect(fibonacciNth(30)).toBe(832040);
    expect(fibonacciNth(50)).toBe(12586269025);
    expect(fibonacciNth(70)).toBe(190392490709135);
    expect(fibonacciNth(71)).toBe(308061521170129);
    expect(fibonacciNth(72)).toBe(498454011879264);
    expect(fibonacciNth(73)).toBe(806515533049393);
    expect(fibonacciNth(74)).toBe(1304969544928657);
    expect(fibonacciNth(75)).toBe(2111485077978050);
    expect(fibonacciNth(80)).toBe(23416728348467685);
    expect(fibonacciNth(90)).toBe(2880067194370816120);
  });
});
```

Listing 90: fibonacciNthClosedForm.test.js

```
import fibonacciNthClosedForm from '../fibonacciNthClosedForm';

describe('fibonacciClosedForm', () => {
  it('should throw an error when trying to calculate fibonacci for not allowed positions', () => {
    const calculateFibonacciForNotAllowedPosition = () => {
      fibonacciNthClosedForm(76);
    };

    expect(calculateFibonacciForNotAllowedPosition).toThrow();
  });

  it('should calculate fibonacci correctly', () => {
    expect(fibonacciNthClosedForm(1)).toBe(1);
    expect(fibonacciNthClosedForm(2)).toBe(1);
    expect(fibonacciNthClosedForm(3)).toBe(2);
    expect(fibonacciNthClosedForm(4)).toBe(3);
    expect(fibonacciNthClosedForm(5)).toBe(5);
    expect(fibonacciNthClosedForm(6)).toBe(8);
    expect(fibonacciNthClosedForm(7)).toBe(13);
    expect(fibonacciNthClosedForm(8)).toBe(21);
    expect(fibonacciNthClosedForm(20)).toBe(6765);
    expect(fibonacciNthClosedForm(30)).toBe(832040);
    expect(fibonacciNthClosedForm(50)).toBe(12586269025);
    expect(fibonacciNthClosedForm(70)).toBe(190392490709135);
  });
});
```

Listing 91: fibonacci.js

```
/**
* Return a fibonacci sequence as an array.
*
* @param n
* @return {number[]}
*/
export default function fibonacci(n) {
  const fibSequence = [1];

  let currentValue = 1;
  let previousValue = 0;

  if (n === 1) {
    return fibSequence;
  }

  let iterationsCounter = n - 1;

  while (iterationsCounter) {
    currentValue += previousValue;
    previousValue = currentValue - previousValue;

    fibSequence.push(currentValue);

    iterationsCounter -= 1;
  }

  return fibSequence;
}
```

Listing 92: fibonacciNth.js

```
/**
 * Calculate fibonacci number at specific position using Dynamic Programming approach.
 *
 * @param n
 * @return {number}
 */
export default function fibonacciNth(n) {
  let currentValue = 1;
  let previousValue = 0;

  if (n === 1) {
    return 1;
  }

  let iterationsCounter = n - 1;

  while (iterationsCounter) {
    currentValue += previousValue;
    previousValue = currentValue - previousValue;

    iterationsCounter -= 1;
  }

  return currentValue;
}
```

Listing 93: fibonacciNthClosedForm.js

```
/**
 * Calculate fibonacci number at specific position using closed form function (Binet's formula).
 * @see: https://en.wikipedia.org/wiki/Fibonacci_number#Closed-form_expression
 *
 * @param {number} position - Position number of fibonacci sequence (must be number from 1 to 75).
 * @return {number}
 */
export default function fibonacciClosedForm(position) {
  const topMaxValidPosition = 70;

  // Check that position is valid.
  if (position < 1 || position > topMaxValidPosition) {
    throw new Error(`Can't handle position smaller than 1 or greater than ${topMaxValidPosition}`);
  }

  // Calculate 5 to re-use it in further formulas.
  const sqrt5 = Math.sqrt(5);
  // Calculate  constant ( 1.61803).
  const phi = (1 + sqrt5) / 2;

  // Calculate fibonacci number using Binet's formula.
  return Math.floor((phi ** position) / sqrt5 + 0.5);
}
```

Listing 94: discreteFourierTransform.test.js

```
import discreteFourierTransform from '../discreteFourierTransform';
import FourierTester from './FourierTester';

describe('discreteFourierTransform', () => {
  it('should split signal into frequencies', () => {
    FourierTester.testDirectFourierTransform(discreteFourierTransform);
  });
});
```

Listing 95: fastFourierTransform.test.js

```
import fastFourierTransform from '../fastFourierTransform';
import ComplexNumber from '../../complex-number/ComplexNumber';

/**
 * @param {ComplexNumber[]} sequence1
 * @param {ComplexNumber[]} sequence2
 * @param {Number} delta
 * @return {boolean}
 */
function sequencesApproximatelyEqual(sequence1, sequence2, delta) {
  if (sequence1.length !== sequence2.length) {
    return false;
  }

  for (let numberIndex = 0; numberIndex < sequence1.length; numberIndex += 1) {
    if (Math.abs(sequence1[numberIndex].re - sequence2[numberIndex].re) > delta) {
      return false;
    }

    if (Math.abs(sequence1[numberIndex].im - sequence2[numberIndex].im) > delta) {
      return false;
    }
  }

  return true;
}

const delta = 1e-6;

describe('fastFourierTransform', () => {
  it('should calculate the radix-2 discrete fourier transform #1', () => {
    const input = [new ComplexNumber({ re: 0, im: 0 })];
    const expectedOutput = [new ComplexNumber({ re: 0, im: 0 })];
    const output = fastFourierTransform(input);
    const invertedOutput = fastFourierTransform(output, true);

    expect(sequencesApproximatelyEqual(expectedOutput, output, delta)).toBe(true);
    expect(sequencesApproximatelyEqual(input, invertedOutput, delta)).toBe(true);
  });

  it('should calculate the radix-2 discrete fourier transform #2', () => {
    const input = [
      new ComplexNumber({ re: 1, im: 2 }),
      new ComplexNumber({ re: 2, im: 3 }),
      new ComplexNumber({ re: 8, im: 4 }),
    ];

    const expectedOutput = [
      new ComplexNumber({ re: 11, im: 9 }),
      new ComplexNumber({ re: -10, im: 0 }),
      new ComplexNumber({ re: 7, im: 3 }),
      new ComplexNumber({ re: -4, im: -4 }),
    ];

    const output = fastFourierTransform(input);
    const invertedOutput = fastFourierTransform(output, true);

    expect(sequencesApproximatelyEqual(expectedOutput, output, delta)).toBe(true);
    expect(sequencesApproximatelyEqual(input, invertedOutput, delta)).toBe(true);
  });

  it('should calculate the radix-2 discrete fourier transform #3', () => {
    const input = [
      new ComplexNumber({ re: -83656.9359385182, im: 98724.08038374918 }),
      new ComplexNumber({ re: -47537.415125808424, im: 88441.58381765135 }),
      new ComplexNumber({ re: -24849.657029355192, im: -72621.79007878687 }),
      new ComplexNumber({ re: 31451.27290052717, im: -21113.301128347346 }),
      new ComplexNumber({ re: 13973.90836288876, im: -73378.36721594246 }),
      new ComplexNumber({ re: 14981.520420492234, im: 63279.524958963884 }),
      new ComplexNumber({ re: -9892.575367044381, im: -81748.44671677813 }),
      new ComplexNumber({ re: -35933.00356823792, im: -46153.47157161784 }),
      new ComplexNumber({ re: -22425.008561855735, im: -86284.24507370662 }),
      new ComplexNumber({ re: -39327.43830818355, im: 30611.949874562706 }),
    ];

    const expectedOutput = [
      new ComplexNumber({ re: -203215.3322151, im: -100242.4827503 }),
      new ComplexNumber({ re: 99217.0805705, im: 270646.9331932 }),
      new ComplexNumber({ re: -305990.9040412, im: 68224.8435751 }),
```

```
      new ComplexNumber({ re: -14135.7758282, im: 199223.9878095 }),
      new ComplexNumber({ re: -306965.6350922, im: 26030.1025439 }),
      new ComplexNumber({ re: -76477.6755206, im: 40781.9078990 }),
      new ComplexNumber({ re: -48409.3099088, im: 54674.7959662 }),
      new ComplexNumber({ re: -329683.0131713, im: 164287.7995937 }),
      new ComplexNumber({ re: -50485.2048527, im: -330375.0546527 }),
      new ComplexNumber({ re: 122235.7738708, im: 91091.6398019 }),
      new ComplexNumber({ re: 47625.8850387, im: 73497.3981523 }),
      new ComplexNumber({ re: -15619.8231136, im: 80804.8685410 }),
      new ComplexNumber({ re: 192234.0276101, im: 160833.3072355 }),
      new ComplexNumber({ re: -96389.4195635, im: 393408.4543872 }),
      new ComplexNumber({ re: -173449.0825417, im: 146875.7724104 }),
      new ComplexNumber({ re: -179002.5662573, im: 239821.0124341 }),
    ];

    const output = fastFourierTransform(input);
    const invertedOutput = fastFourierTransform(output, true);

    expect(sequencesApproximatelyEqual(expectedOutput, output, delta)).toBe(true);
    expect(sequencesApproximatelyEqual(input, invertedOutput, delta)).toBe(true);
  });
});
```

Listing 96: FourierTester.js

```
 import ComplexNumber from '../../complex-number/ComplexNumber';

export const fourierTestCases = [
  {
    input: [
      { amplitude: 1 },
    ],
    output: [
      {
        frequency: 0, amplitude: 1, phase: 0, re: 1, im: 0,
      },
    ],
  },
  {
    input: [
      { amplitude: 1 },
      { amplitude: 0 },
    ],
    output: [
      {
        frequency: 0, amplitude: 0.5, phase: 0, re: 0.5, im: 0,
      },
      {
        frequency: 1, amplitude: 0.5, phase: 0, re: 0.5, im: 0,
      },
    ],
  },
  {
    input: [
      { amplitude: 2 },
      { amplitude: 0 },
    ],
    output: [
      {
        frequency: 0, amplitude: 1, phase: 0, re: 1, im: 0,
      },
      {
        frequency: 1, amplitude: 1, phase: 0, re: 1, im: 0,
      },
    ],
  },
  {
    input: [
      { amplitude: 1 },
      { amplitude: 0 },
      { amplitude: 0 },
    ],
    output: [
      {
        frequency: 0, amplitude: 0.33333, phase: 0, re: 0.33333, im: 0,
      },
      {
        frequency: 1, amplitude: 0.33333, phase: 0, re: 0.33333, im: 0,
      },
      {
        frequency: 2, amplitude: 0.33333, phase: 0, re: 0.33333, im: 0,
      },
    ],
  },
  {
    input: [
      { amplitude: 1 },
      { amplitude: 0 },
      { amplitude: 0 },
      { amplitude: 0 },
    ],
    output: [
      {
        frequency: 0, amplitude: 0.25, phase: 0, re: 0.25, im: 0,
      },
      {
        frequency: 1, amplitude: 0.25, phase: 0, re: 0.25, im: 0,
      },
      {
        frequency: 2, amplitude: 0.25, phase: 0, re: 0.25, im: 0,
      },
      {
        frequency: 3, amplitude: 0.25, phase: 0, re: 0.25, im: 0,
      },
```

```
    ],
  },
  {
    input: [
      { amplitude: 0 },
      { amplitude: 1 },
      { amplitude: 0 },
      { amplitude: 0 },
    ],
    output: [
      {
        frequency: 0, amplitude: 0.25, phase: 0, re: 0.25, im: 0,
      },
      {
        frequency: 1, amplitude: 0.25, phase: -90, re: 0, im: -0.25,
      },
      {
        frequency: 2, amplitude: 0.25, phase: 180, re: -0.25, im: 0,
      },
      {
        frequency: 3, amplitude: 0.25, phase: 90, re: 0, im: 0.25,
      },
    ],
  },
  {
    input: [
      { amplitude: 0 },
      { amplitude: 0 },
      { amplitude: 1 },
      { amplitude: 0 },
    ],
    output: [
      {
        frequency: 0, amplitude: 0.25, phase: 0, re: 0.25, im: 0,
      },
      {
        frequency: 1, amplitude: 0.25, phase: 180, re: -0.25, im: 0,
      },
      {
        frequency: 2, amplitude: 0.25, phase: 0, re: 0.25, im: 0,
      },
      {
        frequency: 3, amplitude: 0.25, phase: 180, re: -0.25, im: 0,
      },
    ],
  },
  {
    input: [
      { amplitude: 0 },
      { amplitude: 0 },
      { amplitude: 0 },
      { amplitude: 2 },
    ],
    output: [
      {
        frequency: 0, amplitude: 0.5, phase: 0, re: 0.5, im: 0,
      },
      {
        frequency: 1, amplitude: 0.5, phase: 90, re: 0, im: 0.5,
      },
      {
        frequency: 2, amplitude: 0.5, phase: 180, re: -0.5, im: 0,
      },
      {
        frequency: 3, amplitude: 0.5, phase: -90, re: 0, im: -0.5,
      },
    ],
  },
  {
    input: [
      { amplitude: 0 },
      { amplitude: 1 },
      { amplitude: 0 },
      { amplitude: 2 },
    ],
    output: [
      {
        frequency: 0, amplitude: 0.75, phase: 0, re: 0.75, im: 0,
      },
      {
        frequency: 1, amplitude: 0.25, phase: 90, re: 0, im: 0.25,
      },
```

```
      {
        frequency: 2, amplitude: 0.75, phase: 180, re: -0.75, im: 0,
      },
      {
        frequency: 3, amplitude: 0.25, phase: -90, re: 0, im: -0.25,
      },
    ],
  },
  {
    input: [
      { amplitude: 4 },
      { amplitude: 1 },
      { amplitude: 0 },
      { amplitude: 2 },
    ],
    output: [
      {
        frequency: 0, amplitude: 1.75, phase: 0, re: 1.75, im: 0,
      },
      {
        frequency: 1, amplitude: 1.03077, phase: 14.03624, re: 0.99999, im: 0.25,
      },
      {
        frequency: 2, amplitude: 0.25, phase: 0, re: 0.25, im: 0,
      },
      {
        frequency: 3, amplitude: 1.03077, phase: -14.03624, re: 1, im: -0.25,
      },
    ],
  },
  {
    input: [
      { amplitude: 4 },
      { amplitude: 1 },
      { amplitude: -3 },
      { amplitude: 2 },
    ],
    output: [
      {
        frequency: 0, amplitude: 1, phase: 0, re: 1, im: 0,
      },
      {
        frequency: 1, amplitude: 1.76776, phase: 8.13010, re: 1.75, im: 0.25,
      },
      {
        frequency: 2, amplitude: 0.5, phase: 180, re: -0.5, im: 0,
      },
      {
        frequency: 3, amplitude: 1.76776, phase: -8.13010, re: 1.75, im: -0.24999,
      },
    ],
  },
  {
    input: [
      { amplitude: 1 },
      { amplitude: 2 },
      { amplitude: 3 },
      { amplitude: 4 },
    ],
    output: [
      {
        frequency: 0, amplitude: 2.5, phase: 0, re: 2.5, im: 0,
      },
      {
        frequency: 1, amplitude: 0.70710, phase: 135, re: -0.5, im: 0.49999,
      },
      {
        frequency: 2, amplitude: 0.5, phase: 180, re: -0.5, im: 0,
      },
      {
        frequency: 3, amplitude: 0.70710, phase: -134.99999, re: -0.49999, im: -0.5,
      },
    ],
  },
];

export default class FourierTester {
  /**
   * @param {function} fourierTransform
   */
  static testDirectFourierTransform(fourierTransform) {
    fourierTestCases.forEach((testCase) => {
```

```
      const { input, output: expectedOutput } = testCase;

      // Try to split input signal into sequence of pure sinusoids.
      const formattedInput = input.map(sample => sample.amplitude);
      const currentOutput = fourierTransform(formattedInput);

      // Check the signal has been split into proper amount of sub-signals.
      expect(currentOutput.length).toBeGreaterThanOrEqual(formattedInput.length);

      // Now go through all the signals and check their frequency, amplitude and phase.
      expectedOutput.forEach((expectedSignal, frequency) => {
        // Get template data we want to test against.
        const currentSignal = currentOutput[frequency];
        const currentPolarSignal = currentSignal.getPolarForm(false);

        // Check all signal parameters.
        expect(frequency).toBe(expectedSignal.frequency);
        expect(currentSignal.re).toBeCloseTo(expectedSignal.re, 4);
        expect(currentSignal.im).toBeCloseTo(expectedSignal.im, 4);
        expect(currentPolarSignal.phase).toBeCloseTo(expectedSignal.phase, 4);
        expect(currentPolarSignal.radius).toBeCloseTo(expectedSignal.amplitude, 4);
      });
    });
  }

  /**
   * @param {function} inverseFourierTransform
   */
  static testInverseFourierTransform(inverseFourierTransform) {
    fourierTestCases.forEach((testCase) => {
      const { input: expectedOutput, output: inputFrequencies } = testCase;

      // Try to join frequencies into time signal.
      const formattedInput = inputFrequencies.map((frequency) => {
        return new ComplexNumber({ re: frequency.re, im: frequency.im });
      });
      const currentOutput = inverseFourierTransform(formattedInput);

      // Check the signal has been combined of proper amount of time samples.
      expect(currentOutput.length).toBeLessThanOrEqual(formattedInput.length);

      // Now go through all the amplitudes and check their values.
      expectedOutput.forEach((expectedAmplitudes, timer) => {
        // Get template data we want to test against.
        const currentAmplitude = currentOutput[timer];

        // Check if current amplitude is close enough to the calculated one.
        expect(currentAmplitude).toBeCloseTo(expectedAmplitudes.amplitude, 4);
      });
    });
  }
}
```

Listing 97: inverseDiscreteFourierTransform.test.js

```
import inverseDiscreteFourierTransform from '../inverseDiscreteFourierTransform';
import FourierTester from './FourierTester';

describe('inverseDiscreteFourierTransform', () => {
  it('should calculate output signal out of input frequencies', () => {
    FourierTester.testInverseFourierTransform(inverseDiscreteFourierTransform);
  });
});
```

Listing 98: discreteFourierTransform.js

```
import ComplexNumber from '../complex-number/ComplexNumber';

const CLOSE_TO_ZERO_THRESHOLD = 1e-10;

/**
 * Discrete Fourier Transform (DFT): time to frequencies.
 *
 * Time complexity: O(N^2)
 *
 * @param {number[]} inputAmplitudes - Input signal amplitudes over time (complex
 * numbers with real parts only).
 * @param {number} zeroThreshold - Threshold that is used to convert real and imaginary numbers
 * to zero in case if they are smaller then this.
 *
 * @return {ComplexNumber[]} - Array of complex number. Each of the number represents the frequency
 * or signal. All signals together will form input signal over discrete time periods. Each signal's
 * complex number has radius (amplitude) and phase (angle) in polar form that describes the signal.
 *
 * @see https://gist.github.com/anonymous/129d477ddb1c8025c9ac
 * @see https://betterexplained.com/articles/an-interactive-guide-to-the-fourier-transform/
 */
export default function dft(inputAmplitudes, zeroThreshold = CLOSE_TO_ZERO_THRESHOLD) {
  const N = inputAmplitudes.length;
  const signals = [];

  // Go through every discrete frequency.
  for (let frequency = 0; frequency < N; frequency += 1) {
    // Compound signal at current frequency that will ultimately
    // take part in forming input amplitudes.
    let frequencySignal = new ComplexNumber();

    // Go through every discrete point in time.
    for (let timer = 0; timer < N; timer += 1) {
      const currentAmplitude = inputAmplitudes[timer];

      // Calculate rotation angle.
      const rotationAngle = -1 * (2 * Math.PI) * frequency * (timer / N);

      // Remember that e^ix = cos(x) + i * sin(x);
      const dataPointContribution = new ComplexNumber({
        re: Math.cos(rotationAngle),
        im: Math.sin(rotationAngle),
      }).multiply(currentAmplitude);

      // Add this data point's contribution.
      frequencySignal = frequencySignal.add(dataPointContribution);
    }

    // Close to zero? You're zero.
    if (Math.abs(frequencySignal.re) < zeroThreshold) {
      frequencySignal.re = 0;
    }

    if (Math.abs(frequencySignal.im) < zeroThreshold) {
      frequencySignal.im = 0;
    }

    // Average contribution at this frequency.
    // The 1/N factor is usually moved to the reverse transform (going from frequencies
    // back to time). This is allowed, though it would be nice to have 1/N in the forward
    // transform since it gives the actual sizes for the time spikes.
    frequencySignal = frequencySignal.divide(N);

    // Add current frequency signal to the list of compound signals.
    signals[frequency] = frequencySignal;
  }

  return signals;
}
```

Listing 99: fastFourierTransform.js

```
import ComplexNumber from '../complex-number/ComplexNumber';
import bitLength from '../bits/bitLength';

/**
 * Returns the number which is the flipped binary representation of input.
 *
 * @param {number} input
 * @param {number} bitsCount
 * @return {number}
 */
function reverseBits(input, bitsCount) {
  let reversedBits = 0;

  for (let bitIndex = 0; bitIndex < bitsCount; bitIndex += 1) {
    reversedBits *= 2;

    if (Math.floor(input / (1 << bitIndex)) % 2 === 1) {
      reversedBits += 1;
    }
  }

  return reversedBits;
}

/**
 * Returns the radix-2 fast fourier transform of the given array.
 * Optionally computes the radix-2 inverse fast fourier transform.
 *
 * @param {ComplexNumber[]} inputData
 * @param {boolean} [inverse]
 * @return {ComplexNumber[]}
 */
export default function fastFourierTransform(inputData, inverse = false) {
  const bitsCount = bitLength(inputData.length - 1);
  const N = 1 << bitsCount;

  while (inputData.length < N) {
    inputData.push(new ComplexNumber());
  }

  const output = [];
  for (let dataSampleIndex = 0; dataSampleIndex < N; dataSampleIndex += 1) {
    output[dataSampleIndex] = inputData[reverseBits(dataSampleIndex, bitsCount)];
  }

  for (let blockLength = 2; blockLength <= N; blockLength *= 2) {
    const imaginarySign = inverse ? -1 : 1;
    const phaseStep = new ComplexNumber({
      re: Math.cos(2 * Math.PI / blockLength),
      im: imaginarySign * Math.sin(2 * Math.PI / blockLength),
    });

    for (let blockStart = 0; blockStart < N; blockStart += blockLength) {
      let phase = new ComplexNumber({ re: 1, im: 0 });

      for (let signalId = blockStart; signalId < (blockStart + blockLength / 2); signalId += 1) {
        const component = output[signalId + blockLength / 2].multiply(phase);

        const upd1 = output[signalId].add(component);
        const upd2 = output[signalId].subtract(component);

        output[signalId] = upd1;
        output[signalId + blockLength / 2] = upd2;

        phase = phase.multiply(phaseStep);
      }
    }
  }

  if (inverse) {
    for (let signalId = 0; signalId < N; signalId += 1) {
      output[signalId] /= N;
    }
  }

  return output;
}
```

Listing 100: inverseDiscreteFourierTransform.js

```
import ComplexNumber from '../complex-number/ComplexNumber';

const CLOSE_TO_ZERO_THRESHOLD = 1e-10;

/**
 * Inverse Discrete Fourier Transform (IDFT): frequencies to time.
 *
 * Time complexity: O(N^2)
 *
 * @param {ComplexNumber[]} frequencies - Frequencies summands of the final signal.
 * @param {number} zeroThreshold - Threshold that is used to convert real and imaginary numbers
 * to zero in case if they are smaller then this.
 *
 * @return {number[]} - Discrete amplitudes distributed in time.
 */
export default function inverseDiscreteFourierTransform(
  frequencies,
  zeroThreshold = CLOSE_TO_ZERO_THRESHOLD,
) {
  const N = frequencies.length;
  const amplitudes = [];

  // Go through every discrete point of time.
  for (let timer = 0; timer < N; timer += 1) {
    // Compound amplitude at current time.
    let amplitude = new ComplexNumber();

    // Go through all discrete frequencies.
    for (let frequency = 0; frequency < N; frequency += 1) {
      const currentFrequency = frequencies[frequency];

      // Calculate rotation angle.
      const rotationAngle = (2 * Math.PI) * frequency * (timer / N);

      // Remember that e^ix = cos(x) + i * sin(x);
      const frequencyContribution = new ComplexNumber({
        re: Math.cos(rotationAngle),
        im: Math.sin(rotationAngle),
      }).multiply(currentFrequency);

      amplitude = amplitude.add(frequencyContribution);
    }

    // Close to zero? You're zero.
    if (Math.abs(amplitude.re) < zeroThreshold) {
      amplitude.re = 0;
    }

    if (Math.abs(amplitude.im) < zeroThreshold) {
      amplitude.im = 0;
    }

    // Add current frequency signal to the list of compound signals.
    amplitudes[timer] = amplitude.re;
  }

  return amplitudes;
}
```

Listing 101: integerPartition.test.js

```
import integerPartition from '../integerPartition';

describe('integerPartition', () => {
  it('should partition the number', () => {
    expect(integerPartition(1)).toBe(1);
    expect(integerPartition(2)).toBe(2);
    expect(integerPartition(3)).toBe(3);
    expect(integerPartition(4)).toBe(5);
    expect(integerPartition(5)).toBe(7);
    expect(integerPartition(6)).toBe(11);
    expect(integerPartition(7)).toBe(15);
    expect(integerPartition(8)).toBe(22);
  });
});
```

Listing 102: integerPartition.js

```
/**
* @param {number} number
* @return {number}
*/
export default function integerPartition(number) {
  // Create partition matrix for solving this task using Dynamic Programming.
  const partitionMatrix = Array(number + 1).fill(null).map(() => {
    return Array(number + 1).fill(null);
  });

  // Fill partition matrix with initial values.

  // Let's fill the first row that represents how many ways we would have
  // to combine the numbers 1, 2, 3, ..., n with number 0. We would have zero
  // ways obviously since with zero number we may form only zero.
  for (let numberIndex = 1; numberIndex <= number; numberIndex += 1) {
    partitionMatrix[0][numberIndex] = 0;
  }

  // Let's fill the first column. It represents the number of ways we can form
  // number zero out of numbers 0, 0 and 1, 0 and 1 and 2, 0 and 1 and 2 and 3, ...
  // Obviously there is only one way we could form number 0
  // and it is with number 0 itself.
  for (let summandIndex = 0; summandIndex <= number; summandIndex += 1) {
    partitionMatrix[summandIndex][0] = 1;
  }

  // Now let's go through other possible options of how we could form number m out of
  // summands 0, 1, ..., m using Dynamic Programming approach.
  for (let summandIndex = 1; summandIndex <= number; summandIndex += 1) {
    for (let numberIndex = 1; numberIndex <= number; numberIndex += 1) {
      if (summandIndex > numberIndex) {
        // If summand number is bigger then current number itself then just it won't add
        // any new ways of forming the number. Thus we may just copy the number from row above.
        partitionMatrix[summandIndex][numberIndex] = partitionMatrix[summandIndex - 1][numberIndex];
      } else {
        /*
         * The number of combinations would equal to number of combinations of forming the same
         * number but WITHOUT current summand number PLUS number of combinations of forming the
         * <current number - current summand> number but WITH current summand.
         *
         * Example:
         * Number of ways to form 5 using summands {0, 1, 2} would equal the SUM of:
         * - number of ways to form 5 using summands {0, 1} (we've excluded summand 2)
         * - number of ways to form 3 (because 5 - 2 = 3) using summands {0, 1, 2}
         * (we've included summand 2)
        */
        const combosWithoutSummand = partitionMatrix[summandIndex - 1][numberIndex];
        const combosWithSummand = partitionMatrix[summandIndex][numberIndex - summandIndex];

        partitionMatrix[summandIndex][numberIndex] = combosWithoutSummand + combosWithSummand;
      }
    }
  }

  return partitionMatrix[number][number];
}
```

Listing 103: isPowerOfTwo.test.js

```
import isPowerOfTwo from '../isPowerOfTwo';

describe('isPowerOfTwo', () => {
  it('should check if the number is made by multiplying twos', () => {
    expect(isPowerOfTwo(-1)).toBe(false);
    expect(isPowerOfTwo(0)).toBe(false);
    expect(isPowerOfTwo(1)).toBe(true);
    expect(isPowerOfTwo(2)).toBe(true);
    expect(isPowerOfTwo(3)).toBe(false);
    expect(isPowerOfTwo(4)).toBe(true);
    expect(isPowerOfTwo(5)).toBe(false);
    expect(isPowerOfTwo(6)).toBe(false);
    expect(isPowerOfTwo(7)).toBe(false);
    expect(isPowerOfTwo(8)).toBe(true);
    expect(isPowerOfTwo(10)).toBe(false);
    expect(isPowerOfTwo(12)).toBe(false);
    expect(isPowerOfTwo(16)).toBe(true);
    expect(isPowerOfTwo(31)).toBe(false);
    expect(isPowerOfTwo(64)).toBe(true);
    expect(isPowerOfTwo(1024)).toBe(true);
    expect(isPowerOfTwo(1023)).toBe(false);
  });
});
```

Listing 104: isPowerOfTwoBitwise.test.js

```
import isPowerOfTwoBitwise from '../isPowerOfTwoBitwise';

describe('isPowerOfTwoBitwise', () => {
  it('should check if the number is made by multiplying twos', () => {
    expect(isPowerOfTwoBitwise(-1)).toBe(false);
    expect(isPowerOfTwoBitwise(0)).toBe(false);
    expect(isPowerOfTwoBitwise(1)).toBe(true);
    expect(isPowerOfTwoBitwise(2)).toBe(true);
    expect(isPowerOfTwoBitwise(3)).toBe(false);
    expect(isPowerOfTwoBitwise(4)).toBe(true);
    expect(isPowerOfTwoBitwise(5)).toBe(false);
    expect(isPowerOfTwoBitwise(6)).toBe(false);
    expect(isPowerOfTwoBitwise(7)).toBe(false);
    expect(isPowerOfTwoBitwise(8)).toBe(true);
    expect(isPowerOfTwoBitwise(10)).toBe(false);
    expect(isPowerOfTwoBitwise(12)).toBe(false);
    expect(isPowerOfTwoBitwise(16)).toBe(true);
    expect(isPowerOfTwoBitwise(31)).toBe(false);
    expect(isPowerOfTwoBitwise(64)).toBe(true);
    expect(isPowerOfTwoBitwise(1024)).toBe(true);
    expect(isPowerOfTwoBitwise(1023)).toBe(false);
  });
});
```

Listing 105: isPowerOfTwo.js

```
/**
 * @param {number} number
 * @return {boolean}
 */
export default function isPowerOfTwo(number) {
  // 1 (2^0) is the smallest power of two.
  if (number < 1) {
    return false;
  }

  // Let's find out if we can divide the number by two
  // many times without remainder.
  let dividedNumber = number;
  while (dividedNumber !== 1) {
    if (dividedNumber % 2 !== 0) {
      // For every case when remainder isn't zero we can say that this number
      // couldn't be a result of power of two.
      return false;
    }

    dividedNumber /= 2;
  }

  return true;
}
```

Listing 106: isPowerOfTwoBitwise.js

```
/**
 * @param {number} number
 * @return {boolean}
 */
export default function isPowerOfTwoBitwise(number) {
  // 1 (2^0) is the smallest power of two.
  if (number < 1) {
    return false;
  }

  /*
   * Powers of two in binary look like this:
   * 1: 0001
   * 2: 0010
   * 4: 0100
   * 8: 1000
   *
   * Note that there is always exactly 1 bit set. The only exception is with a signed integer.
   * e.g. An 8-bit signed integer with a value of -128 looks like:
   * 10000000
   *
   * So after checking that the number is greater than zero, we can use a clever little bit
   * hack to test that one and only one bit is set.
   */
  return (number & (number - 1)) === 0;
}
```

Listing 107: leastCommonMultiple.test.js

```
import leastCommonMultiple from '../leastCommonMultiple';

describe('leastCommonMultiple', () => {
  it('should find least common multiple', () => {
    expect(leastCommonMultiple(0, 0)).toBe(0);
    expect(leastCommonMultiple(1, 0)).toBe(0);
    expect(leastCommonMultiple(0, 1)).toBe(0);
    expect(leastCommonMultiple(4, 6)).toBe(12);
    expect(leastCommonMultiple(6, 21)).toBe(42);
    expect(leastCommonMultiple(7, 2)).toBe(14);
    expect(leastCommonMultiple(3, 5)).toBe(15);
    expect(leastCommonMultiple(7, 3)).toBe(21);
    expect(leastCommonMultiple(1000000, 2)).toBe(1000000);
    expect(leastCommonMultiple(-9, -18)).toBe(18);
    expect(leastCommonMultiple(-7, -9)).toBe(63);
    expect(leastCommonMultiple(-7, 9)).toBe(63);
  });
});
```

Listing 108: leastCommonMultiple.js

```
import euclideanAlgorithm from '../euclidean-algorithm/euclideanAlgorithm';

/**
 * @param {number} a
 * @param {number} b
 * @return {number}
 */

export default function leastCommonMultiple(a, b) {
  return ((a === 0) || (b === 0)) ? 0 : Math.abs(a * b) / euclideanAlgorithm(a, b);
}
```

Listing 109: liuHui.test.js

```
 import liuHui from '../liuHui';

describe('liuHui', () => {
  it('should calculate  based on 12-gon', () => {
    expect(liuHui(1)).toBe(3);
  });

  it('should calculate  based on 24-gon', () => {
    expect(liuHui(2)).toBe(3.105828541230249);
  });

  it('should calculate  based on 6144-gon', () => {
    expect(liuHui(10)).toBe(3.1415921059992717);
  });

  it('should calculate  based on 201326592-gon', () => {
    expect(liuHui(25)).toBe(3.141592653589793);
  });
});
```

Listing 110: liuHui.js

```
/*
 * Let circleRadius is the radius of circle.
 * circleRadius is also the side length of the inscribed hexagon
 */
const circleRadius = 1;

/**
 * @param {number} sideLength
 * @param {number} splitCounter
 * @return {number}
 */
function getNGonSideLength(sideLength, splitCounter) {
  if (splitCounter <= 0) {
    return sideLength;
  }

  const halfSide = sideLength / 2;

  // Liu Hui used the Gou Gu (Pythagorean theorem) theorem repetitively.
  const perpendicular = Math.sqrt((circleRadius ** 2) - (halfSide ** 2));
  const excessRadius = circleRadius - perpendicular;
  const splitSideLength = Math.sqrt((excessRadius ** 2) + (halfSide ** 2));

  return getNGonSideLength(splitSideLength, splitCounter - 1);
}

/**
 * @param {number} splitCount
 * @return {number}
 */
function getNGonSideCount(splitCount) {
  // Liu Hui began with an inscribed hexagon (6-gon).
  const hexagonSidesCount = 6;

  // On every split iteration we make N-gons: 6-gon, 12-gon, 24-gon, 48-gon and so on.
  return hexagonSidesCount * (splitCount ? 2 ** splitCount : 1);
}

/**
 * Calculate the  value using Liu Hui's  algorithm
 *
 * @param {number} splitCount - number of times we're going to split 6-gon.
 *  On each split we will receive 12-gon, 24-gon and so on.
 * @return {number}
 */
export default function liuHui(splitCount = 1) {
  const nGonSideLength = getNGonSideLength(circleRadius, splitCount - 1);
  const nGonSideCount = getNGonSideCount(splitCount - 1);
  const nGonPerimeter = nGonSideLength * nGonSideCount;
  const approximateCircleArea = (nGonPerimeter / 2) * circleRadius;

  // Return approximate value of pi.
  return approximateCircleArea / (circleRadius ** 2);
}
```

Listing 111: pascalTriangle.test.js

```
import pascalTriangle from '../pascalTriangle';

describe('pascalTriangle', () => {
  it('should calculate Pascal Triangle coefficients for specific line number', () => {
    expect(pascalTriangle(0)).toEqual([1]);
    expect(pascalTriangle(1)).toEqual([1, 1]);
    expect(pascalTriangle(2)).toEqual([1, 2, 1]);
    expect(pascalTriangle(3)).toEqual([1, 3, 3, 1]);
    expect(pascalTriangle(4)).toEqual([1, 4, 6, 4, 1]);
    expect(pascalTriangle(5)).toEqual([1, 5, 10, 10, 5, 1]);
    expect(pascalTriangle(6)).toEqual([1, 6, 15, 20, 15, 6, 1]);
    expect(pascalTriangle(7)).toEqual([1, 7, 21, 35, 35, 21, 7, 1]);
  });
});
```

Listing 112: pascalTriangleRecursive.test.js

```
import pascalTriangleRecursive from '../pascalTriangleRecursive';

describe('pascalTriangleRecursive', () => {
  it('should calculate Pascal Triangle coefficients for specific line number', () => {
    expect(pascalTriangleRecursive(0)).toEqual([1]);
    expect(pascalTriangleRecursive(1)).toEqual([1, 1]);
    expect(pascalTriangleRecursive(2)).toEqual([1, 2, 1]);
    expect(pascalTriangleRecursive(3)).toEqual([1, 3, 3, 1]);
    expect(pascalTriangleRecursive(4)).toEqual([1, 4, 6, 4, 1]);
    expect(pascalTriangleRecursive(5)).toEqual([1, 5, 10, 10, 5, 1]);
    expect(pascalTriangleRecursive(6)).toEqual([1, 6, 15, 20, 15, 6, 1]);
    expect(pascalTriangleRecursive(7)).toEqual([1, 7, 21, 35, 35, 21, 7, 1]);
  });
});
```

Listing 113: pascalTriangle.js

```
/**
 * @param {number} lineNumber - zero based.
 * @return {number[]}
 */
export default function pascalTriangle(lineNumber) {
  const currentLine = [1];

  const currentLineSize = lineNumber + 1;

  for (let numIndex = 1; numIndex < currentLineSize; numIndex += 1) {
    // See explanation of this formula in README.
    currentLine[numIndex] = currentLine[numIndex - 1] * (lineNumber - numIndex + 1) / numIndex;
  }

  return currentLine;
}
```

Listing 114: pascalTriangleRecursive.js

```
/**
 * @param {number} lineNumber - zero based.
 * @return {number[]}
 */
export default function pascalTriangleRecursive(lineNumber) {
  if (lineNumber === 0) {
    return [1];
  }

  const currentLineSize = lineNumber + 1;
  const previousLineSize = currentLineSize - 1;

  // Create container for current line values.
  const currentLine = [];

  // We'll calculate current line based on previous one.
  const previousLine = pascalTriangleRecursive(lineNumber - 1);

  // Let's go through all elements of current line except the first and
  // last one (since they were and will be filled with 1's) and calculate
  // current coefficient based on previous line.
  for (let numIndex = 0; numIndex < currentLineSize; numIndex += 1) {
    const leftCoefficient = (numIndex - 1) >= 0 ? previousLine[numIndex - 1] : 0;
    const rightCoefficient = numIndex < previousLineSize ? previousLine[numIndex] : 0;

    currentLine[numIndex] = leftCoefficient + rightCoefficient;
  }

  return currentLine;
}
```

Listing 115: trialDivision.test.js

```
 import trialDivision from '../trialDivision';

/**
 * @param {function(n: number)} testFunction
 */
function primalityTest(testFunction) {
  expect(testFunction(1)).toBe(false);
  expect(testFunction(2)).toBe(true);
  expect(testFunction(3)).toBe(true);
  expect(testFunction(5)).toBe(true);
  expect(testFunction(11)).toBe(true);
  expect(testFunction(191)).toBe(true);
  expect(testFunction(191)).toBe(true);
  expect(testFunction(199)).toBe(true);

  expect(testFunction(-1)).toBe(false);
  expect(testFunction(0)).toBe(false);
  expect(testFunction(4)).toBe(false);
  expect(testFunction(6)).toBe(false);
  expect(testFunction(12)).toBe(false);
  expect(testFunction(14)).toBe(false);
  expect(testFunction(25)).toBe(false);
  expect(testFunction(192)).toBe(false);
  expect(testFunction(200)).toBe(false);
  expect(testFunction(400)).toBe(false);

  // It should also deal with floats.
  expect(testFunction(0.5)).toBe(false);
  expect(testFunction(1.3)).toBe(false);
  expect(testFunction(10.5)).toBe(false);
}

describe('trialDivision', () => {
  it('should detect prime numbers', () => {
    primalityTest(trialDivision);
  });
});
```

Listing 116: trialDivision.js

```
/**
 * @param {number} number
 * @return {boolean}
 */
export default function trialDivision(number) {
  // Check if number is integer.
  if (number % 1 !== 0) {
    return false;
  }

  if (number <= 1) {
    // If number is less than one then it isn't prime by definition.
    return false;
  }

  if (number <= 3) {
    // All numbers from 2 to 3 are prime.
    return true;
  }

  // If the number is not divided by 2 then we may eliminate all further even dividers.
  if (number % 2 === 0) {
    return false;
  }

  // If there is no dividers up to square root of n then there is no higher dividers as well.
  const dividerLimit = Math.sqrt(number);
  for (let divider = 3; divider <= dividerLimit; divider += 2) {
    if (number % divider === 0) {
      return false;
    }
  }

  return true;
}
```

Listing 117: degreeToRadian.test.js

```
import degreeToRadian from '../degreeToRadian';

describe('degreeToRadian', () => {
  it('should convert degree to radian', () => {
    expect(degreeToRadian(0)).toBe(0);
    expect(degreeToRadian(45)).toBe(Math.PI / 4);
    expect(degreeToRadian(90)).toBe(Math.PI / 2);
    expect(degreeToRadian(180)).toBe(Math.PI);
    expect(degreeToRadian(270)).toBe(3 * Math.PI / 2);
    expect(degreeToRadian(360)).toBe(2 * Math.PI);
  });
});
```

Listing 118: radianToDegree.test.js

```
import radianToDegree from '../radianToDegree';

describe('radianToDegree', () => {
  it('should convert radian to degree', () => {
    expect(radianToDegree(0)).toBe(0);
    expect(radianToDegree(Math.PI / 4)).toBe(45);
    expect(radianToDegree(Math.PI / 2)).toBe(90);
    expect(radianToDegree(Math.PI)).toBe(180);
    expect(radianToDegree(3 * Math.PI / 2)).toBe(270);
    expect(radianToDegree(2 * Math.PI)).toBe(360);
  });
});
```

Listing 119: degreeToRadian.js

```
/**
 * @param {number} degree
 * @return {number}
 */
export default function degreeToRadian(degree) {
  return degree * (Math.PI / 180);
}
```

Listing 120: radianToDegree.js

```
/**
 * @param {number} radian
 * @return {number}
 */
export default function radianToDegree(radian) {
  return radian * (180 / Math.PI);
}
```

Listing 121: sieveOfEratosthenes.test.js

```
import sieveOfEratosthenes from '../sieveOfEratosthenes';

describe('sieveOfEratosthenes', () => {
  it('should find all primes less than or equal to n', () => {
    expect(sieveOfEratosthenes(5)).toEqual([2, 3, 5]);
    expect(sieveOfEratosthenes(10)).toEqual([2, 3, 5, 7]);
    expect(sieveOfEratosthenes(100)).toEqual([
      2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
      43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97,
    ]);
  });
});
```

Listing 122: sieveOfEratosthenes.js

```
/**
 * @param {number} maxNumber
 * @return {number[]}
 */
export default function sieveOfEratosthenes(maxNumber) {
  const isPrime = new Array(maxNumber + 1).fill(true);
  isPrime[0] = false;
  isPrime[1] = false;

  const primes = [];

  for (let number = 2; number <= maxNumber; number += 1) {
    if (isPrime[number] === true) {
      primes.push(number);

      /*
       * Optimisation.
       * Start marking multiples of `p` from `p * p`, and not from `2 * p`.
       * The reason why this works is because, at that point, smaller multiples
       * of `p` will have already been marked `false`.
       *
       * Warning: When working with really big numbers, the following line may cause overflow
       * In that case, it can be changed to:
       * let nextNumber = 2 * number;
       */
      let nextNumber = number * number;

      while (nextNumber <= maxNumber) {
        isPrime[nextNumber] = false;
        nextNumber += number;
      }
    }
  }

  return primes;
}
```

Listing 123: squareRoot.test.js

```
 import squareRoot from '../squareRoot';

describe('squareRoot', () => {
  it('should throw for negative numbers', () => {
    function failingSquareRoot() {
      squareRoot(-5);
    }
    expect(failingSquareRoot).toThrow();
  });

  it('should correctly calculate square root with default tolerance', () => {
    expect(squareRoot(0)).toBe(0);
    expect(squareRoot(1)).toBe(1);
    expect(squareRoot(2)).toBe(1);
    expect(squareRoot(3)).toBe(2);
    expect(squareRoot(4)).toBe(2);
    expect(squareRoot(15)).toBe(4);
    expect(squareRoot(16)).toBe(4);
    expect(squareRoot(256)).toBe(16);
    expect(squareRoot(473)).toBe(22);
    expect(squareRoot(14723)).toBe(121);
  });

  it('should correctly calculate square root for integers with custom tolerance', () => {
    let tolerance = 1;

    expect(squareRoot(0, tolerance)).toBe(0);
    expect(squareRoot(1, tolerance)).toBe(1);
    expect(squareRoot(2, tolerance)).toBe(1.4);
    expect(squareRoot(3, tolerance)).toBe(1.8);
    expect(squareRoot(4, tolerance)).toBe(2);
    expect(squareRoot(15, tolerance)).toBe(3.9);
    expect(squareRoot(16, tolerance)).toBe(4);
    expect(squareRoot(256, tolerance)).toBe(16);
    expect(squareRoot(473, tolerance)).toBe(21.7);
    expect(squareRoot(14723, tolerance)).toBe(121.3);

    tolerance = 3;

    expect(squareRoot(0, tolerance)).toBe(0);
    expect(squareRoot(1, tolerance)).toBe(1);
    expect(squareRoot(2, tolerance)).toBe(1.414);
    expect(squareRoot(3, tolerance)).toBe(1.732);
    expect(squareRoot(4, tolerance)).toBe(2);
    expect(squareRoot(15, tolerance)).toBe(3.873);
    expect(squareRoot(16, tolerance)).toBe(4);
    expect(squareRoot(256, tolerance)).toBe(16);
    expect(squareRoot(473, tolerance)).toBe(21.749);
    expect(squareRoot(14723, tolerance)).toBe(121.338);

    tolerance = 10;

    expect(squareRoot(0, tolerance)).toBe(0);
    expect(squareRoot(1, tolerance)).toBe(1);
    expect(squareRoot(2, tolerance)).toBe(1.4142135624);
    expect(squareRoot(3, tolerance)).toBe(1.7320508076);
    expect(squareRoot(4, tolerance)).toBe(2);
    expect(squareRoot(15, tolerance)).toBe(3.8729833462);
    expect(squareRoot(16, tolerance)).toBe(4);
    expect(squareRoot(256, tolerance)).toBe(16);
    expect(squareRoot(473, tolerance)).toBe(21.7485631709);
    expect(squareRoot(14723, tolerance)).toBe(121.3383698588);
  });

  it('should correctly calculate square root for integers with custom tolerance', () => {
    expect(squareRoot(4.5, 10)).toBe(2.1213203436);
    expect(squareRoot(217.534, 10)).toBe(14.7490338667);
  });
});
```

Listing 124: squareRoot.js

```
/**
 * Calculates the square root of the number with given tolerance (precision)
 * by using Newton's method.
 *
 * @param number - the number we want to find a square root for.
 * @param [tolerance] - how many precise numbers after the floating point we want to get.
 * @return {number}
 */
export default function squareRoot(number, tolerance = 0) {
  // For now we won't support operations that involves manipulation with complex numbers.
  if (number < 0) {
    throw new Error('The method supports only positive integers');
  }

  // Handle edge case with finding the square root of zero.
  if (number === 0) {
    return 0;
  }

  // We will start approximation from value 1.
  let root = 1;

  // Delta is a desired distance between the number and the square of the root.
  // - if tolerance=0 then delta=1
  // - if tolerance=1 then delta=0.1
  // - if tolerance=2 then delta=0.01
  // - and so on...
  const requiredDelta = 1 / (10 ** tolerance);

  // Approximating the root value to the point when we get a desired precision.
  while (Math.abs(number - (root ** 2)) > requiredDelta) {
    // Newton's method reduces in this case to the so-called Babylonian method.
    // These methods generally yield approximate results, but can be made arbitrarily
    // precise by increasing the number of calculation steps.
    root -= ((root ** 2) - number) / (2 * root);
  }

  // Cut off undesired floating digits and return the root value.
  return Math.round(root * (10 ** tolerance)) / (10 ** tolerance);
}
```

Listing 125: binarySearch.test.js

```
import binarySearch from '../binarySearch';

describe('binarySearch', () => {
  it('should search number in sorted array', () => {
    expect(binarySearch([], 1)).toBe(-1);
    expect(binarySearch([1], 1)).toBe(0);
    expect(binarySearch([1, 2], 1)).toBe(0);
    expect(binarySearch([1, 2], 2)).toBe(1);
    expect(binarySearch([1, 5, 10, 12], 1)).toBe(0);
    expect(binarySearch([1, 5, 10, 12, 14, 17, 22, 100], 17)).toBe(5);
    expect(binarySearch([1, 5, 10, 12, 14, 17, 22, 100], 1)).toBe(0);
    expect(binarySearch([1, 5, 10, 12, 14, 17, 22, 100], 100)).toBe(7);
    expect(binarySearch([1, 5, 10, 12, 14, 17, 22, 100], 0)).toBe(-1);
  });

  it('should search object in sorted array', () => {
    const sortedArrayOfObjects = [
      { key: 1, value: 'value1' },
      { key: 2, value: 'value2' },
      { key: 3, value: 'value3' },
    ];

    const comparator = (a, b) => {
      if (a.key === b.key) return 0;
      return a.key < b.key ? -1 : 1;
    };

    expect(binarySearch([], { key: 1 }, comparator)).toBe(-1);
    expect(binarySearch(sortedArrayOfObjects, { key: 4 }, comparator)).toBe(-1);
    expect(binarySearch(sortedArrayOfObjects, { key: 1 }, comparator)).toBe(0);
    expect(binarySearch(sortedArrayOfObjects, { key: 2 }, comparator)).toBe(1);
    expect(binarySearch(sortedArrayOfObjects, { key: 3 }, comparator)).toBe(2);
  });
});
```

Listing 126: binarySearch.js

```
import Comparator from '../../../utils/comparator/Comparator';

/**
 * Binary search implementation.
 *
 * @param {*[]} sortedArray
 * @param {*} seekElement
 * @param {function(a, b)} [comparatorCallback]
 * @return {number}
 */

export default function binarySearch(sortedArray, seekElement, comparatorCallback) {
  // Let's create comparator from the comparatorCallback function.
  // Comparator object will give us common comparison methods like equal() and lessThen().
  const comparator = new Comparator(comparatorCallback);

  // These two indices will contain current array (sub-array) boundaries.
  let startIndex = 0;
  let endIndex = sortedArray.length - 1;

  // Let's continue to split array until boundaries are collapsed
  // and there is nothing to split anymore.
  while (startIndex <= endIndex) {
    // Let's calculate the index of the middle element.
    const middleIndex = startIndex + Math.floor((endIndex - startIndex) / 2);

    // If we've found the element just return its position.
    if (comparator.equal(sortedArray[middleIndex], seekElement)) {
      return middleIndex;
    }

    // Decide which half to choose for seeking next: left or right one.
    if (comparator.lessThan(sortedArray[middleIndex], seekElement)) {
      // Go to the right half of the array.
      startIndex = middleIndex + 1;
    } else {
      // Go to the left half of the array.
      endIndex = middleIndex - 1;
    }
  }

  // Return -1 if we have not found anything.
  return -1;
}
```

Listing 127: interpolationSearch.test.js

```
import interpolationSearch from '../interpolationSearch';

describe('interpolationSearch', () => {
  it('should search elements in sorted array of numbers', () => {
    expect(interpolationSearch([], 1)).toBe(-1);
    expect(interpolationSearch([1], 1)).toBe(0);
    expect(interpolationSearch([1], 0)).toBe(-1);
    expect(interpolationSearch([1, 1], 1)).toBe(0);
    expect(interpolationSearch([1, 2], 1)).toBe(0);
    expect(interpolationSearch([1, 2], 2)).toBe(1);
    expect(interpolationSearch([10, 20, 30, 40, 50], 40)).toBe(3);
    expect(interpolationSearch([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15], 14)).toBe(13);
    expect(interpolationSearch([1, 6, 7, 8, 12, 13, 14, 19, 21, 23, 24, 24, 24, 300], 24)).toBe(10);
    expect(interpolationSearch([1, 2, 3, 700, 800, 1200, 1300, 1400, 1900], 600)).toBe(-1);
    expect(interpolationSearch([1, 2, 3, 700, 800, 1200, 1300, 1400, 1900], 1)).toBe(0);
    expect(interpolationSearch([1, 2, 3, 700, 800, 1200, 1300, 1400, 1900], 2)).toBe(1);
    expect(interpolationSearch([1, 2, 3, 700, 800, 1200, 1300, 1400, 1900], 3)).toBe(2);
    expect(interpolationSearch([1, 2, 3, 700, 800, 1200, 1300, 1400, 1900], 700)).toBe(3);
    expect(interpolationSearch([1, 2, 3, 700, 800, 1200, 1300, 1400, 1900], 800)).toBe(4);
    expect(interpolationSearch([0, 2, 3, 700, 800, 1200, 1300, 1400, 1900], 1200)).toBe(5);
    expect(interpolationSearch([1, 2, 3, 700, 800, 1200, 1300, 1400, 19000], 800)).toBe(4);
    expect(interpolationSearch([0, 10, 11, 12, 13, 14, 15], 10)).toBe(1);
  });
});
```

Listing 128: interpolationSearch.js

```
/**
 * Interpolation search implementation.
 *
 * @param {*[]} sortedArray - sorted array with uniformly distributed values
 * @param {*} seekElement
 * @return {number}
 */
export default function interpolationSearch(sortedArray, seekElement) {
  let leftIndex = 0;
  let rightIndex = sortedArray.length - 1;

  while (leftIndex <= rightIndex) {
    const rangeDelta = sortedArray[rightIndex] - sortedArray[leftIndex];
    const indexDelta = rightIndex - leftIndex;
    const valueDelta = seekElement - sortedArray[leftIndex];

    // If valueDelta is less then zero it means that there is no seek element
    // exists in array since the lowest element from the range is already higher
    // then seek element.
    if (valueDelta < 0) {
      return -1;
    }

    // If range delta is zero then subarray contains all the same numbers
    // and thus there is nothing to search for unless this range is all
    // consists of seek number.
    if (!rangeDelta) {
      // By doing this we're also avoiding division by zero while
      // calculating the middleIndex later.
      return sortedArray[leftIndex] === seekElement ? leftIndex : -1;
    }

    // Do interpolation of the middle index.
    const middleIndex = leftIndex + Math.floor(valueDelta * indexDelta / rangeDelta);

    // If we've found the element just return its position.
    if (sortedArray[middleIndex] === seekElement) {
      return middleIndex;
    }

    // Decide which half to choose for seeking next: left or right one.
    if (sortedArray[middleIndex] < seekElement) {
      // Go to the right half of the array.
      leftIndex = middleIndex + 1;
    } else {
      // Go to the left half of the array.
      rightIndex = middleIndex - 1;
    }
  }

  return -1;
}
```

Listing 129: jumpSearch.test.js

```
import jumpSearch from '../jumpSearch';

describe('jumpSearch', () => {
  it('should search for an element in sorted array', () => {
    expect(jumpSearch([], 1)).toBe(-1);
    expect(jumpSearch([1], 2)).toBe(-1);
    expect(jumpSearch([1], 1)).toBe(0);
    expect(jumpSearch([1, 2], 1)).toBe(0);
    expect(jumpSearch([1, 2], 1)).toBe(0);
    expect(jumpSearch([1, 1, 1], 1)).toBe(0);
    expect(jumpSearch([1, 2, 5, 10, 20, 21, 24, 30, 48], 2)).toBe(1);
    expect(jumpSearch([1, 2, 5, 10, 20, 21, 24, 30, 48], 0)).toBe(-1);
    expect(jumpSearch([1, 2, 5, 10, 20, 21, 24, 30, 48], 0)).toBe(-1);
    expect(jumpSearch([1, 2, 5, 10, 20, 21, 24, 30, 48], 7)).toBe(-1);
    expect(jumpSearch([1, 2, 5, 10, 20, 21, 24, 30, 48], 5)).toBe(2);
    expect(jumpSearch([1, 2, 5, 10, 20, 21, 24, 30, 48], 20)).toBe(4);
    expect(jumpSearch([1, 2, 5, 10, 20, 21, 24, 30, 48], 30)).toBe(7);
    expect(jumpSearch([1, 2, 5, 10, 20, 21, 24, 30, 48], 48)).toBe(8);
  });

  it('should search object in sorted array', () => {
    const sortedArrayOfObjects = [
      { key: 1, value: 'value1' },
      { key: 2, value: 'value2' },
      { key: 3, value: 'value3' },
    ];

    const comparator = (a, b) => {
      if (a.key === b.key) return 0;
      return a.key < b.key ? -1 : 1;
    };

    expect(jumpSearch([], { key: 1 }, comparator)).toBe(-1);
    expect(jumpSearch(sortedArrayOfObjects, { key: 4 }, comparator)).toBe(-1);
    expect(jumpSearch(sortedArrayOfObjects, { key: 1 }, comparator)).toBe(0);
    expect(jumpSearch(sortedArrayOfObjects, { key: 2 }, comparator)).toBe(1);
    expect(jumpSearch(sortedArrayOfObjects, { key: 3 }, comparator)).toBe(2);
  });
});
```

Listing 130: jumpSearch.js

```
import Comparator from '../../../utils/comparator/Comparator';

/**
 * Jump (block) search implementation.
 *
 * @param {*[]} sortedArray
 * @param {*} seekElement
 * @param {function(a, b)} [comparatorCallback]
 * @return {number}
 */
export default function jumpSearch(sortedArray, seekElement, comparatorCallback) {
  const comparator = new Comparator(comparatorCallback);
  const arraySize = sortedArray.length;

  if (!arraySize) {
    // We can't find anything in empty array.
    return -1;
  }

  // Calculate optimal jump size.
  // Total number of comparisons in the worst case will be ((arraySize/jumpSize) + jumpSize - 1).
  // The value of the function ((arraySize/jumpSize) + jumpSize - 1) will be minimum
  // when jumpSize = array.length.
  const jumpSize = Math.floor(Math.sqrt(arraySize));

  // Find the block where the seekElement belong to.
  let blockStart = 0;
  let blockEnd = jumpSize;
  while (comparator.greaterThan(seekElement, sortedArray[Math.min(blockEnd, arraySize) - 1])) {
    // Jump to the next block.
    blockStart = blockEnd;
    blockEnd += jumpSize;

    // If our next block is out of array then we couldn't found the element.
    if (blockStart > arraySize) {
      return -1;
    }
  }

  // Do linear search for seekElement in subarray starting from blockStart.
  let currentIndex = blockStart;
  while (currentIndex < Math.min(blockEnd, arraySize)) {
    if (comparator.equal(sortedArray[currentIndex], seekElement)) {
      return currentIndex;
    }

    currentIndex += 1;
  }

  return -1;
}
```

Listing 131: linearSearch.test.js

```
 import linearSearch from '../linearSearch';

describe('linearSearch', () => {
  it('should search all numbers in array', () => {
    const array = [1, 2, 4, 6, 2];

    expect(linearSearch(array, 10)).toEqual([]);
    expect(linearSearch(array, 1)).toEqual([0]);
    expect(linearSearch(array, 2)).toEqual([1, 4]);
  });

  it('should search all strings in array', () => {
    const array = ['a', 'b', 'a'];

    expect(linearSearch(array, 'c')).toEqual([]);
    expect(linearSearch(array, 'b')).toEqual([1]);
    expect(linearSearch(array, 'a')).toEqual([0, 2]);
  });

  it('should search through objects as well', () => {
    const comparatorCallback = (a, b) => {
      if (a.key === b.key) {
        return 0;
      }

      return a.key <= b.key ? -1 : 1;
    };

    const array = [
      { key: 5 },
      { key: 6 },
      { key: 7 },
      { key: 6 },
    ];

    expect(linearSearch(array, { key: 10 }, comparatorCallback)).toEqual([]);
    expect(linearSearch(array, { key: 5 }, comparatorCallback)).toEqual([0]);
    expect(linearSearch(array, { key: 6 }, comparatorCallback)).toEqual([1, 3]);
  });
});
```

Listing 132: linearSearch.js

```
import Comparator from '../../../utils/comparator/Comparator';

/**
 * Linear search implementation.
 *
 * @param {*[]} array
 * @param {*} seekElement
 * @param {function(a, b)} [comparatorCallback]
 * @return {number[]}
 */
export default function linearSearch(array, seekElement, comparatorCallback) {
  const comparator = new Comparator(comparatorCallback);
  const foundIndices = [];

  array.forEach((element, index) => {
    if (comparator.equal(element, seekElement)) {
      foundIndices.push(index);
    }
  });

  return foundIndices;
}
```

Listing 133: cartesianProduct.test.js

```
import cartesianProduct from '../cartesianProduct';

describe('cartesianProduct', () => {
  it('should return null if there is not enough info for calculation', () => {
    const product1 = cartesianProduct([1], null);
    const product2 = cartesianProduct([], null);

    expect(product1).toBeNull();
    expect(product2).toBeNull();
  });

  it('should calculate the product of two sets', () => {
    const product1 = cartesianProduct([1], [1]);
    const product2 = cartesianProduct([1, 2], [3, 5]);

    expect(product1).toEqual([[1, 1]]);
    expect(product2).toEqual([[1, 3], [1, 5], [2, 3], [2, 5]]);
  });
});
```

Listing 134: cartesianProduct.js

```
/**
 * Generates Cartesian Product of two sets.
 * @param {*[]} setA
 * @param {*[]} setB
 * @return {*[]}
 */
export default function cartesianProduct(setA, setB) {
  // Check if input sets are not empty.
  // Otherwise return null since we can't generate Cartesian Product out of them.
  if (!setA || !setB || !setA.length || !setB.length) {
    return null;
  }

  // Init product set.
  const product = [];

  // Now, let's go through all elements of a first and second set and form all possible pairs.
  for (let indexA = 0; indexA < setA.length; indexA += 1) {
    for (let indexB = 0; indexB < setB.length; indexB += 1) {
      // Add current product pair to the product set.
      product.push([setA[indexA], setB[indexB]]);
    }
  }

  // Return cartesian product set.
  return product;
}
```

Listing 135: combineWithoutRepetitions.test.js

```
import combineWithoutRepetitions from '../combineWithoutRepetitions';
import factorial from '../../../math/factorial/factorial';
import pascalTriangle from '../../../math/pascal-triangle/pascalTriangle';

describe('combineWithoutRepetitions', () => {
  it('should combine string without repetitions', () => {
    expect(combineWithoutRepetitions(['A', 'B'], 3)).toEqual([]);

    expect(combineWithoutRepetitions(['A', 'B'], 1)).toEqual([
      ['A'],
      ['B'],
    ]);

    expect(combineWithoutRepetitions(['A'], 1)).toEqual([
      ['A'],
    ]);

    expect(combineWithoutRepetitions(['A', 'B'], 2)).toEqual([
      ['A', 'B'],
    ]);

    expect(combineWithoutRepetitions(['A', 'B', 'C'], 2)).toEqual([
      ['A', 'B'],
      ['A', 'C'],
      ['B', 'C'],
    ]);

    expect(combineWithoutRepetitions(['A', 'B', 'C'], 3)).toEqual([
      ['A', 'B', 'C'],
    ]);

    expect(combineWithoutRepetitions(['A', 'B', 'C', 'D'], 3)).toEqual([
      ['A', 'B', 'C'],
      ['A', 'B', 'D'],
      ['A', 'C', 'D'],
      ['B', 'C', 'D'],
    ]);

    expect(combineWithoutRepetitions(['A', 'B', 'C', 'D', 'E'], 3)).toEqual([
      ['A', 'B', 'C'],
      ['A', 'B', 'D'],
      ['A', 'B', 'E'],
      ['A', 'C', 'D'],
      ['A', 'C', 'E'],
      ['A', 'D', 'E'],
      ['B', 'C', 'D'],
      ['B', 'C', 'E'],
      ['B', 'D', 'E'],
      ['C', 'D', 'E'],
    ]);

    const combinationOptions = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'];
    const combinationSlotsNumber = 4;
    const combinations = combineWithoutRepetitions(combinationOptions, combinationSlotsNumber);
    const n = combinationOptions.length;
    const r = combinationSlotsNumber;
    const expectedNumberOfCombinations = factorial(n) / (factorial(r) * factorial(n - r));

    expect(combinations.length).toBe(expectedNumberOfCombinations);

    // This one is just to see one of the way of Pascal's triangle application.
    expect(combinations.length).toBe(pascalTriangle(n)[r]);
  });
});
```

Listing 136: combineWithRepetitions.test.js

```
import combineWithRepetitions from '../combineWithRepetitions';
import factorial from '../../../math/factorial/factorial';

describe('combineWithRepetitions', () => {
  it('should combine string with repetitions', () => {
    expect(combineWithRepetitions(['A'], 1)).toEqual([
      ['A'],
    ]);

    expect(combineWithRepetitions(['A', 'B'], 1)).toEqual([
      ['A'],
      ['B'],
    ]);

    expect(combineWithRepetitions(['A', 'B'], 2)).toEqual([
      ['A', 'A'],
      ['A', 'B'],
      ['B', 'B'],
    ]);

    expect(combineWithRepetitions(['A', 'B'], 3)).toEqual([
      ['A', 'A', 'A'],
      ['A', 'A', 'B'],
      ['A', 'B', 'B'],
      ['B', 'B', 'B'],
    ]);

    expect(combineWithRepetitions(['A', 'B', 'C'], 2)).toEqual([
      ['A', 'A'],
      ['A', 'B'],
      ['A', 'C'],
      ['B', 'B'],
      ['B', 'C'],
      ['C', 'C'],
    ]);

    expect(combineWithRepetitions(['A', 'B', 'C'], 3)).toEqual([
      ['A', 'A', 'A'],
      ['A', 'A', 'B'],
      ['A', 'A', 'C'],
      ['A', 'B', 'B'],
      ['A', 'B', 'C'],
      ['A', 'C', 'C'],
      ['B', 'B', 'B'],
      ['B', 'B', 'C'],
      ['B', 'C', 'C'],
      ['C', 'C', 'C'],
    ]);

    const combinationOptions = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'];
    const combinationSlotsNumber = 4;
    const combinations = combineWithRepetitions(combinationOptions, combinationSlotsNumber);
    const n = combinationOptions.length;
    const r = combinationSlotsNumber;
    const expectedNumberOfCombinations = factorial((r + n) - 1) / (factorial(r) * factorial(n - 1));

    expect(combinations.length).toBe(expectedNumberOfCombinations);
  });
});
```

Listing 137: combineWithoutRepetitions.js

```
/**
 * @param {*[]} comboOptions
 * @param {number} comboLength
 * @return {*[]}
 */
export default function combineWithoutRepetitions(comboOptions, comboLength) {
  // If the length of the combination is 1 then each element of the original array
  // is a combination itself.
  if (comboLength === 1) {
    return comboOptions.map(comboOption => [comboOption]);
  }

  // Init combinations array.
  const combos = [];

  // Extract characters one by one and concatenate them to combinations of smaller lengths.
  // We need to extract them because we don't want to have repetitions after concatenation.
  comboOptions.forEach((currentOption, optionIndex) => {
    // Generate combinations of smaller size.
    const smallerCombos = combineWithoutRepetitions(
      comboOptions.slice(optionIndex + 1),
      comboLength - 1,
    );

    // Concatenate currentOption with all combinations of smaller size.
    smallerCombos.forEach((smallerCombo) => {
      combos.push([currentOption].concat(smallerCombo));
    });
  });

  return combos;
}
```

Listing 138: combineWithRepetitions.js

```
/**
 * @param {*[]} comboOptions
 * @param {number} comboLength
 * @return {*[]}
 */
export default function combineWithRepetitions(comboOptions, comboLength) {
  // If the length of the combination is 1 then each element of the original array
  // is a combination itself.
  if (comboLength === 1) {
    return comboOptions.map(comboOption => [comboOption]);
  }

  // Init combinations array.
  const combos = [];

  // Remember characters one by one and concatenate them to combinations of smaller lengths.
  // We don't extract elements here because the repetitions are allowed.
  comboOptions.forEach((currentOption, optionIndex) => {
    // Generate combinations of smaller size.
    const smallerCombos = combineWithRepetitions(
      comboOptions.slice(optionIndex),
      comboLength - 1,
    );

    // Concatenate currentOption with all combinations of smaller size.
    smallerCombos.forEach((smallerCombo) => {
      combos.push([currentOption].concat(smallerCombo));
    });
  });

  return combos;
}
```

Listing 139: combinationSum.test.js

```
import combinationSum from '../combinationSum';

describe('combinationSum', () => {
  it('should find all combinations with specific sum', () => {
    expect(combinationSum([1], 4)).toEqual([
      [1, 1, 1, 1],
    ]);

    expect(combinationSum([2, 3, 6, 7], 7)).toEqual([
      [2, 2, 3],
      [7],
    ]);

    expect(combinationSum([2, 3, 5], 8)).toEqual([
      [2, 2, 2, 2],
      [2, 3, 3],
      [3, 5],
    ]);

    expect(combinationSum([2, 5], 3)).toEqual([]);

    expect(combinationSum([], 3)).toEqual([]);
  });
});
```

Listing 140: combinationSum.js

```
/**
 * @param {number[]} candidates - candidate numbers we're picking from.
 * @param {number} remainingSum - remaining sum after adding candidates to currentCombination.
 * @param {number[][]} finalCombinations - resulting list of combinations.
 * @param {number[]} currentCombination - currently explored candidates.
 * @param {number} startFrom - index of the candidate to start further exploration from.
 * @return {number[][]}
 */
function combinationSumRecursive(
  candidates,
  remainingSum,
  finalCombinations = [],
  currentCombination = [],
  startFrom = 0,
) {
  if (remainingSum < 0) {
    // By adding another candidate we've gone below zero.
    // This would mean that the last candidate was not acceptable.
    return finalCombinations;
  }

  if (remainingSum === 0) {
    // If after adding the previous candidate our remaining sum
    // became zero - we need to save the current combination since it is one
    // of the answers we're looking for.
    finalCombinations.push(currentCombination.slice());

    return finalCombinations;
  }

  // If we haven't reached zero yet let's continue to add all
  // possible candidates that are left.
  for (let candidateIndex = startFrom; candidateIndex < candidates.length; candidateIndex += 1) {
    const currentCandidate = candidates[candidateIndex];

    // Let's try to add another candidate.
    currentCombination.push(currentCandidate);

    // Explore further option with current candidate being added.
    combinationSumRecursive(
      candidates,
      remainingSum - currentCandidate,
      finalCombinations,
      currentCombination,
      candidateIndex,
    );

    // BACKTRACKING.
    // Let's get back, exclude current candidate and try another ones later.
    currentCombination.pop();
  }

  return finalCombinations;
}

/**
 * Backtracking algorithm of finding all possible combination for specific sum.
 *
 * @param {number[]} candidates
 * @param {number} target
 * @return {number[][]}
 */
export default function combinationSum(candidates, target) {
  return combinationSumRecursive(candidates, target);
}
```

Listing 141: fisherYates.test.js

```
import fisherYates from '../fisherYates';
import { sortedArr } from '../../../sorting/SortTester';
import QuickSort from '../../../sorting/quick-sort/QuickSort';

describe('fisherYates', () => {
  it('should shuffle small arrays', () => {
    expect(fisherYates([])).toEqual([]);
    expect(fisherYates([1])).toEqual([1]);
  });

  it('should shuffle array randomly', () => {
    const shuffledArray = fisherYates(sortedArr);
    const sorter = new QuickSort();

    expect(shuffledArray.length).toBe(sortedArr.length);
    expect(shuffledArray).not.toEqual(sortedArr);
    expect(sorter.sort(shuffledArray)).toEqual(sortedArr);
  });
});
```

Listing 142: fisherYates.js

```
/**
 * @param {*[]} originalArray
 * @return {*[]}
 */
export default function fisherYates(originalArray) {
  // Clone array from preventing original array from modification (for testing purpose).
  const array = originalArray.slice(0);

  for (let i = (array.length - 1); i > 0; i -= 1) {
    const randomIndex = Math.floor(Math.random() * (i + 1));
    [array[i], array[randomIndex]] = [array[randomIndex], array[i]];
  }

  return array;
}
```

Listing 143: Knapsack.test.js

```
import Knapsack from '../Knapsack';
import KnapsackItem from '../KnapsackItem';

describe('Knapsack', () => {
  it('should solve 0/1 knapsack problem', () => {
    const possibleKnapsackItems = [
      new KnapsackItem({ value: 1, weight: 1 }),
      new KnapsackItem({ value: 4, weight: 3 }),
      new KnapsackItem({ value: 5, weight: 4 }),
      new KnapsackItem({ value: 7, weight: 5 }),
    ];

    const maxKnapsackWeight = 7;

    const knapsack = new Knapsack(possibleKnapsackItems, maxKnapsackWeight);

    knapsack.solveZeroOneKnapsackProblem();

    expect(knapsack.totalValue).toBe(9);
    expect(knapsack.totalWeight).toBe(7);
    expect(knapsack.selectedItems.length).toBe(2);
    expect(knapsack.selectedItems[0].toString()).toBe('v5 w4 x 1');
    expect(knapsack.selectedItems[1].toString()).toBe('v4 w3 x 1');
  });

  it('should solve 0/1 knapsack problem regardless of items order', () => {
    const possibleKnapsackItems = [
      new KnapsackItem({ value: 5, weight: 4 }),
      new KnapsackItem({ value: 1, weight: 1 }),
      new KnapsackItem({ value: 7, weight: 5 }),
      new KnapsackItem({ value: 4, weight: 3 }),
    ];

    const maxKnapsackWeight = 7;

    const knapsack = new Knapsack(possibleKnapsackItems, maxKnapsackWeight);

    knapsack.solveZeroOneKnapsackProblem();

    expect(knapsack.totalValue).toBe(9);
    expect(knapsack.totalWeight).toBe(7);
    expect(knapsack.selectedItems.length).toBe(2);
    expect(knapsack.selectedItems[0].toString()).toBe('v5 w4 x 1');
    expect(knapsack.selectedItems[1].toString()).toBe('v4 w3 x 1');
  });

  it('should solve 0/1 knapsack problem with impossible items set', () => {
    const possibleKnapsackItems = [
      new KnapsackItem({ value: 5, weight: 40 }),
      new KnapsackItem({ value: 1, weight: 10 }),
      new KnapsackItem({ value: 7, weight: 50 }),
      new KnapsackItem({ value: 4, weight: 30 }),
    ];

    const maxKnapsackWeight = 7;

    const knapsack = new Knapsack(possibleKnapsackItems, maxKnapsackWeight);

    knapsack.solveZeroOneKnapsackProblem();

    expect(knapsack.totalValue).toBe(0);
    expect(knapsack.totalWeight).toBe(0);
    expect(knapsack.selectedItems.length).toBe(0);
  });

  it('should solve 0/1 knapsack problem with all equal weights', () => {
    const possibleKnapsackItems = [
      new KnapsackItem({ value: 5, weight: 1 }),
      new KnapsackItem({ value: 1, weight: 1 }),
      new KnapsackItem({ value: 7, weight: 1 }),
      new KnapsackItem({ value: 4, weight: 1 }),
      new KnapsackItem({ value: 4, weight: 1 }),
      new KnapsackItem({ value: 4, weight: 1 }),
    ];

    const maxKnapsackWeight = 3;

    const knapsack = new Knapsack(possibleKnapsackItems, maxKnapsackWeight);
```

```
    knapsack.solveZeroOneKnapsackProblem();

    expect(knapsack.totalValue).toBe(16);
    expect(knapsack.totalWeight).toBe(3);
    expect(knapsack.selectedItems.length).toBe(3);
    expect(knapsack.selectedItems[0].toString()).toBe('v4 w1 x 1');
    expect(knapsack.selectedItems[1].toString()).toBe('v5 w1 x 1');
    expect(knapsack.selectedItems[2].toString()).toBe('v7 w1 x 1');
  });

  it('should solve unbound knapsack problem', () => {
    const possibleKnapsackItems = [
      new KnapsackItem({ value: 84, weight: 7 }), // v/w ratio is 12
      new KnapsackItem({ value: 5, weight: 2 }), // v/w ratio is 2.5
      new KnapsackItem({ value: 12, weight: 3 }), // v/w ratio is 4
      new KnapsackItem({ value: 10, weight: 1 }), // v/w ratio is 10
      new KnapsackItem({ value: 20, weight: 2 }), // v/w ratio is 10
    ];

    const maxKnapsackWeight = 15;

    const knapsack = new Knapsack(possibleKnapsackItems, maxKnapsackWeight);

    knapsack.solveUnboundedKnapsackProblem();

    expect(knapsack.totalValue).toBe(84 + 20 + 12 + 10 + 5);
    expect(knapsack.totalWeight).toBe(15);
    expect(knapsack.selectedItems.length).toBe(5);
    expect(knapsack.selectedItems[0].toString()).toBe('v84 w7 x 1');
    expect(knapsack.selectedItems[1].toString()).toBe('v20 w2 x 1');
    expect(knapsack.selectedItems[2].toString()).toBe('v10 w1 x 1');
    expect(knapsack.selectedItems[3].toString()).toBe('v12 w3 x 1');
    expect(knapsack.selectedItems[4].toString()).toBe('v5 w2 x 1');
  });

  it('should solve unbound knapsack problem with items in stock', () => {
    const possibleKnapsackItems = [
      new KnapsackItem({ value: 84, weight: 7, itemsInStock: 3 }), // v/w ratio is 12
      new KnapsackItem({ value: 5, weight: 2, itemsInStock: 2 }), // v/w ratio is 2.5
      new KnapsackItem({ value: 12, weight: 3, itemsInStock: 1 }), // v/w ratio is 4
      new KnapsackItem({ value: 10, weight: 1, itemsInStock: 6 }), // v/w ratio is 10
      new KnapsackItem({ value: 20, weight: 2, itemsInStock: 8 }), // v/w ratio is 10
    ];

    const maxKnapsackWeight = 17;

    const knapsack = new Knapsack(possibleKnapsackItems, maxKnapsackWeight);

    knapsack.solveUnboundedKnapsackProblem();

    expect(knapsack.totalValue).toBe(84 + 84 + 20 + 10);
    expect(knapsack.totalWeight).toBe(17);
    expect(knapsack.selectedItems.length).toBe(3);
    expect(knapsack.selectedItems[0].toString()).toBe('v84 w7 x 2');
    expect(knapsack.selectedItems[1].toString()).toBe('v20 w2 x 1');
    expect(knapsack.selectedItems[2].toString()).toBe('v10 w1 x 1');
  });

  it('should solve unbound knapsack problem with items in stock and max weight more than sum of all items',
      () => {
    const possibleKnapsackItems = [
      new KnapsackItem({ value: 84, weight: 7, itemsInStock: 3 }), // v/w ratio is 12
      new KnapsackItem({ value: 5, weight: 2, itemsInStock: 2 }), // v/w ratio is 2.5
      new KnapsackItem({ value: 12, weight: 3, itemsInStock: 1 }), // v/w ratio is 4
      new KnapsackItem({ value: 10, weight: 1, itemsInStock: 6 }), // v/w ratio is 10
      new KnapsackItem({ value: 20, weight: 2, itemsInStock: 8 }), // v/w ratio is 10
    ];

    const maxKnapsackWeight = 60;

    const knapsack = new Knapsack(possibleKnapsackItems, maxKnapsackWeight);

    knapsack.solveUnboundedKnapsackProblem();

    expect(knapsack.totalValue).toBe((3 * 84) + (2 * 5) + (1 * 12) + (6 * 10) + (8 * 20));
    expect(knapsack.totalWeight).toBe((3 * 7) + (2 * 2) + (1 * 3) + (6 * 1) + (8 * 2));
    expect(knapsack.selectedItems.length).toBe(5);
    expect(knapsack.selectedItems[0].toString()).toBe('v84 w7 x 3');
    expect(knapsack.selectedItems[1].toString()).toBe('v20 w2 x 8');
    expect(knapsack.selectedItems[2].toString()).toBe('v10 w1 x 6');
    expect(knapsack.selectedItems[3].toString()).toBe('v12 w3 x 1');
    expect(knapsack.selectedItems[4].toString()).toBe('v5 w2 x 2');
```

```
  });
});
```

Listing 144: KnapsackItem.test.js

```
import KnapsackItem from '../KnapsackItem';

describe('KnapsackItem', () => {
  it('should create knapsack item and count its total weight and value', () => {
    const knapsackItem = new KnapsackItem({ value: 3, weight: 2 });

    expect(knapsackItem.value).toBe(3);
    expect(knapsackItem.weight).toBe(2);
    expect(knapsackItem.quantity).toBe(1);
    expect(knapsackItem.valuePerWeightRatio).toBe(1.5);
    expect(knapsackItem.toString()).toBe('v3 w2 x 1');
    expect(knapsackItem.totalValue).toBe(3);
    expect(knapsackItem.totalWeight).toBe(2);

    knapsackItem.quantity = 0;

    expect(knapsackItem.value).toBe(3);
    expect(knapsackItem.weight).toBe(2);
    expect(knapsackItem.quantity).toBe(0);
    expect(knapsackItem.valuePerWeightRatio).toBe(1.5);
    expect(knapsackItem.toString()).toBe('v3 w2 x 0');
    expect(knapsackItem.totalValue).toBe(0);
    expect(knapsackItem.totalWeight).toBe(0);

    knapsackItem.quantity = 2;

    expect(knapsackItem.value).toBe(3);
    expect(knapsackItem.weight).toBe(2);
    expect(knapsackItem.quantity).toBe(2);
    expect(knapsackItem.valuePerWeightRatio).toBe(1.5);
    expect(knapsackItem.toString()).toBe('v3 w2 x 2');
    expect(knapsackItem.totalValue).toBe(6);
    expect(knapsackItem.totalWeight).toBe(4);
  });
});
```

Listing 145: Knapsack.js

```
 import MergeSort from '../../sorting/merge-sort/MergeSort';

export default class Knapsack {
  /**
   * @param {KnapsackItem[]} possibleItems
   * @param {number} weightLimit
   */
  constructor(possibleItems, weightLimit) {
    this.selectedItems = [];
    this.weightLimit = weightLimit;
    this.possibleItems = possibleItems;
  }

  sortPossibleItemsByWeight() {
    this.possibleItems = new MergeSort({
      /**
       * @var KnapsackItem itemA
       * @var KnapsackItem itemB
       */
      compareCallback: (itemA, itemB) => {
        if (itemA.weight === itemB.weight) {
          return 0;
        }

        return itemA.weight < itemB.weight ? -1 : 1;
      },
    }).sort(this.possibleItems);
  }

  sortPossibleItemsByValue() {
    this.possibleItems = new MergeSort({
      /**
       * @var KnapsackItem itemA
       * @var KnapsackItem itemB
       */
      compareCallback: (itemA, itemB) => {
        if (itemA.value === itemB.value) {
          return 0;
        }

        return itemA.value > itemB.value ? -1 : 1;
      },
    }).sort(this.possibleItems);
  }

  sortPossibleItemsByValuePerWeightRatio() {
    this.possibleItems = new MergeSort({
      /**
       * @var KnapsackItem itemA
       * @var KnapsackItem itemB
       */
      compareCallback: (itemA, itemB) => {
        if (itemA.valuePerWeightRatio === itemB.valuePerWeightRatio) {
          return 0;
        }

        return itemA.valuePerWeightRatio > itemB.valuePerWeightRatio ? -1 : 1;
      },
    }).sort(this.possibleItems);
  }

  // Solve 0/1 knapsack problem
  // Dynamic Programming approach.
  solveZeroOneKnapsackProblem() {
    // We do two sorts because in case of equal weights but different values
    // we need to take the most valuable items first.
    this.sortPossibleItemsByValue();
    this.sortPossibleItemsByWeight();

    this.selectedItems = [];

    // Create knapsack values matrix.
    const numberOfRows = this.possibleItems.length;
    const numberOfColumns = this.weightLimit;
    const knapsackMatrix = Array(numberOfRows).fill(null).map(() => {
      return Array(numberOfColumns + 1).fill(null);
    });

    // Fill the first column with zeros since it would mean that there is
```

```
  // no items we can add to knapsack in case if weight limitation is zero.
  for (let itemIndex = 0; itemIndex < this.possibleItems.length; itemIndex += 1) {
    knapsackMatrix[itemIndex][0] = 0;
  }

  // Fill the first row with max possible values we would get by just adding
  // or not adding the first item to the knapsack.
  for (let weightIndex = 1; weightIndex <= this.weightLimit; weightIndex += 1) {
    const itemIndex = 0;
    const itemWeight = this.possibleItems[itemIndex].weight;
    const itemValue = this.possibleItems[itemIndex].value;
    knapsackMatrix[itemIndex][weightIndex] = itemWeight <= weightIndex ? itemValue : 0;
  }

  // Go through combinations of how we may add items to knapsack and
  // define what weight/value we would receive using Dynamic Programming
  // approach.
  for (let itemIndex = 1; itemIndex < this.possibleItems.length; itemIndex += 1) {
    for (let weightIndex = 1; weightIndex <= this.weightLimit; weightIndex += 1) {
      const currentItemWeight = this.possibleItems[itemIndex].weight;
      const currentItemValue = this.possibleItems[itemIndex].value;

      if (currentItemWeight > weightIndex) {
        // In case if item's weight is bigger then currently allowed weight
        // then we can't add it to knapsack and the max possible value we can
        // gain at the moment is the max value we got for previous item.
        knapsackMatrix[itemIndex][weightIndex] = knapsackMatrix[itemIndex - 1][weightIndex];
      } else {
        // Else we need to consider the max value we can gain at this point by adding
        // current value or just by keeping the previous item for current weight.
        knapsackMatrix[itemIndex][weightIndex] = Math.max(
          currentItemValue + knapsackMatrix[itemIndex - 1][weightIndex - currentItemWeight],
          knapsackMatrix[itemIndex - 1][weightIndex],
        );
      }
    }
  }

  // Now let's trace back the knapsack matrix to see what items we're going to add
  // to the knapsack.
  let itemIndex = this.possibleItems.length - 1;
  let weightIndex = this.weightLimit;

  while (itemIndex > 0) {
    const currentItem = this.possibleItems[itemIndex];
    const prevItem = this.possibleItems[itemIndex - 1];

    // Check if matrix value came from top (from previous item).
    // In this case this would mean that we need to include previous item
    // to the list of selected items.
    if (
      knapsackMatrix[itemIndex][weightIndex]
      && knapsackMatrix[itemIndex][weightIndex] === knapsackMatrix[itemIndex - 1][weightIndex]
    ) {
      // Check if there are several items with the same weight but with the different values.
      // We need to add highest item in the matrix that is possible to get the highest value.
      const prevSumValue = knapsackMatrix[itemIndex - 1][weightIndex];
      const prevPrevSumValue = knapsackMatrix[itemIndex - 2][weightIndex];
      if (
        !prevSumValue
        || (prevSumValue && prevPrevSumValue !== prevSumValue)
      ) {
        this.selectedItems.push(prevItem);
      }
    } else if (knapsackMatrix[itemIndex - 1][weightIndex - currentItem.weight]) {
      this.selectedItems.push(prevItem);
      weightIndex -= currentItem.weight;
    }

    itemIndex -= 1;
  }
}


// Solve unbounded knapsack problem.
// Greedy approach.
solveUnboundedKnapsackProblem() {
  this.sortPossibleItemsByValue();
  this.sortPossibleItemsByValuePerWeightRatio();

  for (let itemIndex = 0; itemIndex < this.possibleItems.length; itemIndex += 1) {
    if (this.totalWeight < this.weightLimit) {
```

```
        const currentItem = this.possibleItems[itemIndex];

        // Detect how much of current items we can push to knapsack.
        const availableWeight = this.weightLimit - this.totalWeight;
        const maxPossibleItemsCount = Math.floor(availableWeight / currentItem.weight);

        if (maxPossibleItemsCount > currentItem.itemsInStock) {
          // If we have more items in stock then it is allowed to add
          // let's add the maximum allowed number of them.
          currentItem.quantity = currentItem.itemsInStock;
        } else if (maxPossibleItemsCount) {
          // In case if we don't have specified number of items in stock
          // let's add only items we have in stock.
          currentItem.quantity = maxPossibleItemsCount;
        }

        this.selectedItems.push(currentItem);
      }
    }
  }

  get totalValue() {
    /** @var {KnapsackItem} item */
    return this.selectedItems.reduce((accumulator, item) => {
      return accumulator + item.totalValue;
    }, 0);
  }

  get totalWeight() {
    /** @var {KnapsackItem} item */
    return this.selectedItems.reduce((accumulator, item) => {
      return accumulator + item.totalWeight;
    }, 0);
  }
}
```

Listing 146: KnapsackItem.js

```
export default class KnapsackItem {
  /**
   * @param {Object} itemSettings - knapsack item settings,
   * @param {number} itemSettings.value - value of the item.
   * @param {number} itemSettings.weight - weight of the item.
   * @param {number} itemSettings.itemsInStock - how many items are available to be added.
   */
  constructor({ value, weight, itemsInStock = 1 }) {
    this.value = value;
    this.weight = weight;
    this.itemsInStock = itemsInStock;
    // Actual number of items that is going to be added to knapsack.
    this.quantity = 1;
  }

  get totalValue() {
    return this.value * this.quantity;
  }

  get totalWeight() {
    return this.weight * this.quantity;
  }

  // This coefficient shows how valuable the 1 unit of weight is
  // for current item.
  get valuePerWeightRatio() {
    return this.value / this.weight;
  }

  toString() {
    return `v${this.value} w${this.weight} x ${this.quantity}`;
  }
}
```

Listing 147: longestCommonSubsequence.test.js

```
 import longestCommonSubsequence from '../longestCommonSubsequence';

describe('longestCommonSubsequence', () => {
  it('should find longest common subsequence for two strings', () => {
    expect(longestCommonSubsequence([''], [''])).toEqual(['']);

    expect(longestCommonSubsequence([''], ['A', 'B', 'C'])).toEqual(['']);

    expect(longestCommonSubsequence(['A', 'B', 'C'], [''])).toEqual(['']);

    expect(longestCommonSubsequence(
      ['A', 'B', 'C'],
      ['D', 'E', 'F', 'G'],
    )).toEqual(['']);

    expect(longestCommonSubsequence(
      ['A', 'B', 'C', 'D', 'G', 'H'],
      ['A', 'E', 'D', 'F', 'H', 'R'],
    )).toEqual(['A', 'D', 'H']);

    expect(longestCommonSubsequence(
      ['A', 'G', 'G', 'T', 'A', 'B'],
      ['G', 'X', 'T', 'X', 'A', 'Y', 'B'],
    )).toEqual(['G', 'T', 'A', 'B']);

    expect(longestCommonSubsequence(
      ['A', 'B', 'C', 'D', 'A', 'F'],
      ['A', 'C', 'B', 'C', 'F'],
    )).toEqual(['A', 'B', 'C', 'F']);
  });
});
```

Listing 148: longestCommonSubsequence.js

```
/**
 * @param {string[]} set1
 * @param {string[]} set2
 * @return {string[]}
 */
export default function longestCommonSubsequence(set1, set2) {
  // Init LCS matrix.
  const lcsMatrix = Array(set2.length + 1).fill(null).map(() => Array(set1.length + 1).fill(null));

  // Fill first row with zeros.
  for (let columnIndex = 0; columnIndex <= set1.length; columnIndex += 1) {
    lcsMatrix[0][columnIndex] = 0;
  }

  // Fill first column with zeros.
  for (let rowIndex = 0; rowIndex <= set2.length; rowIndex += 1) {
    lcsMatrix[rowIndex][0] = 0;
  }

  // Fill rest of the column that correspond to each of two strings.
  for (let rowIndex = 1; rowIndex <= set2.length; rowIndex += 1) {
    for (let columnIndex = 1; columnIndex <= set1.length; columnIndex += 1) {
      if (set1[columnIndex - 1] === set2[rowIndex - 1]) {
        lcsMatrix[rowIndex][columnIndex] = lcsMatrix[rowIndex - 1][columnIndex - 1] + 1;
      } else {
        lcsMatrix[rowIndex][columnIndex] = Math.max(
          lcsMatrix[rowIndex - 1][columnIndex],
          lcsMatrix[rowIndex][columnIndex - 1],
        );
      }
    }
  }

  // Calculate LCS based on LCS matrix.
  if (!lcsMatrix[set2.length][set1.length]) {
    // If the length of largest common string is zero then return empty string.
    return [''];
  }

  const longestSequence = [];
  let columnIndex = set1.length;
  let rowIndex = set2.length;

  while (columnIndex > 0 || rowIndex > 0) {
    if (set1[columnIndex - 1] === set2[rowIndex - 1]) {
      // Move by diagonal left-top.
      longestSequence.unshift(set1[columnIndex - 1]);
      columnIndex -= 1;
      rowIndex -= 1;
    } else if (lcsMatrix[rowIndex][columnIndex] === lcsMatrix[rowIndex][columnIndex - 1]) {
      // Move left.
      columnIndex -= 1;
    } else {
      // Move up.
      rowIndex -= 1;
    }
  }

  return longestSequence;
}
```

Listing 149: dpLongestIncreasingSubsequence.test.js

```
import dpLongestIncreasingSubsequence from '../dpLongestIncreasingSubsequence';

describe('dpLongestIncreasingSubsequence', () => {
  it('should find longest increasing subsequence length', () => {
    // Should be:
    // 9 or
    // 8 or
    // 7 or
    // 6 or
    // ...
    expect(dpLongestIncreasingSubsequence([
      9, 8, 7, 6, 5, 4, 3, 2, 1, 0,
    ])).toBe(1);

    // Should be:
    // 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    expect(dpLongestIncreasingSubsequence([
      0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
    ])).toBe(10);

    // Should be:
    // -1, 0, 2, 3
    expect(dpLongestIncreasingSubsequence([
      3, 4, -1, 0, 6, 2, 3,
    ])).toBe(4);

    // Should be:
    // 0, 2, 6, 9, 11, 15 or
    // 0, 4, 6, 9, 11, 15 or
    // 0, 2, 6, 9, 13, 15 or
    // 0, 4, 6, 9, 13, 15
    expect(dpLongestIncreasingSubsequence([
      0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15,
    ])).toBe(6);
  });
});
```

Listing 150: dpLongestIncreasingSubsequence.js

```
/**
 * Dynamic programming approach to find longest increasing subsequence.
 * Complexity: O(n * n)
 *
 * @param {number[]} sequence
 * @return {number}
 */
export default function dpLongestIncreasingSubsequence(sequence) {
  // Create array with longest increasing substrings length and
  // fill it with 1-s that would mean that each element of the sequence
  // is itself a minimum increasing subsequence.
  const lengthsArray = Array(sequence.length).fill(1);

  let previousElementIndex = 0;
  let currentElementIndex = 1;

  while (currentElementIndex < sequence.length) {
    if (sequence[previousElementIndex] < sequence[currentElementIndex]) {
      // If current element is bigger then the previous one then
      // current element is a part of increasing subsequence which
      // length is by one bigger then the length of increasing subsequence
      // for previous element.
      const newLength = lengthsArray[previousElementIndex] + 1;
      if (newLength > lengthsArray[currentElementIndex]) {
        // Increase only if previous element would give us bigger subsequence length
        // then we already have for current element.
        lengthsArray[currentElementIndex] = newLength;
      }
    }

    // Move previous element index right.
    previousElementIndex += 1;

    // If previous element index equals to current element index then
    // shift current element right and reset previous element index to zero.
    if (previousElementIndex === currentElementIndex) {
      currentElementIndex += 1;
      previousElementIndex = 0;
    }
  }

  // Find the biggest element in lengthsArray.
  // This number is the biggest length of increasing subsequence.
  let longestIncreasingLength = 0;

  for (let i = 0; i < lengthsArray.length; i += 1) {
    if (lengthsArray[i] > longestIncreasingLength) {
      longestIncreasingLength = lengthsArray[i];
    }
  }

  return longestIncreasingLength;
}
```

Listing 151: bfMaximumSubarray.test.js

```
import bfMaximumSubarray from '../bfMaximumSubarray';

describe('bfMaximumSubarray', () => {
  it('should find maximum subarray using brute force algorithm', () => {
    expect(bfMaximumSubarray([])).toEqual([]);
    expect(bfMaximumSubarray([0, 0])).toEqual([0]);
    expect(bfMaximumSubarray([0, 0, 1])).toEqual([0, 0, 1]);
    expect(bfMaximumSubarray([0, 0, 1, 2])).toEqual([0, 0, 1, 2]);
    expect(bfMaximumSubarray([0, 0, -1, 2])).toEqual([2]);
    expect(bfMaximumSubarray([-1, -2, -3, -4, -5])).toEqual([-1]);
    expect(bfMaximumSubarray([1, 2, 3, 2, 3, 4, 5])).toEqual([1, 2, 3, 2, 3, 4, 5]);
    expect(bfMaximumSubarray([-2, 1, -3, 4, -1, 2, 1, -5, 4])).toEqual([4, -1, 2, 1]);
    expect(bfMaximumSubarray([-2, -3, 4, -1, -2, 1, 5, -3])).toEqual([4, -1, -2, 1, 5]);
    expect(bfMaximumSubarray([1, -3, 2, -5, 7, 6, -1, 4, 11, -23])).toEqual([7, 6, -1, 4, 11]);
  });
});
```

Listing 152: dpMaximumSubarray.test.js

```
import dpMaximumSubarray from '../dpMaximumSubarray';

describe('dpMaximumSubarray', () => {
  it('should find maximum subarray using dynamic programming algorithm', () => {
    expect(dpMaximumSubarray([])).toEqual([]);
    expect(dpMaximumSubarray([0, 0])).toEqual([0]);
    expect(dpMaximumSubarray([0, 0, 1])).toEqual([0, 0, 1]);
    expect(dpMaximumSubarray([0, 0, 1, 2])).toEqual([0, 0, 1, 2]);
    expect(dpMaximumSubarray([0, 0, -1, 2])).toEqual([2]);
    expect(dpMaximumSubarray([-1, -2, -3, -4, -5])).toEqual([-1]);
    expect(dpMaximumSubarray([1, 2, 3, 2, 3, 4, 5])).toEqual([1, 2, 3, 2, 3, 4, 5]);
    expect(dpMaximumSubarray([-2, 1, -3, 4, -1, 2, 1, -5, 4])).toEqual([4, -1, 2, 1]);
    expect(dpMaximumSubarray([-2, -3, 4, -1, -2, 1, 5, -3])).toEqual([4, -1, -2, 1, 5]);
    expect(dpMaximumSubarray([1, -3, 2, -5, 7, 6, -1, 4, 11, -23])).toEqual([7, 6, -1, 4, 11]);
  });
});
```

Listing 153: bfMaximumSubarray.js

```
/**
 * Brute Force solution.
 * Complexity: O(n^2)
 *
 * @param {Number[]} inputArray
 * @return {Number[]}
 */
export default function bfMaximumSubarray(inputArray) {
  let maxSubarrayStartIndex = 0;
  let maxSubarrayLength = 0;
  let maxSubarraySum = null;

  for (let startIndex = 0; startIndex < inputArray.length; startIndex += 1) {
    let subarraySum = 0;
    for (let arrLength = 1; arrLength <= (inputArray.length - startIndex); arrLength += 1) {
      subarraySum += inputArray[startIndex + (arrLength - 1)];
      if (maxSubarraySum === null || subarraySum > maxSubarraySum) {
        maxSubarraySum = subarraySum;
        maxSubarrayStartIndex = startIndex;
        maxSubarrayLength = arrLength;
      }
    }
  }

  return inputArray.slice(maxSubarrayStartIndex, maxSubarrayStartIndex + maxSubarrayLength);
}
```

Listing 154: dpMaximumSubarray.js

```
/**
 * Dynamic Programming solution.
 * Complexity: O(n)
 *
 * @param {Number[]} inputArray
 * @return {Number[]}
 */
export default function dpMaximumSubarray(inputArray) {
  // We iterate through the inputArray once, using a greedy approach to keep track of the maximum
  // sum we've seen so far and the current sum.
  //
  // The currentSum variable gets reset to 0 every time it drops below 0.
  //
  // The maxSum variable is set to -Infinity so that if all numbers are negative, the highest
  // negative number will constitute the maximum subarray.

  let maxSum = -Infinity;
  let currentSum = 0;

  // We need to keep track of the starting and ending indices that contributed to our maxSum
  // so that we can return the actual subarray. From the beginning let's assume that whole array
  // is contributing to maxSum.
  let maxStartIndex = 0;
  let maxEndIndex = inputArray.length - 1;
  let currentStartIndex = 0;

  inputArray.forEach((currentNumber, currentIndex) => {
    currentSum += currentNumber;

    // Update maxSum and the corresponding indices if we have found a new max.
    if (maxSum < currentSum) {
      maxSum = currentSum;
      maxStartIndex = currentStartIndex;
      maxEndIndex = currentIndex;
    }

    // Reset currentSum and currentStartIndex if currentSum drops below 0.
    if (currentSum < 0) {
      currentSum = 0;
      currentStartIndex = currentIndex + 1;
    }
  });

  return inputArray.slice(maxStartIndex, maxEndIndex + 1);
}
```

Listing 155: permutateWithoutRepetitions.test.js

```
import permutateWithoutRepetitions from '../permutateWithoutRepetitions';
import factorial from '../../../math/factorial/factorial';

describe('permutateWithoutRepetitions', () => {
  it('should permutate string', () => {
    const permutations1 = permutateWithoutRepetitions(['A']);
    expect(permutations1).toEqual([
      ['A'],
    ]);

    const permutations2 = permutateWithoutRepetitions(['A', 'B']);
    expect(permutations2.length).toBe(2);
    expect(permutations2).toEqual([
      ['A', 'B'],
      ['B', 'A'],
    ]);

    const permutations6 = permutateWithoutRepetitions(['A', 'A']);
    expect(permutations6.length).toBe(2);
    expect(permutations6).toEqual([
      ['A', 'A'],
      ['A', 'A'],
    ]);

    const permutations3 = permutateWithoutRepetitions(['A', 'B', 'C']);
    expect(permutations3.length).toBe(factorial(3));
    expect(permutations3).toEqual([
      ['A', 'B', 'C'],
      ['B', 'A', 'C'],
      ['B', 'C', 'A'],
      ['A', 'C', 'B'],
      ['C', 'A', 'B'],
      ['C', 'B', 'A'],
    ]);

    const permutations4 = permutateWithoutRepetitions(['A', 'B', 'C', 'D']);
    expect(permutations4.length).toBe(factorial(4));
    expect(permutations4).toEqual([
      ['A', 'B', 'C', 'D'],
      ['B', 'A', 'C', 'D'],
      ['B', 'C', 'A', 'D'],
      ['B', 'C', 'D', 'A'],
      ['A', 'C', 'B', 'D'],
      ['C', 'A', 'B', 'D'],
      ['C', 'B', 'A', 'D'],
      ['C', 'B', 'D', 'A'],
      ['A', 'C', 'D', 'B'],
      ['C', 'A', 'D', 'B'],
      ['C', 'D', 'A', 'B'],
      ['C', 'D', 'B', 'A'],
      ['A', 'B', 'D', 'C'],
      ['B', 'A', 'D', 'C'],
      ['B', 'D', 'A', 'C'],
      ['B', 'D', 'C', 'A'],
      ['A', 'D', 'B', 'C'],
      ['D', 'A', 'B', 'C'],
      ['D', 'B', 'A', 'C'],
      ['D', 'B', 'C', 'A'],
      ['A', 'D', 'C', 'B'],
      ['D', 'A', 'C', 'B'],
      ['D', 'C', 'A', 'B'],
      ['D', 'C', 'B', 'A'],
    ]);

    const permutations5 = permutateWithoutRepetitions(['A', 'B', 'C', 'D', 'E', 'F']);
    expect(permutations5.length).toBe(factorial(6));
  });
});
```

Listing 156: permutateWithRepetitions.test.js

```
import permutateWithRepetitions from '../permutateWithRepetitions';

describe('permutateWithRepetitions', () => {
  it('should permutate string with repetition', () => {
    const permutations1 = permutateWithRepetitions(['A']);
    expect(permutations1).toEqual([
      ['A'],
    ]);

    const permutations2 = permutateWithRepetitions(['A', 'B']);
    expect(permutations2).toEqual([
      ['A', 'A'],
      ['A', 'B'],
      ['B', 'A'],
      ['B', 'B'],
    ]);

    const permutations3 = permutateWithRepetitions(['A', 'B', 'C']);
    expect(permutations3).toEqual([
      ['A', 'A', 'A'],
      ['A', 'A', 'B'],
      ['A', 'A', 'C'],
      ['A', 'B', 'A'],
      ['A', 'B', 'B'],
      ['A', 'B', 'C'],
      ['A', 'C', 'A'],
      ['A', 'C', 'B'],
      ['A', 'C', 'C'],
      ['B', 'A', 'A'],
      ['B', 'A', 'B'],
      ['B', 'A', 'C'],
      ['B', 'B', 'A'],
      ['B', 'B', 'B'],
      ['B', 'B', 'C'],
      ['B', 'C', 'A'],
      ['B', 'C', 'B'],
      ['B', 'C', 'C'],
      ['C', 'A', 'A'],
      ['C', 'A', 'B'],
      ['C', 'A', 'C'],
      ['C', 'B', 'A'],
      ['C', 'B', 'B'],
      ['C', 'B', 'C'],
      ['C', 'C', 'A'],
      ['C', 'C', 'B'],
      ['C', 'C', 'C'],
    ]);

    const permutations4 = permutateWithRepetitions(['A', 'B', 'C', 'D']);
    expect(permutations4.length).toBe(4 * 4 * 4 * 4);
  });
});
```

Listing 157: permutateWithoutRepetitions.js

```
/**
 * @param {*[]} permutationOptions
 * @return {*[]}
 */
export default function permutateWithoutRepetitions(permutationOptions) {
  if (permutationOptions.length === 1) {
    return [permutationOptions];
  }

  // Init permutations array.
  const permutations = [];

  // Get all permutations for permutationOptions excluding the first element.
  const smallerPermutations = permutateWithoutRepetitions(permutationOptions.slice(1));

  // Insert first option into every possible position of every smaller permutation.
  const firstOption = permutationOptions[0];

  for (let permIndex = 0; permIndex < smallerPermutations.length; permIndex += 1) {
    const smallerPermutation = smallerPermutations[permIndex];

    // Insert first option into every possible position of smallerPermutation.
    for (let positionIndex = 0; positionIndex <= smallerPermutation.length; positionIndex += 1) {
      const permutationPrefix = smallerPermutation.slice(0, positionIndex);
      const permutationSuffix = smallerPermutation.slice(positionIndex);
      permutations.push(permutationPrefix.concat([firstOption], permutationSuffix));
    }
  }

  return permutations;
}
```

Listing 158: permutateWithRepetitions.js

```
/**
 * @param {*[]} permutationOptions
 * @param {number} permutationLength
 * @return {*[]}
 */
export default function permutateWithRepetitions(
  permutationOptions,
  permutationLength = permutationOptions.length,
) {
  if (permutationLength === 1) {
    return permutationOptions.map(permutationOption => [permutationOption]);
  }

  // Init permutations array.
  const permutations = [];

  // Get smaller permutations.
  const smallerPermutations = permutateWithRepetitions(
    permutationOptions,
    permutationLength - 1,
  );

  // Go through all options and join it to the smaller permutations.
  permutationOptions.forEach((currentOption) => {
    smallerPermutations.forEach((smallerPermutation) => {
      permutations.push([currentOption].concat(smallerPermutation));
    });
  });

  return permutations;
}
```

Listing 159: btPowerSet.test.js

```
import btPowerSet from '../btPowerSet';

describe('btPowerSet', () => {
  it('should calculate power set of given set using backtracking approach', () => {
    expect(btPowerSet([1])).toEqual([
      [],
      [1],
    ]);

    expect(btPowerSet([1, 2, 3])).toEqual([
      [],
      [1],
      [1, 2],
      [1, 2, 3],
      [1, 3],
      [2],
      [2, 3],
      [3],
    ]);
  });
});
```

Listing 160: bwPowerSet.test.js

```
import bwPowerSet from '../bwPowerSet';

describe('bwPowerSet', () => {
  it('should calculate power set of given set using bitwise approach', () => {
    expect(bwPowerSet([1])).toEqual([
      [],
      [1],
    ]);

    expect(bwPowerSet([1, 2, 3])).toEqual([
      [],
      [1],
      [2],
      [1, 2],
      [3],
      [1, 3],
      [2, 3],
      [1, 2, 3],
    ]);
  });
});
```

Listing 161: btPowerSet.js

```
/**
 * @param {*[]} originalSet - Original set of elements we're forming power-set of.
 * @param {*[][]} allSubsets - All subsets that have been formed so far.
 * @param {*[]} currentSubSet - Current subset that we're forming at the moment.
 * @param {number} startAt - The position of in original set we're starting to form current subset.
 * @return {*[][]} - All subsets of original set.
 */
function btPowerSetRecursive(originalSet, allSubsets = [[]], currentSubSet = [], startAt = 0) {
  // Let's iterate over originalSet elements that may be added to the subset
  // without having duplicates. The value of startAt prevents adding the duplicates.
  for (let position = startAt; position < originalSet.length; position += 1) {
    // Let's push current element to the subset
    currentSubSet.push(originalSet[position]);

    // Current subset is already valid so let's memorize it.
    // We do array destruction here to save the clone of the currentSubSet.
    // We need to save a clone since the original currentSubSet is going to be
    // mutated in further recursive calls.
    allSubsets.push([...currentSubSet]);

    // Let's try to generate all other subsets for the current subset.
    // We're increasing the position by one to avoid duplicates in subset.
    btPowerSetRecursive(originalSet, allSubsets, currentSubSet, position + 1);

    // BACKTRACK. Exclude last element from the subset and try the next valid one.
    currentSubSet.pop();
  }

  // Return all subsets of a set.
  return allSubsets;
}

/**
 * Find power-set of a set using BACKTRACKING approach.
 *
 * @param {*[]} originalSet
 * @return {*[][]}
 */
export default function btPowerSet(originalSet) {
  return btPowerSetRecursive(originalSet);
}
```

Listing 162: bwPowerSet.js

```
/**
 * Find power-set of a set using BITWISE approach.
 *
 * @param {*[]} originalSet
 * @return {*[][]}
 */
export default function bwPowerSet(originalSet) {
  const subSets = [];

  // We will have 2^n possible combinations (where n is a length of original set).
  // It is because for every element of original set we will decide whether to include
  // it or not (2 options for each set element).
  const numberOfCombinations = 2 ** originalSet.length;

  // Each number in binary representation in a range from 0 to 2^n does exactly what we need:
  // it shows by its bits (0 or 1) whether to include related element from the set or not.
  // For example, for the set {1, 2, 3} the binary number of 0b010 would mean that we need to
  // include only "2" to the current set.
  for (let combinationIndex = 0; combinationIndex < numberOfCombinations; combinationIndex += 1) {
    const subSet = [];

    for (let setElementIndex = 0; setElementIndex < originalSet.length; setElementIndex += 1) {
      // Decide whether we need to include current element into the subset or not.
      if (combinationIndex & (1 << setElementIndex)) {
        subSet.push(originalSet[setElementIndex]);
      }
    }

    // Add current subset to the list of all subsets.
    subSets.push(subSet);
  }

  return subSets;
}
```

Listing 163: shortestCommonSupersequence.test.js

```
import shortestCommonSupersequence from '../shortestCommonSupersequence';

describe('shortestCommonSupersequence', () => {
  it('should find shortest common supersequence of two sequences', () => {
    // LCS (longest common subsequence) is empty
    expect(shortestCommonSupersequence(
      ['A', 'B', 'C'],
      ['D', 'E', 'F'],
    )).toEqual(['A', 'B', 'C', 'D', 'E', 'F']);

    // LCS (longest common subsequence) is "EE"
    expect(shortestCommonSupersequence(
      ['G', 'E', 'E', 'K'],
      ['E', 'K', 'E'],
    )).toEqual(['G', 'E', 'K', 'E', 'K']);

    // LCS (longest common subsequence) is "GTAB"
    expect(shortestCommonSupersequence(
      ['A', 'G', 'G', 'T', 'A', 'B'],
      ['G', 'X', 'T', 'X', 'A', 'Y', 'B'],
    )).toEqual(['A', 'G', 'G', 'X', 'T', 'X', 'A', 'Y', 'B']);

    // LCS (longest common subsequence) is "BCBA".
    expect(shortestCommonSupersequence(
      ['A', 'B', 'C', 'B', 'D', 'A', 'B'],
      ['B', 'D', 'C', 'A', 'B', 'A'],
    )).toEqual(['A', 'B', 'D', 'C', 'A', 'B', 'D', 'A', 'B']);

    // LCS (longest common subsequence) is "BDABA".
    expect(shortestCommonSupersequence(
      ['B', 'D', 'C', 'A', 'B', 'A'],
      ['A', 'B', 'C', 'B', 'D', 'A', 'B', 'A', 'C'],
    )).toEqual(['A', 'B', 'C', 'B', 'D', 'C', 'A', 'B', 'A', 'C']);
  });
});
```

Listing 164: shortestCommonSupersequence.js

```
import longestCommonSubsequence from '../longest-common-subsequence/longestCommonSubsequence';

/**
 * @param {string[]} set1
 * @param {string[]} set2
 * @return {string[]}
 */

export default function shortestCommonSupersequence(set1, set2) {
  // Let's first find the longest common subsequence of two sets.
  const lcs = longestCommonSubsequence(set1, set2);

  // If LCS is empty then the shortest common supersequence would be just
  // concatenation of two sequences.
  if (lcs.length === 1 && lcs[0] === '') {
    return set1.concat(set2);
  }

  // Now let's add elements of set1 and set2 in order before/inside/after the LCS.
  let supersequence = [];

  let setIndex1 = 0;
  let setIndex2 = 0;
  let lcsIndex = 0;
  let setOnHold1 = false;
  let setOnHold2 = false;

  while (lcsIndex < lcs.length) {
    // Add elements of the first set to supersequence in correct order.
    if (setIndex1 < set1.length) {
      if (!setOnHold1 && set1[setIndex1] !== lcs[lcsIndex]) {
        supersequence.push(set1[setIndex1]);
        setIndex1 += 1;
      } else {
        setOnHold1 = true;
      }
    }

    // Add elements of the second set to supersequence in correct order.
    if (setIndex2 < set2.length) {
      if (!setOnHold2 && set2[setIndex2] !== lcs[lcsIndex]) {
        supersequence.push(set2[setIndex2]);
        setIndex2 += 1;
      } else {
        setOnHold2 = true;
      }
    }

    // Add LCS element to the supersequence in correct order.
    if (setOnHold1 && setOnHold2) {
      supersequence.push(lcs[lcsIndex]);
      lcsIndex += 1;
      setIndex1 += 1;
      setIndex2 += 1;
      setOnHold1 = false;
      setOnHold2 = false;
    }
  }

  // Attach set1 leftovers.
  if (setIndex1 < set1.length) {
    supersequence = supersequence.concat(set1.slice(setIndex1));
  }

  // Attach set2 leftovers.
  if (setIndex2 < set2.length) {
    supersequence = supersequence.concat(set2.slice(setIndex2));
  }

  return supersequence;
}
```

Listing 165: Sort.test.js

```
import Sort from '../Sort';

describe('Sort', () => {
  it('should throw an error when trying to call Sort.sort() method directly', () => {
    function doForbiddenSort() {
      const sorter = new Sort();
      sorter.sort();
    }

    expect(doForbiddenSort).toThrow();
  });
});
```

Listing 166: BubbleSort.test.js

```
import BubbleSort from '../BubbleSort';
import {
  equalArr,
  notSortedArr,
  reverseArr,
  sortedArr,
  SortTester,
} from '../../SortTester';

// Complexity constants.
const SORTED_ARRAY_VISITING_COUNT = 20;
const NOT_SORTED_ARRAY_VISITING_COUNT = 189;
const REVERSE_SORTED_ARRAY_VISITING_COUNT = 209;
const EQUAL_ARRAY_VISITING_COUNT = 20;

describe('BubbleSort', () => {
  it('should sort array', () => {
    SortTester.testSort(BubbleSort);
  });

  it('should sort array with custom comparator', () => {
    SortTester.testSortWithCustomComparator(BubbleSort);
  });

  it('should do stable sorting', () => {
    SortTester.testSortStability(BubbleSort);
  });

  it('should sort negative numbers', () => {
    SortTester.testNegativeNumbersSort(BubbleSort);
  });

  it('should visit EQUAL array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      BubbleSort,
      equalArr,
      EQUAL_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      BubbleSort,
      sortedArr,
      SORTED_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit NOT SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      BubbleSort,
      notSortedArr,
      NOT_SORTED_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit REVERSE SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      BubbleSort,
      reverseArr,
      REVERSE_SORTED_ARRAY_VISITING_COUNT,
    );
  });
});
```

Listing 167: BubbleSort.js

```
import Sort from '../Sort';

export default class BubbleSort extends Sort {
  sort(originalArray) {
    // Flag that holds info about whether the swap has occur or not.
    let swapped = false;
    // Clone original array to prevent its modification.
    const array = [...originalArray];

    for (let i = 1; i < array.length; i += 1) {
      swapped = false;

      // Call visiting callback.
      this.callbacks.visitingCallback(array[i]);

      for (let j = 0; j < array.length - i; j += 1) {
        // Call visiting callback.
        this.callbacks.visitingCallback(array[j]);

        // Swap elements if they are in wrong order.
        if (this.comparator.lessThan(array[j + 1], array[j])) {
          [array[j], array[j + 1]] = [array[j + 1], array[j]];

          // Register the swap.
          swapped = true;
        }
      }

      // If there were no swaps then array is already sorted and there is
      // no need to proceed.
      if (!swapped) {
        return array;
      }
    }

    return array;
  }
}
```

Listing 168: CountingSort.test.js

```
import CountingSort from '../CountingSort';
import {
  equalArr,
  notSortedArr,
  reverseArr,
  sortedArr,
  SortTester,
} from '../../SortTester';

// Complexity constants.
const SORTED_ARRAY_VISITING_COUNT = 60;
const NOT_SORTED_ARRAY_VISITING_COUNT = 60;
const REVERSE_SORTED_ARRAY_VISITING_COUNT = 60;
const EQUAL_ARRAY_VISITING_COUNT = 60;

describe('CountingSort', () => {
  it('should sort array', () => {
    SortTester.testSort(CountingSort);
  });

  it('should sort negative numbers', () => {
    SortTester.testNegativeNumbersSort(CountingSort);
  });

  it('should allow to use specify max/min integer value in array to make sorting faster', () => {
    const visitingCallback = jest.fn();
    const sorter = new CountingSort({ visitingCallback });

    // Detect biggest number in array in prior.
    const biggestElement = Math.max(...notSortedArr);

    // Detect smallest number in array in prior.
    const smallestElement = Math.min(...notSortedArr);

    const sortedArray = sorter.sort(notSortedArr, smallestElement, biggestElement);

    expect(sortedArray).toEqual(sortedArr);
    // Normally visitingCallback is being called 60 times but in this case
    // it should be called only 40 times.
    expect(visitingCallback).toHaveBeenCalledTimes(40);
  });

  it('should visit EQUAL array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      CountingSort,
      equalArr,
      EQUAL_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      CountingSort,
      sortedArr,
      SORTED_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit NOT SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      CountingSort,
      notSortedArr,
      NOT_SORTED_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit REVERSE SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      CountingSort,
      reverseArr,
      REVERSE_SORTED_ARRAY_VISITING_COUNT,
    );
  });
});
```

Listing 169: CountingSort.js

```
import Sort from '../Sort';

export default class CountingSort extends Sort {
  /**
   * @param {number[]} originalArray
   * @param {number} [smallestElement]
   * @param {number} [biggestElement]
   */
  sort(originalArray, smallestElement = undefined, biggestElement = undefined) {
    // Init biggest and smallest elements in array in order to build number bucket array later.
    let detectedSmallestElement = smallestElement || 0;
    let detectedBiggestElement = biggestElement || 0;

    if (smallestElement === undefined || biggestElement === undefined) {
      originalArray.forEach((element) => {
        // Visit element.
        this.callbacks.visitingCallback(element);

        // Detect biggest element.
        if (this.comparator.greaterThan(element, detectedBiggestElement)) {
          detectedBiggestElement = element;
        }

        // Detect smallest element.
        if (this.comparator.lessThan(element, detectedSmallestElement)) {
          detectedSmallestElement = element;
        }
      });
    }

    // Init buckets array.
    // This array will hold frequency of each number from originalArray.
    const buckets = Array(detectedBiggestElement - detectedSmallestElement + 1).fill(0);

    originalArray.forEach((element) => {
      // Visit element.
      this.callbacks.visitingCallback(element);

      buckets[element - detectedSmallestElement] += 1;
    });

    // Add previous frequencies to the current one for each number in bucket
    // to detect how many numbers less then current one should be standing to
    // the left of current one.
    for (let bucketIndex = 1; bucketIndex < buckets.length; bucketIndex += 1) {
      buckets[bucketIndex] += buckets[bucketIndex - 1];
    }

    // Now let's shift frequencies to the right so that they show correct numbers.
    // I.e. if we won't shift right than the value of buckets[5] will display how many
    // elements less than 5 should be placed to the left of 5 in sorted array
    // INCLUDING 5th. After shifting though this number will not include 5th anymore.
    buckets.pop();
    buckets.unshift(0);

    // Now let's assemble sorted array.
    const sortedArray = Array(originalArray.length).fill(null);
    for (let elementIndex = 0; elementIndex < originalArray.length; elementIndex += 1) {
      // Get the element that we want to put into correct sorted position.
      const element = originalArray[elementIndex];

      // Visit element.
      this.callbacks.visitingCallback(element);

      // Get correct position of this element in sorted array.
      const elementSortedPosition = buckets[element - detectedSmallestElement];

      // Put element into correct position in sorted array.
      sortedArray[elementSortedPosition] = element;

      // Increase position of current element in the bucket for future correct placements.
      buckets[element - detectedSmallestElement] += 1;
    }

    // Return sorted array.
    return sortedArray;
  }
}
```

Listing 170: HeapSort.test.js

```
import HeapSort from '../HeapSort';
import {
  equalArr,
  notSortedArr,
  reverseArr,
  sortedArr,
  SortTester,
} from '../../SortTester';

// Complexity constants.
// These numbers don't take into account up/dow heapifying of the heap.
// Thus these numbers are higher in reality.
const SORTED_ARRAY_VISITING_COUNT = 40;
const NOT_SORTED_ARRAY_VISITING_COUNT = 40;
const REVERSE_SORTED_ARRAY_VISITING_COUNT = 40;
const EQUAL_ARRAY_VISITING_COUNT = 40;

describe('HeapSort', () => {
  it('should sort array', () => {
    SortTester.testSort(HeapSort);
  });

  it('should sort array with custom comparator', () => {
    SortTester.testSortWithCustomComparator(HeapSort);
  });

  it('should sort negative numbers', () => {
    SortTester.testNegativeNumbersSort(HeapSort);
  });

  it('should visit EQUAL array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      HeapSort,
      equalArr,
      EQUAL_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      HeapSort,
      sortedArr,
      SORTED_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit NOT SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      HeapSort,
      notSortedArr,
      NOT_SORTED_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit REVERSE SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      HeapSort,
      reverseArr,
      REVERSE_SORTED_ARRAY_VISITING_COUNT,
    );
  });
});
```

Listing 171: HeapSort.js

```
import Sort from '../Sort';
import MinHeap from '../../../data-structures/heap/MinHeap';

export default class HeapSort extends Sort {
  sort(originalArray) {
    const sortedArray = [];
    const minHeap = new MinHeap(this.callbacks.compareCallback);

    // Insert all array elements to the heap.
    originalArray.forEach((element) => {
      // Call visiting callback.
      this.callbacks.visitingCallback(element);

      minHeap.add(element);
    });

    // Now we have min heap with minimal element always on top.
    // Let's poll that minimal element one by one and thus form the sorted array.
    while (!minHeap.isEmpty()) {
      const nextMinElement = minHeap.poll();

      // Call visiting callback.
      this.callbacks.visitingCallback(nextMinElement);

      sortedArray.push(nextMinElement);
    }

    return sortedArray;
  }
}
```

Listing 172: InsertionSort.test.js

```
import InsertionSort from '../InsertionSort';
import {
  equalArr,
  notSortedArr,
  reverseArr,
  sortedArr,
  SortTester,
} from '../../SortTester';

// Complexity constants.
const SORTED_ARRAY_VISITING_COUNT = 20;
const NOT_SORTED_ARRAY_VISITING_COUNT = 101;
const REVERSE_SORTED_ARRAY_VISITING_COUNT = 210;
const EQUAL_ARRAY_VISITING_COUNT = 20;

describe('InsertionSort', () => {
  it('should sort array', () => {
    SortTester.testSort(InsertionSort);
  });

  it('should sort array with custom comparator', () => {
    SortTester.testSortWithCustomComparator(InsertionSort);
  });

  it('should do stable sorting', () => {
    SortTester.testSortStability(InsertionSort);
  });

  it('should sort negative numbers', () => {
    SortTester.testNegativeNumbersSort(InsertionSort);
  });

  it('should visit EQUAL array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      InsertionSort,
      equalArr,
      EQUAL_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      InsertionSort,
      sortedArr,
      SORTED_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit NOT SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      InsertionSort,
      notSortedArr,
      NOT_SORTED_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit REVERSE SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      InsertionSort,
      reverseArr,
      REVERSE_SORTED_ARRAY_VISITING_COUNT,
    );
  });
});
```

Listing 173: InsertionSort.js

```
import Sort from '../Sort';

export default class InsertionSort extends Sort {
  sort(originalArray) {
    const array = [...originalArray];

    // Go through all array elements...
    for (let i = 0; i < array.length; i += 1) {
      let currentIndex = i;

      // Call visiting callback.
      this.callbacks.visitingCallback(array[i]);

      // Go and check if previous elements and greater then current one.
      // If this is the case then swap that elements.
      while (
        array[currentIndex - 1] !== undefined
        && this.comparator.lessThan(array[currentIndex], array[currentIndex - 1])
      ) {
        // Call visiting callback.
        this.callbacks.visitingCallback(array[currentIndex - 1]);

        // Swap the elements.
        const tmp = array[currentIndex - 1];
        array[currentIndex - 1] = array[currentIndex];
        array[currentIndex] = tmp;

        // Shift current index left.
        currentIndex -= 1;
      }
    }

    return array;
  }
}
```

Listing 174: MergeSort.test.js

```
import MergeSort from '../MergeSort';
import {
  equalArr,
  notSortedArr,
  reverseArr,
  sortedArr,
  SortTester,
} from '../../SortTester';

// Complexity constants.
const SORTED_ARRAY_VISITING_COUNT = 79;
const NOT_SORTED_ARRAY_VISITING_COUNT = 102;
const REVERSE_SORTED_ARRAY_VISITING_COUNT = 87;
const EQUAL_ARRAY_VISITING_COUNT = 79;

describe('MergeSort', () => {
  it('should sort array', () => {
    SortTester.testSort(MergeSort);
  });

  it('should sort array with custom comparator', () => {
    SortTester.testSortWithCustomComparator(MergeSort);
  });

  it('should do stable sorting', () => {
    SortTester.testSortStability(MergeSort);
  });

  it('should sort negative numbers', () => {
    SortTester.testNegativeNumbersSort(MergeSort);
  });

  it('should visit EQUAL array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      MergeSort,
      equalArr,
      EQUAL_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      MergeSort,
      sortedArr,
      SORTED_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit NOT SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      MergeSort,
      notSortedArr,
      NOT_SORTED_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit REVERSE SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      MergeSort,
      reverseArr,
      REVERSE_SORTED_ARRAY_VISITING_COUNT,
    );
  });
});
```

Listing 175: MergeSort.js

```
import Sort from '../Sort';

export default class MergeSort extends Sort {
  sort(originalArray) {
    // Call visiting callback.
    this.callbacks.visitingCallback(null);

    // If array is empty or consists of one element then return this array since it is sorted.
    if (originalArray.length <= 1) {
      return originalArray;
    }

    // Split array on two halves.
    const middleIndex = Math.floor(originalArray.length / 2);
    const leftArray = originalArray.slice(0, middleIndex);
    const rightArray = originalArray.slice(middleIndex, originalArray.length);

    // Sort two halves of split array
    const leftSortedArray = this.sort(leftArray);
    const rightSortedArray = this.sort(rightArray);

    // Merge two sorted arrays into one.
    return this.mergeSortedArrays(leftSortedArray, rightSortedArray);
  }

  mergeSortedArrays(leftArray, rightArray) {
    let sortedArray = [];

    // In case if arrays are not of size 1.
    while (leftArray.length && rightArray.length) {
      let minimumElement = null;

      // Find minimum element of two arrays.
      if (this.comparator.lessThanOrEqual(leftArray[0], rightArray[0])) {
        minimumElement = leftArray.shift();
      } else {
        minimumElement = rightArray.shift();
      }

      // Call visiting callback.
      this.callbacks.visitingCallback(minimumElement);

      // Push the minimum element of two arrays to the sorted array.
      sortedArray.push(minimumElement);
    }

    // If one of two array still have elements we need to just concatenate
    // this element to the sorted array since it is already sorted.
    if (leftArray.length) {
      sortedArray = sortedArray.concat(leftArray);
    }

    if (rightArray.length) {
      sortedArray = sortedArray.concat(rightArray);
    }

    return sortedArray;
  }
}
```

Listing 176: QuickSort.test.js

```
import QuickSort from '../QuickSort';
import {
  equalArr,
  notSortedArr,
  reverseArr,
  sortedArr,
  SortTester,
} from '../../SortTester';

// Complexity constants.
const SORTED_ARRAY_VISITING_COUNT = 190;
const NOT_SORTED_ARRAY_VISITING_COUNT = 62;
const REVERSE_SORTED_ARRAY_VISITING_COUNT = 190;
const EQUAL_ARRAY_VISITING_COUNT = 19;

describe('QuickSort', () => {
  it('should sort array', () => {
    SortTester.testSort(QuickSort);
  });

  it('should sort array with custom comparator', () => {
    SortTester.testSortWithCustomComparator(QuickSort);
  });

  it('should do stable sorting', () => {
    SortTester.testSortStability(QuickSort);
  });

  it('should sort negative numbers', () => {
    SortTester.testNegativeNumbersSort(QuickSort);
  });

  it('should visit EQUAL array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      QuickSort,
      equalArr,
      EQUAL_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      QuickSort,
      sortedArr,
      SORTED_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit NOT SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      QuickSort,
      notSortedArr,
      NOT_SORTED_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit REVERSE SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      QuickSort,
      reverseArr,
      REVERSE_SORTED_ARRAY_VISITING_COUNT,
    );
  });
});
```

Listing 177: QuickSortInPlace.test.js

```
import QuickSortInPlace from '../QuickSortInPlace';
import {
  equalArr,
  notSortedArr,
  reverseArr,
  sortedArr,
  SortTester,
} from '../../SortTester';

// Complexity constants.
const SORTED_ARRAY_VISITING_COUNT = 19;
const NOT_SORTED_ARRAY_VISITING_COUNT = 12;
const REVERSE_SORTED_ARRAY_VISITING_COUNT = 19;
const EQUAL_ARRAY_VISITING_COUNT = 19;

describe('QuickSortInPlace', () => {
  it('should sort array', () => {
    SortTester.testSort(QuickSortInPlace);
  });

  it('should sort array with custom comparator', () => {
    SortTester.testSortWithCustomComparator(QuickSortInPlace);
  });

  it('should sort negative numbers', () => {
    SortTester.testNegativeNumbersSort(QuickSortInPlace);
  });

  it('should visit EQUAL array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      QuickSortInPlace,
      equalArr,
      EQUAL_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      QuickSortInPlace,
      sortedArr,
      SORTED_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit NOT SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      QuickSortInPlace,
      notSortedArr,
      NOT_SORTED_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit REVERSE SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      QuickSortInPlace,
      reverseArr,
      REVERSE_SORTED_ARRAY_VISITING_COUNT,
    );
  });
});
```

Listing 178: QuickSort.js

```
import Sort from '../Sort';

export default class QuickSort extends Sort {
  /**
   * @param {*[]} originalArray
   * @return {*[]}
   */
  sort(originalArray) {
    // Clone original array to prevent it from modification.
    const array = [...originalArray];

    // If array has less than or equal to one elements then it is already sorted.
    if (array.length <= 1) {
      return array;
    }

    // Init left and right arrays.
    const leftArray = [];
    const rightArray = [];

    // Take the first element of array as a pivot.
    const pivotElement = array.shift();
    const centerArray = [pivotElement];

    // Split all array elements between left, center and right arrays.
    while (array.length) {
      const currentElement = array.shift();

      // Call visiting callback.
      this.callbacks.visitingCallback(currentElement);

      if (this.comparator.equal(currentElement, pivotElement)) {
        centerArray.push(currentElement);
      } else if (this.comparator.lessThan(currentElement, pivotElement)) {
        leftArray.push(currentElement);
      } else {
        rightArray.push(currentElement);
      }
    }

    // Sort left and right arrays.
    const leftArraySorted = this.sort(leftArray);
    const rightArraySorted = this.sort(rightArray);

    // Let's now join sorted left array with center array and with sorted right array.
    return leftArraySorted.concat(centerArray, rightArraySorted);
  }
}
```

Listing 179: QuickSortInPlace.js

```
import Sort from '../Sort';

export default class QuickSortInPlace extends Sort {
  /** Sorting in place avoids unnecessary use of additional memory, but modifies input array.
   *
   * This process is difficult to describe, but much clearer with a visualization:
   * @see: http://www.algomation.com/algorithm/quick-sort-visualization
   *
   * @param {*[]} originalArray - Not sorted array.
   * @param {number} inputLowIndex
   * @param {number} inputHighIndex
   * @param {boolean} recursiveCall
   * @return {*[]} - Sorted array.
   */
  sort(
    originalArray,
    inputLowIndex = 0,
    inputHighIndex = originalArray.length - 1,
    recursiveCall = false,
  ) {
    // Copies array on initial call, and then sorts in place.
    const array = recursiveCall ? originalArray : [...originalArray];

    /**
     * The partitionArray() operates on the subarray between lowIndex and highIndex, inclusive.
     * It arbitrarily chooses the last element in the subarray as the pivot.
     * Then, it partially sorts the subarray into elements than are less than the pivot,
     * and elements that are greater than or equal to the pivot.
     * Each time partitionArray() is executed, the pivot element is in its final sorted position.
     *
     * @param {number} lowIndex
     * @param {number} highIndex
     * @return {number}
     */
    const partitionArray = (lowIndex, highIndex) => {
      /**
       * Swaps two elements in array.
       * @param {number} leftIndex
       * @param {number} rightIndex
       */
      const swap = (leftIndex, rightIndex) => {
        const temp = array[leftIndex];
        array[leftIndex] = array[rightIndex];
        array[rightIndex] = temp;
      };

      const pivot = array[highIndex];
      // visitingCallback is used for time-complexity analysis.
      this.callbacks.visitingCallback(pivot);

      let partitionIndex = lowIndex;
      for (let currentIndex = lowIndex; currentIndex < highIndex; currentIndex += 1) {
        if (this.comparator.lessThan(array[currentIndex], pivot)) {
          swap(partitionIndex, currentIndex);
          partitionIndex += 1;
        }
      }

      // The element at the partitionIndex is guaranteed to be greater than or equal to pivot.
      // All elements to the left of partitionIndex are guaranteed to be less than pivot.
      // Swapping the pivot with the partitionIndex therefore places the pivot in its
      // final sorted position.
      swap(partitionIndex, highIndex);

      return partitionIndex;
    };

    // Base case is when low and high converge.
    if (inputLowIndex < inputHighIndex) {
      const partitionIndex = partitionArray(inputLowIndex, inputHighIndex);
      const RECURSIVE_CALL = true;
      this.sort(array, inputLowIndex, partitionIndex - 1, RECURSIVE_CALL);
      this.sort(array, partitionIndex + 1, inputHighIndex, RECURSIVE_CALL);
    }

    return array;
  }
}
```

Listing 180: RadixSort.test.js

```
import RadixSort from '../RadixSort';
import { SortTester } from '../../SortTester';

// Complexity constants.
const ARRAY_OF_STRINGS_VISIT_COUNT = 24;
const ARRAY_OF_INTEGERS_VISIT_COUNT = 77;
describe('RadixSort', () => {
  it('should sort array', () => {
    SortTester.testSort(RadixSort);
  });

  it('should visit array of strings n (number of strings) x m (length of longest element) times', () => {
    SortTester.testAlgorithmTimeComplexity(
      RadixSort,
      ['zzz', 'bb', 'a', 'rr', 'rrb', 'rrba'],
      ARRAY_OF_STRINGS_VISIT_COUNT,
    );
  });

  it('should visit array of integers n (number of elements) x m (length of longest integer) times', () => {
    SortTester.testAlgorithmTimeComplexity(
      RadixSort,
      [3, 1, 75, 32, 884, 523, 4343456, 232, 123, 656, 343],
      ARRAY_OF_INTEGERS_VISIT_COUNT,
    );
  });
});
```

Listing 181: RadixSort.js

```
import Sort from '../Sort';

// Using charCode (a = 97, b = 98, etc), we can map characters to buckets from 0 - 25
const BASE_CHAR_CODE = 97;
const NUMBER_OF_POSSIBLE_DIGITS = 10;
const ENGLISH_ALPHABET_LENGTH = 26;

export default class RadixSort extends Sort {
  /**
   * @param {*[]} originalArray
   * @return {*[]}
   */
  sort(originalArray) {
    // Assumes all elements of array are of the same type
    const isArrayOfNumbers = this.isArrayOfNumbers(originalArray);

    let sortedArray = [...originalArray];
    const numPasses = this.determineNumPasses(sortedArray);

    for (let currentIndex = 0; currentIndex < numPasses; currentIndex += 1) {
      const buckets = isArrayOfNumbers
        ? this.placeElementsInNumberBuckets(sortedArray, currentIndex)
        : this.placeElementsInCharacterBuckets(sortedArray, currentIndex, numPasses);

      // Flatten buckets into sortedArray, and repeat at next index
      sortedArray = buckets.reduce((acc, val) => {
        return [...acc, ...val];
      }, []);
    }

    return sortedArray;
  }

  /**
   * @param {*[]} array
   * @param {number} index
   * @return {*[]}
   */
  placeElementsInNumberBuckets(array, index) {
    // See below. These are used to determine which digit to use for bucket allocation
    const modded = 10 ** (index + 1);
    const divided = 10 ** index;
    const buckets = this.createBuckets(NUMBER_OF_POSSIBLE_DIGITS);

    array.forEach((element) => {
      this.callbacks.visitingCallback(element);
      if (element < divided) {
        buckets[0].push(element);
      } else {
        /**
         * Say we have element of 1,052 and are currently on index 1 (starting from 0). This means
         * we want to use '5' as the bucket. `modded` would be 10 ** (1 + 1), which
         * is 100. So we take 1,052 % 100 (52) and divide it by 10 (5.2) and floor it (5).
         */
        const currentDigit = Math.floor((element % modded) / divided);
        buckets[currentDigit].push(element);
      }
    });

    return buckets;
  }

  /**
   * @param {*[]} array
   * @param {number} index
   * @param {number} numPasses
   * @return {*[]}
   */
  placeElementsInCharacterBuckets(array, index, numPasses) {
    const buckets = this.createBuckets(ENGLISH_ALPHABET_LENGTH);

    array.forEach((element) => {
      this.callbacks.visitingCallback(element);
      const currentBucket = this.getCharCodeOfElementAtIndex(element, index, numPasses);
      buckets[currentBucket].push(element);
    });

    return buckets;
  }
```

```
  /**
   * @param {string} element
   * @param {number} index
   * @param {number} numPasses
   * @return {number}
   */
  getCharCodeOfElementAtIndex(element, index, numPasses) {
    // Place element in last bucket if not ready to organize
    if ((numPasses - index) > element.length) {
      return ENGLISH_ALPHABET_LENGTH - 1;
    }

    /**
     * If each character has been organized, use first character to determine bucket,
     * otherwise iterate backwards through element
     */
    const charPos = index > element.length - 1 ? 0 : element.length - index - 1;

    return element.toLowerCase().charCodeAt(charPos) - BASE_CHAR_CODE;
  }

  /**
   * Number of passes is determined by the length of the longest element in the array.
   * For integers, this log10(num), and for strings, this would be the length of the string.
   */
  determineNumPasses(array) {
    return this.getLengthOfLongestElement(array);
  }

  /**
   * @param {*[]} array
   * @return {number}
   */
  getLengthOfLongestElement(array) {
    if (this.isArrayOfNumbers(array)) {
      return Math.floor(Math.log10(Math.max(...array))) + 1;
    }

    return array.reduce((acc, val) => {
      return val.length > acc ? val.length : acc;
    }, -Infinity);
  }

  /**
   * @param {*[]} array
   * @return {boolean}
   */
  isArrayOfNumbers(array) {
    // Assumes all elements of array are of the same type
    return this.isNumber(array[0]);
  }

  /**
   * @param {number} numBuckets
   * @return {*[]}
   */
  createBuckets(numBuckets) {
    /**
     * Mapping buckets to an array instead of filling them with
     * an array prevents each bucket from containing a reference to the same array
     */
    return new Array(numBuckets).fill(null).map(() => []);
  }

  /**
   * @param {*} element
   * @return {boolean}
   */
  isNumber(element) {
    return Number.isInteger(element);
  }
}
```

Listing 182: SelectionSort.test.js

```
import SelectionSort from '../SelectionSort';
import {
  equalArr,
  notSortedArr,
  reverseArr,
  sortedArr,
  SortTester,
} from '../../SortTester';

// Complexity constants.
const SORTED_ARRAY_VISITING_COUNT = 209;
const NOT_SORTED_ARRAY_VISITING_COUNT = 209;
const REVERSE_SORTED_ARRAY_VISITING_COUNT = 209;
const EQUAL_ARRAY_VISITING_COUNT = 209;

describe('SelectionSort', () => {
  it('should sort array', () => {
    SortTester.testSort(SelectionSort);
  });

  it('should sort array with custom comparator', () => {
    SortTester.testSortWithCustomComparator(SelectionSort);
  });

  it('should sort negative numbers', () => {
    SortTester.testNegativeNumbersSort(SelectionSort);
  });

  it('should visit EQUAL array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      SelectionSort,
      equalArr,
      EQUAL_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      SelectionSort,
      sortedArr,
      SORTED_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit NOT SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      SelectionSort,
      notSortedArr,
      NOT_SORTED_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit REVERSE SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      SelectionSort,
      reverseArr,
      REVERSE_SORTED_ARRAY_VISITING_COUNT,
    );
  });
});
```

Listing 183: SelectionSort.js

```
import Sort from '../Sort';

export default class SelectionSort extends Sort {
  sort(originalArray) {
    // Clone original array to prevent its modification.
    const array = [...originalArray];

    for (let i = 0; i < array.length - 1; i += 1) {
      let minIndex = i;

      // Call visiting callback.
      this.callbacks.visitingCallback(array[i]);

      // Find minimum element in the rest of array.
      for (let j = i + 1; j < array.length; j += 1) {
        // Call visiting callback.
        this.callbacks.visitingCallback(array[j]);

        if (this.comparator.lessThan(array[j], array[minIndex])) {
          minIndex = j;
        }
      }

      // If new minimum element has been found then swap it with current i-th element.
      if (minIndex !== i) {
        [array[i], array[minIndex]] = [array[minIndex], array[i]];
      }
    }

    return array;
  }
}
```

Listing 184: ShellSort.test.js

```
import ShellSort from '../ShellSort';
import {
  equalArr,
  notSortedArr,
  reverseArr,
  sortedArr,
  SortTester,
} from '../../SortTester';

// Complexity constants.
const SORTED_ARRAY_VISITING_COUNT = 320;
const NOT_SORTED_ARRAY_VISITING_COUNT = 320;
const REVERSE_SORTED_ARRAY_VISITING_COUNT = 320;
const EQUAL_ARRAY_VISITING_COUNT = 320;

describe('ShellSort', () => {
  it('should sort array', () => {
    SortTester.testSort(ShellSort);
  });

  it('should sort array with custom comparator', () => {
    SortTester.testSortWithCustomComparator(ShellSort);
  });

  it('should sort negative numbers', () => {
    SortTester.testNegativeNumbersSort(ShellSort);
  });

  it('should visit EQUAL array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      ShellSort,
      equalArr,
      EQUAL_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      ShellSort,
      sortedArr,
      SORTED_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit NOT SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      ShellSort,
      notSortedArr,
      NOT_SORTED_ARRAY_VISITING_COUNT,
    );
  });

  it('should visit REVERSE SORTED array element specified number of times', () => {
    SortTester.testAlgorithmTimeComplexity(
      ShellSort,
      reverseArr,
      REVERSE_SORTED_ARRAY_VISITING_COUNT,
    );
  });
});
```

Listing 185: ShellSort.js

```
 import Sort from '../Sort';

export default class ShellSort extends Sort {
  sort(originalArray) {
    // Prevent original array from mutations.
    const array = [...originalArray];

    // Define a gap distance.
    let gap = Math.floor(array.length / 2);

    // Until gap is bigger then zero do elements comparisons and swaps.
    while (gap > 0) {
      // Go and compare all distant element pairs.
      for (let i = 0; i < (array.length - gap); i += 1) {
        let currentIndex = i;
        let gapShiftedIndex = i + gap;

        while (currentIndex >= 0) {
          // Call visiting callback.
          this.callbacks.visitingCallback(array[currentIndex]);

          // Compare and swap array elements if needed.
          if (this.comparator.lessThan(array[gapShiftedIndex], array[currentIndex])) {
            const tmp = array[currentIndex];
            array[currentIndex] = array[gapShiftedIndex];
            array[gapShiftedIndex] = tmp;
          }

          gapShiftedIndex = currentIndex;
          currentIndex -= gap;
        }
      }

      // Shrink the gap.
      gap = Math.floor(gap / 2);
    }

    // Return sorted copy of an original array.
    return array;
  }
}
```

Listing 186: Sort.js

```
import Comparator from '../../utils/comparator/Comparator';

/**
 * @typedef {Object} SorterCallbacks
 * @property {function(a: *, b: *)} compareCallback - If provided then all elements comparisons
 *  will be done through this callback.
 * @property {function(a: *)} visitingCallback - If provided it will be called each time the sorting
 *  function is visiting the next element.
 */

export default class Sort {
  constructor(originalCallbacks) {
    this.callbacks = Sort.initSortingCallbacks(originalCallbacks);
    this.comparator = new Comparator(this.callbacks.compareCallback);
  }

  /**
   * @param {SorterCallbacks} originalCallbacks
   * @returns {SorterCallbacks}
   */
  static initSortingCallbacks(originalCallbacks) {
    const callbacks = originalCallbacks || {};
    const stubCallback = () => {};

    callbacks.compareCallback = callbacks.compareCallback || undefined;
    callbacks.visitingCallback = callbacks.visitingCallback || stubCallback;

    return callbacks;
  }

  sort() {
    throw new Error('sort method must be implemented');
  }
}
```

Listing 187: SortTester.js

```
export const sortedArr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20];
export const reverseArr = [20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1];
export const notSortedArr = [15, 8, 5, 12, 10, 1, 16, 9, 11, 7, 20, 3, 2, 6, 17, 18, 4, 13, 14, 19];
export const equalArr = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1];
export const negativeArr = [-1, 0, 5, -10, 20, 13, -7, 3, 2, -3];
export const negativeArrSorted = [-10, -7, -3, -1, 0, 2, 3, 5, 13, 20];

export class SortTester {
  static testSort(SortingClass) {
    const sorter = new SortingClass();

    expect(sorter.sort([])).toEqual([]);
    expect(sorter.sort([1])).toEqual([1]);
    expect(sorter.sort([1, 2])).toEqual([1, 2]);
    expect(sorter.sort([2, 1])).toEqual([1, 2]);
    expect(sorter.sort([3, 4, 2, 1, 0, 0, 4, 3, 4, 2])).toEqual([0, 0, 1, 2, 2, 3, 3, 4, 4, 4]);
    expect(sorter.sort(sortedArr)).toEqual(sortedArr);
    expect(sorter.sort(reverseArr)).toEqual(sortedArr);
    expect(sorter.sort(notSortedArr)).toEqual(sortedArr);
    expect(sorter.sort(equalArr)).toEqual(equalArr);
  }

  static testNegativeNumbersSort(SortingClass) {
    const sorter = new SortingClass();
    expect(sorter.sort(negativeArr)).toEqual(negativeArrSorted);
  }

  static testSortWithCustomComparator(SortingClass) {
    const callbacks = {
      compareCallback: (a, b) => {
        if (a.length === b.length) {
          return 0;
        }
        return a.length < b.length ? -1 : 1;
      },
    };

    const sorter = new SortingClass(callbacks);

    expect(sorter.sort([''])).toEqual(['']);
    expect(sorter.sort(['a'])).toEqual(['a']);
    expect(sorter.sort(['aa', 'a'])).toEqual(['a', 'aa']);
    expect(sorter.sort(['aa', 'q', 'bbbb', 'ccc'])).toEqual(['q', 'aa', 'ccc', 'bbbb']);
    expect(sorter.sort(['aa', 'aa'])).toEqual(['aa', 'aa']);
  }

  static testSortStability(SortingClass) {
    const callbacks = {
      compareCallback: (a, b) => {
        if (a.length === b.length) {
          return 0;
        }
        return a.length < b.length ? -1 : 1;
      },
    };

    const sorter = new SortingClass(callbacks);

    expect(sorter.sort(['bb', 'aa', 'c'])).toEqual(['c', 'bb', 'aa']);
    expect(sorter.sort(['aa', 'q', 'a', 'bbbb', 'ccc'])).toEqual(['q', 'a', 'aa', 'ccc', 'bbbb']);
  }

  static testAlgorithmTimeComplexity(SortingClass, arrayToBeSorted, numberOfVisits) {
    const visitingCallback = jest.fn();
    const callbacks = { visitingCallback };
    const sorter = new SortingClass(callbacks);

    sorter.sort(arrayToBeSorted);

    expect(visitingCallback).toHaveBeenCalledTimes(numberOfVisits);
  }
}
```

Listing 188: hammingDistance.test.js

```
import hammingDistance from '../hammingDistance';

describe('hammingDistance', () => {
  it('should throw an error when trying to compare the strings of different lengths', () => {
    const compareStringsOfDifferentLength = () => {
      hammingDistance('a', 'aa');
    };

    expect(compareStringsOfDifferentLength).toThrowError();
  });

  it('should calculate difference between two strings', () => {
    expect(hammingDistance('a', 'a')).toBe(0);
    expect(hammingDistance('a', 'b')).toBe(1);
    expect(hammingDistance('abc', 'add')).toBe(2);
    expect(hammingDistance('karolin', 'kathrin')).toBe(3);
    expect(hammingDistance('karolin', 'kerstin')).toBe(3);
    expect(hammingDistance('1011101', '1001001')).toBe(2);
    expect(hammingDistance('2173896', '2233796')).toBe(3);
  });
});
```

Listing 189: hammingDistance.js

```
/**
 * @param {string} a
 * @param {string} b
 * @return {number}
 */
export default function hammingDistance(a, b) {
  if (a.length !== b.length) {
    throw new Error('Strings must be of the same length');
  }

  let distance = 0;

  for (let i = 0; i < a.length; i += 1) {
    if (a[i] !== b[i]) {
      distance += 1;
    }
  }

  return distance;
}
```

Listing 190: knuthMorrisPratt.test.js

```
import knuthMorrisPratt from '../knuthMorrisPratt';

describe('knuthMorrisPratt', () => {
  it('should find word position in given text', () => {
    expect(knuthMorrisPratt('', '')).toBe(0);
    expect(knuthMorrisPratt('a', '')).toBe(0);
    expect(knuthMorrisPratt('a', 'a')).toBe(0);
    expect(knuthMorrisPratt('abcbcglx', 'abca')).toBe(-1);
    expect(knuthMorrisPratt('abcbcglx', 'bcgl')).toBe(3);
    expect(knuthMorrisPratt('abcxabcdabxabcdabcdabcy', 'abcdabcy')).toBe(15);
    expect(knuthMorrisPratt('abcxabcdabxabcdabcdabcy', 'abcdabca')).toBe(-1);
    expect(knuthMorrisPratt('abcxabcdabxaabcdabcabcdabcdabcy', 'abcdabca')).toBe(12);
    expect(knuthMorrisPratt('abcxabcdabxaabaabaaaabcdabcdabcy', 'aabaabaaa')).toBe(11);
  });
});
```

Listing 191: knuthMorrisPratt.js

```
/**
 * @see https://www.youtube.com/watch?v=GTJr8OvyEVQ
 * @param {string} word
 * @return {number[]}
 */
function buildPatternTable(word) {
  const patternTable = [0];
  let prefixIndex = 0;
  let suffixIndex = 1;

  while (suffixIndex < word.length) {
    if (word[prefixIndex] === word[suffixIndex]) {
      patternTable[suffixIndex] = prefixIndex + 1;
      suffixIndex += 1;
      prefixIndex += 1;
    } else if (prefixIndex === 0) {
      patternTable[suffixIndex] = 0;
      suffixIndex += 1;
    } else {
      prefixIndex = patternTable[prefixIndex - 1];
    }
  }

  return patternTable;
}

/**
 * @param {string} text
 * @param {string} word
 * @return {number}
 */
export default function knuthMorrisPratt(text, word) {
  if (word.length === 0) {
    return 0;
  }

  let textIndex = 0;
  let wordIndex = 0;

  const patternTable = buildPatternTable(word);

  while (textIndex < text.length) {
    if (text[textIndex] === word[wordIndex]) {
      // We've found a match.
      if (wordIndex === word.length - 1) {
        return (textIndex - word.length) + 1;
      }
      wordIndex += 1;
      textIndex += 1;
    } else if (wordIndex > 0) {
      wordIndex = patternTable[wordIndex - 1];
    } else {
      wordIndex = 0;
      textIndex += 1;
    }
  }

  return -1;
}
```

Listing 192: levenshteinDistance.test.js

```
import levenshteinDistance from '../levenshteinDistance';

describe('levenshteinDistance', () => {
  it('should calculate edit distance between two strings', () => {
    expect(levenshteinDistance('', '')).toBe(0);
    expect(levenshteinDistance('a', '')).toBe(1);
    expect(levenshteinDistance('', 'a')).toBe(1);
    expect(levenshteinDistance('abc', '')).toBe(3);
    expect(levenshteinDistance('', 'abc')).toBe(3);

    // Should just add I to the beginning.
    expect(levenshteinDistance('islander', 'slander')).toBe(1);

    // Needs to substitute M by K, T by M and add an A to the end
    expect(levenshteinDistance('mart', 'karma')).toBe(3);

    // Substitute K by S, E by I and insert G at the end.
    expect(levenshteinDistance('kitten', 'sitting')).toBe(3);

    // Should add 4 letters FOOT at the beginning.
    expect(levenshteinDistance('ball', 'football')).toBe(4);

    // Should delete 4 letters FOOT at the beginning.
    expect(levenshteinDistance('football', 'foot')).toBe(4);

    // Needs to substitute the first 5 chars: INTEN by EXECU
    expect(levenshteinDistance('intention', 'execution')).toBe(5);
  });
});
```

Listing 193: levenshteinDistance.js

```
/**
* @param {string} a
* @param {string} b
* @return {number}
*/
export default function levenshteinDistance(a, b) {
  // Create empty edit distance matrix for all possible modifications of
  // substrings of a to substrings of b.
  const distanceMatrix = Array(b.length + 1).fill(null).map(() => Array(a.length + 1).fill(null));

  // Fill the first row of the matrix.
  // If this is first row then we're transforming empty string to a.
  // In this case the number of transformations equals to size of a substring.
  for (let i = 0; i <= a.length; i += 1) {
    distanceMatrix[0][i] = i;
  }

  // Fill the first column of the matrix.
  // If this is first column then we're transforming empty string to b.
  // In this case the number of transformations equals to size of b substring.
  for (let j = 0; j <= b.length; j += 1) {
    distanceMatrix[j][0] = j;
  }

  for (let j = 1; j <= b.length; j += 1) {
    for (let i = 1; i <= a.length; i += 1) {
      const indicator = a[i - 1] === b[j - 1] ? 0 : 1;
      distanceMatrix[j][i] = Math.min(
        distanceMatrix[j][i - 1] + 1, // deletion
        distanceMatrix[j - 1][i] + 1, // insertion
        distanceMatrix[j - 1][i - 1] + indicator, // substitution
      );
    }
  }

  return distanceMatrix[b.length][a.length];
}
```

Listing 194: longestCommonSubstring.test.js

```
import longestCommonSubstring from '../longestCommonSubstring';

describe('longestCommonSubstring', () => {
  it('should find longest common substring between two strings', () => {
    expect(longestCommonSubstring('', '')).toBe('');
    expect(longestCommonSubstring('ABC', '')).toBe('');
    expect(longestCommonSubstring('', 'ABC')).toBe('');
    expect(longestCommonSubstring('ABABC', 'BABCA')).toBe('BABC');
    expect(longestCommonSubstring('BABCA', 'ABCBA')).toBe('ABC');
    expect(longestCommonSubstring(
      'Algorithms and data structures implemented in JavaScript',
      'Here you may find Algorithms and data structures that are implemented in JavaScript',
    )).toBe('Algorithms and data structures ');
  });

  it('should handle unicode correctly', () => {
    expect(longestCommonSubstring('**ABC', '--ABC')).toBe('ABC');
    expect(longestCommonSubstring('**A', '--A')).toBe('');
    expect(longestCommonSubstring('AB', 'BGD')).toBe('B');
    expect(longestCommonSubstring('After test case', 'another_test')).toBe('test');
  });
});
```

Listing 195: longestCommonSubstring.js

```
/**
 * @param {string} string1
 * @param {string} string2
 * @return {string}
 */
export default function longestCommonSubstring(string1, string2) {
  // Convert strings to arrays to treat unicode symbols length correctly.
  // For example:
  // ''.length === 2
  // [...''].length === 1
  const s1 = [...string1];
  const s2 = [...string2];

  // Init the matrix of all substring lengths to use Dynamic Programming approach.
  const substringMatrix = Array(s2.length + 1).fill(null).map(() => {
    return Array(s1.length + 1).fill(null);
  });

  // Fill the first row and first column with zeros to provide initial values.
  for (let columnIndex = 0; columnIndex <= s1.length; columnIndex += 1) {
    substringMatrix[0][columnIndex] = 0;
  }

  for (let rowIndex = 0; rowIndex <= s2.length; rowIndex += 1) {
    substringMatrix[rowIndex][0] = 0;
  }

  // Build the matrix of all substring lengths to use Dynamic Programming approach.
  let longestSubstringLength = 0;
  let longestSubstringColumn = 0;
  let longestSubstringRow = 0;

  for (let rowIndex = 1; rowIndex <= s2.length; rowIndex += 1) {
    for (let columnIndex = 1; columnIndex <= s1.length; columnIndex += 1) {
      if (s1[columnIndex - 1] === s2[rowIndex - 1]) {
        substringMatrix[rowIndex][columnIndex] = substringMatrix[rowIndex - 1][columnIndex - 1] + 1;
      } else {
        substringMatrix[rowIndex][columnIndex] = 0;
      }

      // Try to find the biggest length of all common substring lengths
      // and to memorize its last character position (indices)
      if (substringMatrix[rowIndex][columnIndex] > longestSubstringLength) {
        longestSubstringLength = substringMatrix[rowIndex][columnIndex];
        longestSubstringColumn = columnIndex;
        longestSubstringRow = rowIndex;
      }
    }
  }

  if (longestSubstringLength === 0) {
    // Longest common substring has not been found.
    return '';
  }

  // Detect the longest substring from the matrix.
  let longestSubstring = '';

  while (substringMatrix[longestSubstringRow][longestSubstringColumn] > 0) {
    longestSubstring = s1[longestSubstringColumn - 1] + longestSubstring;
    longestSubstringRow -= 1;
    longestSubstringColumn -= 1;
  }

  return longestSubstring;
}
```

Listing 196: rabinKarp.test.js

```
import rabinKarp from '../rabinKarp';

describe('rabinKarp', () => {
  it('should find substring in a string', () => {
    expect(rabinKarp('', '')).toBe(0);
    expect(rabinKarp('a', '')).toBe(0);
    expect(rabinKarp('a', 'a')).toBe(0);
    expect(rabinKarp('ab', 'b')).toBe(1);
    expect(rabinKarp('abcbcglx', 'abca')).toBe(-1);
    expect(rabinKarp('abcbcglx', 'bcgl')).toBe(3);
    expect(rabinKarp('abcxabcdabxabcdabcdabcy', 'abcdabcy')).toBe(15);
    expect(rabinKarp('abcxabcdabxabcdabcdabcy', 'abcdabca')).toBe(-1);
    expect(rabinKarp('abcxabcdabxaabcdabcabcdabcdabcy', 'abcdabca')).toBe(12);
    expect(rabinKarp('abcxabcdabxaabaabaaaabcdabcdabcy', 'aabaabaaa')).toBe(11);
    expect(rabinKarp('^ !/\'#\'pp', ' !/\'#\'pp')).toBe(1);
  });

  it('should work with bigger texts', () => {
    const text = 'Lorem Ipsum is simply dummy text of the printing and '
    + 'typesetting industry. Lorem Ipsum has been the industry\'s standard '
    + 'dummy text ever since the 1500s, when an unknown printer took a '
    + 'galley of type and scrambled it to make a type specimen book. It '
    + 'has survived not only five centuries, but also the leap into '
    + 'electronic typesetting, remaining essentially unchanged. It was '
    + 'popularised in the 1960s with the release of Letraset sheets '
    + 'containing Lorem Ipsum passages, and more recently with desktop'
    + 'publishing software like Aldus PageMaker including versions of Lorem '
    + 'Ipsum.';

    expect(rabinKarp(text, 'Lorem')).toBe(0);
    expect(rabinKarp(text, 'versions')).toBe(549);
    expect(rabinKarp(text, 'versions of Lorem Ipsum.')).toBe(549);
    expect(rabinKarp(text, 'versions of Lorem Ipsum:')).toBe(-1);
    expect(rabinKarp(text, 'Lorem Ipsum passages, and more recently with')).toBe(446);
  });

  it('should work with UTF symbols', () => {
    expect(rabinKarp('a\u{ffff}', '\u{ffff}')).toBe(1);
    expect(rabinKarp('\u0000\u0000', '\u0000')).toBe(1);
    // @TODO: Provide Unicode support.
    // expect(rabinKarp('a\u{20000}', '\u{20000}')).toBe(1);
  });
});
```

Listing 197: rabinKarp.js

```
import PolynomialHash from '../../cryptography/polynomial-hash/PolynomialHash';

/**
 * @param {string} text - Text that may contain the searchable word.
 * @param {string} word - Word that is being searched in text.
 * @return {number} - Position of the word in text.
 */
export default function rabinKarp(text, word) {
  const hasher = new PolynomialHash();

  // Calculate word hash that we will use for comparison with other substring hashes.
  const wordHash = hasher.hash(word);

  let prevFrame = null;
  let currentFrameHash = null;

  // Go through all substring of the text that may match.
  for (let charIndex = 0; charIndex <= (text.length - word.length); charIndex += 1) {
    const currentFrame = text.substring(charIndex, charIndex + word.length);

    // Calculate the hash of current substring.
    if (currentFrameHash === null) {
      currentFrameHash = hasher.hash(currentFrame);
    } else {
      currentFrameHash = hasher.roll(currentFrameHash, prevFrame, currentFrame);
    }

    prevFrame = currentFrame;

    // Compare the hash of current substring and seeking string.
    // In case if hashes match let's make sure that substrings are equal.
    // In case of hash collision the strings may not be equal.
    if (
      wordHash === currentFrameHash
      && text.substr(charIndex, word.length) === word
    ) {
      return charIndex;
    }
  }

  return -1;
}
```

Listing 198: regularExpressionMatching.test.js

```
import regularExpressionMatching from '../regularExpressionMatching';

describe('regularExpressionMatching', () => {
  it('should match regular expressions in a string', () => {
    expect(regularExpressionMatching('', '')).toBe(true);
    expect(regularExpressionMatching('a', 'a')).toBe(true);
    expect(regularExpressionMatching('aa', 'aa')).toBe(true);
    expect(regularExpressionMatching('aab', 'aab')).toBe(true);
    expect(regularExpressionMatching('aab', 'aa.')).toBe(true);
    expect(regularExpressionMatching('aab', '.a.')).toBe(true);
    expect(regularExpressionMatching('aab', '...')).toBe(true);
    expect(regularExpressionMatching('a', 'a*')).toBe(true);
    expect(regularExpressionMatching('aaa', 'a*')).toBe(true);
    expect(regularExpressionMatching('aaab', 'a*b')).toBe(true);
    expect(regularExpressionMatching('aaabb', 'a*b*')).toBe(true);
    expect(regularExpressionMatching('aaabb', 'a*b*c*')).toBe(true);
    expect(regularExpressionMatching('', 'a*')).toBe(true);
    expect(regularExpressionMatching('xaabyc', 'xa*b.c')).toBe(true);
    expect(regularExpressionMatching('aab', 'c*a*b*')).toBe(true);
    expect(regularExpressionMatching('mississippi', 'mis*is*.p*.')).toBe(true);
    expect(regularExpressionMatching('ab', '.*')).toBe(true);

    expect(regularExpressionMatching('', 'a')).toBe(false);
    expect(regularExpressionMatching('a', '')).toBe(false);
    expect(regularExpressionMatching('aab', 'aa')).toBe(false);
    expect(regularExpressionMatching('aab', 'baa')).toBe(false);
    expect(regularExpressionMatching('aabc', '...')).toBe(false);
    expect(regularExpressionMatching('aaabbdd', 'a*b*c*')).toBe(false);
    expect(regularExpressionMatching('mississippi', 'mis*is*p*.')).toBe(false);
    expect(regularExpressionMatching('ab', 'a*')).toBe(false);
    expect(regularExpressionMatching('abba', 'a*b*.c')).toBe(false);
    expect(regularExpressionMatching('abba', '.*c')).toBe(false);
  });
});
```

Listing 199: regularExpressionMatching.js

```
const ZERO_OR_MORE_CHARS = '*';
const ANY_CHAR = '.';

/**
 * Dynamic programming approach.
 *
 * @param {string} string
 * @param {string} pattern
 * @return {boolean}
 */
export default function regularExpressionMatching(string, pattern) {
  /*
   * Let's initiate dynamic programming matrix for this string and pattern.
   * We will have pattern characters on top (as columns) and string characters
   * will be placed to the left of the table (as rows).
   *
   * Example:
   *
   *     a * b . b
   *   - - - - - -
   * a - - - - - -
   * a - - - - - -
   * b - - - - - -
   * y - - - - - -
   * b - - - - - -
   */
  const matchMatrix = Array(string.length + 1).fill(null).map(() => {
    return Array(pattern.length + 1).fill(null);
  });

  // Let's fill the top-left cell with true. This would mean that empty
  // string '' matches to empty pattern ''.
  matchMatrix[0][0] = true;

  // Let's fill the first row of the matrix with false. That would mean that
  // empty string can't match any non-empty pattern.
  //
  // Example:
  // string: ''
  // pattern: 'a.z'
  //
  // The one exception here is patterns like a*b* that matches the empty string.
  for (let columnIndex = 1; columnIndex <= pattern.length; columnIndex += 1) {
    const patternIndex = columnIndex - 1;

    if (pattern[patternIndex] === ZERO_OR_MORE_CHARS) {
      matchMatrix[0][columnIndex] = matchMatrix[0][columnIndex - 2];
    } else {
      matchMatrix[0][columnIndex] = false;
    }
  }

  // Let's fill the first column with false. That would mean that empty pattern
  // can't match any non-empty string.
  //
  // Example:
  // string: 'ab'
  // pattern: ''
  for (let rowIndex = 1; rowIndex <= string.length; rowIndex += 1) {
    matchMatrix[rowIndex][0] = false;
  }

  // Not let's go through every letter of the pattern and every letter of
  // the string and compare them one by one.
  for (let rowIndex = 1; rowIndex <= string.length; rowIndex += 1) {
    for (let columnIndex = 1; columnIndex <= pattern.length; columnIndex += 1) {
      // Take into account that fact that matrix contain one extra column and row.
      const stringIndex = rowIndex - 1;
      const patternIndex = columnIndex - 1;

      if (pattern[patternIndex] === ZERO_OR_MORE_CHARS) {
        /*
         * In case if current pattern character is special '*' character we have
         * two options:
         *
         * 1. Since * char allows it previous char to not be presented in a string we
         * need to check if string matches the pattern without '*' char and without the
         * char that goes before '*'. That would mean to go two positions left on the
         * same row.
```

```
         *
         * 2. Since * char allows it previous char to be presented in a string many times we
         * need to check if char before * is the same as current string char. If they are the
         * same that would mean that current string matches the current pattern in case if
         * the string WITHOUT current char matches the same pattern. This would mean to go
         * one position up in the same row.
         */
        if (matchMatrix[rowIndex][columnIndex - 2] === true) {
          matchMatrix[rowIndex][columnIndex] = true;
        } else if (
          (
            pattern[patternIndex - 1] === string[stringIndex]
            || pattern[patternIndex - 1] === ANY_CHAR
          )
          && matchMatrix[rowIndex - 1][columnIndex] === true
        ) {
          matchMatrix[rowIndex][columnIndex] = true;
        } else {
          matchMatrix[rowIndex][columnIndex] = false;
        }
      } else if (
        pattern[patternIndex] === string[stringIndex]
        || pattern[patternIndex] === ANY_CHAR
      ) {
        /*
         * In case if current pattern char is the same as current string char
         * or it may be any character (in case if pattern contains '.' char)
         * we need to check if there was a match for the pattern and for the
         * string by WITHOUT current char. This would mean that we may copy
         * left-top diagonal value.
         *
         * Example:
         *
         *   a b
         * a 1 -
         * b - 1
         */
        matchMatrix[rowIndex][columnIndex] = matchMatrix[rowIndex - 1][columnIndex - 1];
      } else {
        /*
         * In case if pattern char and string char are different we may
         * treat this case as "no-match".
         *
         * Example:
         *
         *   a b
         * a - -
         * c - 0
         */
        matchMatrix[rowIndex][columnIndex] = false;
      }
    }
  }

  return matchMatrix[string.length][pattern.length];
}
```

Listing 200: zAlgorithm.test.js

```
import zAlgorithm from '../zAlgorithm';

describe('zAlgorithm', () => {
  it('should find word positions in given text', () => {
    expect(zAlgorithm('abcbcglx', 'abca')).toEqual([]);
    expect(zAlgorithm('abca', 'abca')).toEqual([0]);
    expect(zAlgorithm('abca', 'abcadfd')).toEqual([]);
    expect(zAlgorithm('abcbcglabcx', 'abc')).toEqual([0, 7]);
    expect(zAlgorithm('abcbcglx', 'bcgl')).toEqual([3]);
    expect(zAlgorithm('abcbcglx', 'cglx')).toEqual([4]);
    expect(zAlgorithm('abcxabcdabxabcdabcdabcy', 'abcdabcy')).toEqual([15]);
    expect(zAlgorithm('abcxabcdabxabcdabcdabcy', 'abcdabca')).toEqual([]);
    expect(zAlgorithm('abcxabcdabxaabcdabcabcdabcdabcy', 'abcdabca')).toEqual([12]);
    expect(zAlgorithm('abcxabcdabxaabaabaaaabcdabcdabcy', 'aabaabaaa')).toEqual([11]);
  });
});
```

Listing 201: zAlgorithm.js

```
// The string separator that is being used for "word" and "text" concatenation.
const SEPARATOR = '$';

/**
 * @param {string} zString
 * @return {number[]}
 */
function buildZArray(zString) {
  // Initiate zArray and fill it with zeros.
  const zArray = new Array(zString.length).fill(null).map(() => 0);

  // Z box boundaries.
  let zBoxLeftIndex = 0;
  let zBoxRightIndex = 0;

  // Position of current zBox character that is also a position of
  // the same character in prefix.
  // For example:
  // Z string: ab$xxabxx
  // Indices:  012345678
  // Prefix:   ab.......
  // Z box:    .....ab..
  // Z box shift for 'a' would be 0 (0-position in prefix and 0-position in Z box)
  // Z box shift for 'b' would be 1 (1-position in prefix and 1-position in Z box)
  let zBoxShift = 0;

  // Go through all characters of the zString.
  for (let charIndex = 1; charIndex < zString.length; charIndex += 1) {
    if (charIndex > zBoxRightIndex) {
      // We're OUTSIDE of Z box. In other words this is a case when we're
      // starting from Z box of size 1.

      // In this case let's make current character to be a Z box of length 1.
      zBoxLeftIndex = charIndex;
      zBoxRightIndex = charIndex;

      // Now let's go and check current and the following characters to see if
      // they are the same as a prefix. By doing this we will also expand our
      // Z box. For example if starting from current position we will find 3
      // more characters that are equal to the ones in the prefix we will expand
      // right Z box boundary by 3.
      while (
        zBoxRightIndex < zString.length
        && zString[zBoxRightIndex - zBoxLeftIndex] === zString[zBoxRightIndex]
      ) {
        // Expanding Z box right boundary.
        zBoxRightIndex += 1;
      }

      // Now we may calculate how many characters starting from current position
      // are are the same as the prefix. We may calculate it by difference between
      // right and left Z box boundaries.
      zArray[charIndex] = zBoxRightIndex - zBoxLeftIndex;

      // Move right Z box boundary left by one position just because we've used
      // [zBoxRightIndex - zBoxLeftIndex] index calculation above.
      zBoxRightIndex -= 1;
    } else {
      // We're INSIDE of Z box.

      // Calculate corresponding Z box shift. Because we want to copy the values
      // from zArray that have been calculated before.
      zBoxShift = charIndex - zBoxLeftIndex;

      // Check if the value that has been already calculated before
      // leaves us inside of Z box or it goes beyond the checkbox
      // right boundary.
      if (zArray[zBoxShift] < (zBoxRightIndex - charIndex) + 1) {
        // If calculated value don't force us to go outside Z box
        // then we're safe and we may simply use previously calculated value.
        zArray[charIndex] = zArray[zBoxShift];
      } else {
        // In case if previously calculated values forces us to go outside of Z box
        // we can't safely copy previously calculated zArray value. It is because
        // we are sure that there is no further prefix matches outside of Z box.
        // Thus such values must be re-calculated and reduced to certain point.

        // To do so we need to shift left boundary of Z box to current position.
        zBoxLeftIndex = charIndex;
```

```
        // And start comparing characters one by one as we normally do for the case
        // when we are outside of checkbox.
        while (
          zBoxRightIndex < zString.length
          && zString[zBoxRightIndex - zBoxLeftIndex] === zString[zBoxRightIndex]
        ) {
          zBoxRightIndex += 1;
        }

        zArray[charIndex] = zBoxRightIndex - zBoxLeftIndex;

        zBoxRightIndex -= 1;
      }
    }
  }

  // Return generated zArray.
  return zArray;
}

/**
 * @param {string} text
 * @param {string} word
 * @return {number[]}
 */
export default function zAlgorithm(text, word) {
  // The list of word's positions in text. Word may be found in the same text
  // in several different positions. Thus it is an array.
  const wordPositions = [];

  // Concatenate word and string. Word will be a prefix to a string.
  const zString = `${word}${SEPARATOR}${text}`;

  // Generate Z-array for concatenated string.
  const zArray = buildZArray(zString);

  // Based on Z-array properties each cell will tell us the length of the match between
  // the string prefix and current sub-text. Thus we're may find all positions in zArray
  // with the number that equals to the length of the word (zString prefix) and based on
  // that positions we'll be able to calculate word positions in text.
  for (let charIndex = 1; charIndex < zArray.length; charIndex += 1) {
    if (zArray[charIndex] === word.length) {
      // Since we did concatenation to form zString we need to subtract prefix
      // and separator lengths.
      const wordPosition = charIndex - word.length - SEPARATOR.length;
      wordPositions.push(wordPosition);
    }
  }

  // Return the list of word positions.
  return wordPositions;
}
```

Listing 202: breadthFirstSearch.test.js

```
 import BinaryTreeNode from '../../../../data-structures/tree/BinaryTreeNode';
import breadthFirstSearch from '../breadthFirstSearch';

describe('breadthFirstSearch', () => {
  it('should perform DFS operation on tree', () => {
    const nodeA = new BinaryTreeNode('A');
    const nodeB = new BinaryTreeNode('B');
    const nodeC = new BinaryTreeNode('C');
    const nodeD = new BinaryTreeNode('D');
    const nodeE = new BinaryTreeNode('E');
    const nodeF = new BinaryTreeNode('F');
    const nodeG = new BinaryTreeNode('G');

    nodeA.setLeft(nodeB).setRight(nodeC);
    nodeB.setLeft(nodeD).setRight(nodeE);
    nodeC.setLeft(nodeF).setRight(nodeG);

    // In-order traversing.
    expect(nodeA.toString()).toBe('D,B,E,A,F,C,G');

    const enterNodeCallback = jest.fn();
    const leaveNodeCallback = jest.fn();

    // Traverse tree without callbacks first to check default ones.
    breadthFirstSearch(nodeA);

    // Traverse tree with callbacks.
    breadthFirstSearch(nodeA, {
      enterNode: enterNodeCallback,
      leaveNode: leaveNodeCallback,
    });

    expect(enterNodeCallback).toHaveBeenCalledTimes(7);
    expect(leaveNodeCallback).toHaveBeenCalledTimes(7);

    // Check node entering.
    expect(enterNodeCallback.mock.calls[0][0].value).toEqual('A');
    expect(enterNodeCallback.mock.calls[1][0].value).toEqual('B');
    expect(enterNodeCallback.mock.calls[2][0].value).toEqual('C');
    expect(enterNodeCallback.mock.calls[3][0].value).toEqual('D');
    expect(enterNodeCallback.mock.calls[4][0].value).toEqual('E');
    expect(enterNodeCallback.mock.calls[5][0].value).toEqual('F');
    expect(enterNodeCallback.mock.calls[6][0].value).toEqual('G');

    // Check node leaving.
    expect(leaveNodeCallback.mock.calls[0][0].value).toEqual('A');
    expect(leaveNodeCallback.mock.calls[1][0].value).toEqual('B');
    expect(leaveNodeCallback.mock.calls[2][0].value).toEqual('C');
    expect(leaveNodeCallback.mock.calls[3][0].value).toEqual('D');
    expect(leaveNodeCallback.mock.calls[4][0].value).toEqual('E');
    expect(leaveNodeCallback.mock.calls[5][0].value).toEqual('F');
    expect(leaveNodeCallback.mock.calls[6][0].value).toEqual('G');
  });

  it('allow users to redefine node visiting logic', () => {
    const nodeA = new BinaryTreeNode('A');
    const nodeB = new BinaryTreeNode('B');
    const nodeC = new BinaryTreeNode('C');
    const nodeD = new BinaryTreeNode('D');
    const nodeE = new BinaryTreeNode('E');
    const nodeF = new BinaryTreeNode('F');
    const nodeG = new BinaryTreeNode('G');

    nodeA.setLeft(nodeB).setRight(nodeC);
    nodeB.setLeft(nodeD).setRight(nodeE);
    nodeC.setLeft(nodeF).setRight(nodeG);

    // In-order traversing.
    expect(nodeA.toString()).toBe('D,B,E,A,F,C,G');

    const enterNodeCallback = jest.fn();
    const leaveNodeCallback = jest.fn();

    // Traverse tree without callbacks first to check default ones.
    breadthFirstSearch(nodeA);

    // Traverse tree with callbacks.
    breadthFirstSearch(nodeA, {
      allowTraversal: (node, child) => {
```

```
        // Forbid traversing left half of the tree.
        return child.value !== 'B';
      },
      enterNode: enterNodeCallback,
      leaveNode: leaveNodeCallback,
    });

    expect(enterNodeCallback).toHaveBeenCalledTimes(4);
    expect(leaveNodeCallback).toHaveBeenCalledTimes(4);

    // Check node entering.
    expect(enterNodeCallback.mock.calls[0][0].value).toEqual('A');
    expect(enterNodeCallback.mock.calls[1][0].value).toEqual('C');
    expect(enterNodeCallback.mock.calls[2][0].value).toEqual('F');
    expect(enterNodeCallback.mock.calls[3][0].value).toEqual('G');

    // Check node leaving.
    expect(leaveNodeCallback.mock.calls[0][0].value).toEqual('A');
    expect(leaveNodeCallback.mock.calls[1][0].value).toEqual('C');
    expect(leaveNodeCallback.mock.calls[2][0].value).toEqual('F');
    expect(leaveNodeCallback.mock.calls[3][0].value).toEqual('G');
  });
});
```

Listing 203: breadthFirstSearch.js

```
import Queue from '../../../data-structures/queue/Queue';

/**
 * @typedef {Object} Callbacks
 * @property {function(node: BinaryTreeNode, child: BinaryTreeNode): boolean} allowTraversal -
 *   Determines whether DFS should traverse from the node to its child.
 * @property {function(node: BinaryTreeNode)} enterNode - Called when DFS enters the node.
 * @property {function(node: BinaryTreeNode)} leaveNode - Called when DFS leaves the node.
 */

/**
 * @param {Callbacks} [callbacks]
 * @returns {Callbacks}
 */
function initCallbacks(callbacks = {}) {
  const initiatedCallback = callbacks;

  const stubCallback = () => {};
  const defaultAllowTraversal = () => true;

  initiatedCallback.allowTraversal = callbacks.allowTraversal || defaultAllowTraversal;
  initiatedCallback.enterNode = callbacks.enterNode || stubCallback;
  initiatedCallback.leaveNode = callbacks.leaveNode || stubCallback;

  return initiatedCallback;
}

/**
 * @param {BinaryTreeNode} rootNode
 * @param {Callbacks} [originalCallbacks]
 */
export default function breadthFirstSearch(rootNode, originalCallbacks) {
  const callbacks = initCallbacks(originalCallbacks);
  const nodeQueue = new Queue();

  // Do initial queue setup.
  nodeQueue.enqueue(rootNode);

  while (!nodeQueue.isEmpty()) {
    const currentNode = nodeQueue.dequeue();

    callbacks.enterNode(currentNode);

    // Add all children to the queue for future traversals.

    // Traverse left branch.
    if (currentNode.left && callbacks.allowTraversal(currentNode, currentNode.left)) {
      nodeQueue.enqueue(currentNode.left);
    }

    // Traverse right branch.
    if (currentNode.right && callbacks.allowTraversal(currentNode, currentNode.right)) {
      nodeQueue.enqueue(currentNode.right);
    }

    callbacks.leaveNode(currentNode);
  }
}
```

Listing 204: depthFirstSearch.test.js

```
import BinaryTreeNode from '../../../../data-structures/tree/BinaryTreeNode';
import depthFirstSearch from '../depthFirstSearch';

describe('depthFirstSearch', () => {
  it('should perform DFS operation on tree', () => {
    const nodeA = new BinaryTreeNode('A');
    const nodeB = new BinaryTreeNode('B');
    const nodeC = new BinaryTreeNode('C');
    const nodeD = new BinaryTreeNode('D');
    const nodeE = new BinaryTreeNode('E');
    const nodeF = new BinaryTreeNode('F');
    const nodeG = new BinaryTreeNode('G');

    nodeA.setLeft(nodeB).setRight(nodeC);
    nodeB.setLeft(nodeD).setRight(nodeE);
    nodeC.setLeft(nodeF).setRight(nodeG);

    // In-order traversing.
    expect(nodeA.toString()).toBe('D,B,E,A,F,C,G');

    const enterNodeCallback = jest.fn();
    const leaveNodeCallback = jest.fn();

    // Traverse tree without callbacks first to check default ones.
    depthFirstSearch(nodeA);

    // Traverse tree with callbacks.
    depthFirstSearch(nodeA, {
      enterNode: enterNodeCallback,
      leaveNode: leaveNodeCallback,
    });

    expect(enterNodeCallback).toHaveBeenCalledTimes(7);
    expect(leaveNodeCallback).toHaveBeenCalledTimes(7);

    // Check node entering.
    expect(enterNodeCallback.mock.calls[0][0].value).toEqual('A');
    expect(enterNodeCallback.mock.calls[1][0].value).toEqual('B');
    expect(enterNodeCallback.mock.calls[2][0].value).toEqual('D');
    expect(enterNodeCallback.mock.calls[3][0].value).toEqual('E');
    expect(enterNodeCallback.mock.calls[4][0].value).toEqual('C');
    expect(enterNodeCallback.mock.calls[5][0].value).toEqual('F');
    expect(enterNodeCallback.mock.calls[6][0].value).toEqual('G');

    // Check node leaving.
    expect(leaveNodeCallback.mock.calls[0][0].value).toEqual('D');
    expect(leaveNodeCallback.mock.calls[1][0].value).toEqual('E');
    expect(leaveNodeCallback.mock.calls[2][0].value).toEqual('B');
    expect(leaveNodeCallback.mock.calls[3][0].value).toEqual('F');
    expect(leaveNodeCallback.mock.calls[4][0].value).toEqual('G');
    expect(leaveNodeCallback.mock.calls[5][0].value).toEqual('C');
    expect(leaveNodeCallback.mock.calls[6][0].value).toEqual('A');
  });

  it('allow users to redefine node visiting logic', () => {
    const nodeA = new BinaryTreeNode('A');
    const nodeB = new BinaryTreeNode('B');
    const nodeC = new BinaryTreeNode('C');
    const nodeD = new BinaryTreeNode('D');
    const nodeE = new BinaryTreeNode('E');
    const nodeF = new BinaryTreeNode('F');
    const nodeG = new BinaryTreeNode('G');

    nodeA.setLeft(nodeB).setRight(nodeC);
    nodeB.setLeft(nodeD).setRight(nodeE);
    nodeC.setLeft(nodeF).setRight(nodeG);

    // In-order traversing.
    expect(nodeA.toString()).toBe('D,B,E,A,F,C,G');

    const enterNodeCallback = jest.fn();
    const leaveNodeCallback = jest.fn();

    // Traverse tree without callbacks first to check default ones.
    depthFirstSearch(nodeA);

    // Traverse tree with callbacks.
    depthFirstSearch(nodeA, {
      allowTraversal: (node, child) => {
```

```
        // Forbid traversing left part of the tree.
        return child.value !== 'B';
      },
      enterNode: enterNodeCallback,
      leaveNode: leaveNodeCallback,
    });

    expect(enterNodeCallback).toHaveBeenCalledTimes(4);
    expect(leaveNodeCallback).toHaveBeenCalledTimes(4);

    // Check node entering.
    expect(enterNodeCallback.mock.calls[0][0].value).toEqual('A');
    expect(enterNodeCallback.mock.calls[1][0].value).toEqual('C');
    expect(enterNodeCallback.mock.calls[2][0].value).toEqual('F');
    expect(enterNodeCallback.mock.calls[3][0].value).toEqual('G');

    // Check node leaving.
    expect(leaveNodeCallback.mock.calls[0][0].value).toEqual('F');
    expect(leaveNodeCallback.mock.calls[1][0].value).toEqual('G');
    expect(leaveNodeCallback.mock.calls[2][0].value).toEqual('C');
    expect(leaveNodeCallback.mock.calls[3][0].value).toEqual('A');
  });
});
```

Listing 205: depthFirstSearch.js

```
/**
 * @typedef {Object} TraversalCallbacks
 *
 * @property {function(node: BinaryTreeNode, child: BinaryTreeNode): boolean} allowTraversal
 * - Determines whether DFS should traverse from the node to its child.
 *
 * @property {function(node: BinaryTreeNode)} enterNode - Called when DFS enters the node.
 *
 * @property {function(node: BinaryTreeNode)} leaveNode - Called when DFS leaves the node.
 */

/**
 * Extend missing traversal callbacks with default callbacks.
 *
 * @param {TraversalCallbacks} [callbacks] - The object that contains traversal callbacks.
 * @returns {TraversalCallbacks} - Traversal callbacks extended with defaults callbacks.
 */
function initCallbacks(callbacks = {}) {
  // Init empty callbacks object.
  const initiatedCallbacks = {};

  // Empty callback that we will use in case if user didn't provide real callback function.
  const stubCallback = () => {};
  // By default we will allow traversal of every node
  // in case if user didn't provide a callback for that.
  const defaultAllowTraversalCallback = () => true;

  // Copy original callbacks to our initiatedCallbacks object or use default callbacks instead.
  initiatedCallbacks.allowTraversal = callbacks.allowTraversal || defaultAllowTraversalCallback;
  initiatedCallbacks.enterNode = callbacks.enterNode || stubCallback;
  initiatedCallbacks.leaveNode = callbacks.leaveNode || stubCallback;

  // Returned processed list of callbacks.
  return initiatedCallbacks;
}

/**
 * Recursive depth-first-search traversal for binary.
 *
 * @param {BinaryTreeNode} node - binary tree node that we will start traversal from.
 * @param {TraversalCallbacks} callbacks - the object that contains traversal callbacks.
 */
export function depthFirstSearchRecursive(node, callbacks) {
  // Call the "enterNode" callback to notify that the node is going to be entered.
  callbacks.enterNode(node);

  // Traverse left branch only if case if traversal of the left node is allowed.
  if (node.left && callbacks.allowTraversal(node, node.left)) {
    depthFirstSearchRecursive(node.left, callbacks);
  }

  // Traverse right branch only if case if traversal of the right node is allowed.
  if (node.right && callbacks.allowTraversal(node, node.right)) {
    depthFirstSearchRecursive(node.right, callbacks);
  }

  // Call the "leaveNode" callback to notify that traversal
  // of the current node and its children is finished.
  callbacks.leaveNode(node);
}

/**
 * Perform depth-first-search traversal of the rootNode.
 * For every traversal step call "allowTraversal", "enterNode" and "leaveNode" callbacks.
 * See TraversalCallbacks type definition for more details about the shape of callbacks object.
 *
 * @param {BinaryTreeNode} rootNode - The node from which we start traversing.
 * @param {TraversalCallbacks} [callbacks] - Traversal callbacks.
 */
export default function depthFirstSearch(rootNode, callbacks) {
  // In case if user didn't provide some callback we need to replace them with default ones.
  const processedCallbacks = initCallbacks(callbacks);

  // Now, when we have all necessary callbacks we may proceed to recursive traversal.
  depthFirstSearchRecursive(rootNode, processedCallbacks);
}
```

Listing 206: hanoiTower.test.js

```
import hanoiTower from '../hanoiTower';
import Stack from '../../../../data-structures/stack/Stack';

describe('hanoiTower', () => {
  it('should solve tower of hanoi puzzle with 2 discs', () => {
    const moveCallback = jest.fn();
    const numberOfDiscs = 2;

    const fromPole = new Stack();
    const withPole = new Stack();
    const toPole = new Stack();

    hanoiTower({
      numberOfDiscs,
      moveCallback,
      fromPole,
      withPole,
      toPole,
    });

    expect(moveCallback).toHaveBeenCalledTimes((2 ** numberOfDiscs) - 1);

    expect(fromPole.toArray()).toEqual([]);
    expect(toPole.toArray()).toEqual([1, 2]);

    expect(moveCallback.mock.calls[0][0]).toBe(1);
    expect(moveCallback.mock.calls[0][1]).toEqual([1, 2]);
    expect(moveCallback.mock.calls[0][2]).toEqual([]);

    expect(moveCallback.mock.calls[1][0]).toBe(2);
    expect(moveCallback.mock.calls[1][1]).toEqual([2]);
    expect(moveCallback.mock.calls[1][2]).toEqual([]);

    expect(moveCallback.mock.calls[2][0]).toBe(1);
    expect(moveCallback.mock.calls[2][1]).toEqual([1]);
    expect(moveCallback.mock.calls[2][2]).toEqual([2]);
  });

  it('should solve tower of hanoi puzzle with 3 discs', () => {
    const moveCallback = jest.fn();
    const numberOfDiscs = 3;

    hanoiTower({
      numberOfDiscs,
      moveCallback,
    });

    expect(moveCallback).toHaveBeenCalledTimes((2 ** numberOfDiscs) - 1);
  });

  it('should solve tower of hanoi puzzle with 6 discs', () => {
    const moveCallback = jest.fn();
    const numberOfDiscs = 6;

    hanoiTower({
      numberOfDiscs,
      moveCallback,
    });

    expect(moveCallback).toHaveBeenCalledTimes((2 ** numberOfDiscs) - 1);
  });
});
```

Listing 207: hanoiTower.js

```
import Stack from '../../../data-structures/stack/Stack';

/**
 * @param {number} numberOfDiscs
 * @param {Stack} fromPole
 * @param {Stack} withPole
 * @param {Stack} toPole
 * @param {function(disc: number, fromPole: number[], toPole: number[])} moveCallback
 */
function hanoiTowerRecursive({
  numberOfDiscs,
  fromPole,
  withPole,
  toPole,
  moveCallback,
}) {
  if (numberOfDiscs === 1) {
    // Base case with just one disc.
    moveCallback(fromPole.peek(), fromPole.toArray(), toPole.toArray());
    const disc = fromPole.pop();
    toPole.push(disc);
  } else {
    // In case if there are more discs then move them recursively.

    // Expose the bottom disc on fromPole stack.
    hanoiTowerRecursive({
      numberOfDiscs: numberOfDiscs - 1,
      fromPole,
      withPole: toPole,
      toPole: withPole,
      moveCallback,
    });

    // Move the disc that was exposed to its final destination.
    hanoiTowerRecursive({
      numberOfDiscs: 1,
      fromPole,
      withPole,
      toPole,
      moveCallback,
    });

    // Move temporary tower from auxiliary pole to its final destination.
    hanoiTowerRecursive({
      numberOfDiscs: numberOfDiscs - 1,
      fromPole: withPole,
      withPole: fromPole,
      toPole,
      moveCallback,
    });
  }
}

/**
 * @param {number} numberOfDiscs
 * @param {function(disc: number, fromPole: number[], toPole: number[])} moveCallback
 * @param {Stack} [fromPole]
 * @param {Stack} [withPole]
 * @param {Stack} [toPole]
 */
export default function hanoiTower({
  numberOfDiscs,
  moveCallback,
  fromPole = new Stack(),
  withPole = new Stack(),
  toPole = new Stack(),
}) {
  // Each of three poles of Tower of Hanoi puzzle is represented as a stack
  // that might contain elements (discs). Each disc is represented as a number.
  // Larger discs have bigger number equivalent.

  // Let's create the discs and put them to the fromPole.
  for (let discSize = numberOfDiscs; discSize > 0; discSize -= 1) {
    fromPole.push(discSize);
  }

  hanoiTowerRecursive({
    numberOfDiscs,
    fromPole,
```

```
    withPole,
    toPole,
    moveCallback,
  });
}
```

Listing 208: backtrackingJumpGame.test.js

```
import backtrackingJumpGame from '../backtrackingJumpGame';

describe('backtrackingJumpGame', () => {
  it('should solve Jump Game problem in backtracking manner', () => {
    expect(backtrackingJumpGame([1, 0])).toBe(true);
    expect(backtrackingJumpGame([100, 0])).toBe(true);
    expect(backtrackingJumpGame([2, 3, 1, 1, 4])).toBe(true);
    expect(backtrackingJumpGame([1, 1, 1, 1, 1])).toBe(true);
    expect(backtrackingJumpGame([1, 1, 1, 10, 1])).toBe(true);
    expect(backtrackingJumpGame([1, 5, 2, 1, 0, 2, 0])).toBe(true);

    expect(backtrackingJumpGame([1, 0, 1])).toBe(false);
    expect(backtrackingJumpGame([3, 2, 1, 0, 4])).toBe(false);
    expect(backtrackingJumpGame([0, 0, 0, 0, 0])).toBe(false);
    expect(backtrackingJumpGame([5, 4, 3, 2, 1, 0, 0])).toBe(false);
  });
});
```

Listing 209: dpBottomUpJumpGame.test.js

```
import dpBottomUpJumpGame from '../dpBottomUpJumpGame';

describe('dpBottomUpJumpGame', () => {
  it('should solve Jump Game problem in bottom-up dynamic programming manner', () => {
    expect(dpBottomUpJumpGame([1, 0])).toBe(true);
    expect(dpBottomUpJumpGame([100, 0])).toBe(true);
    expect(dpBottomUpJumpGame([2, 3, 1, 1, 4])).toBe(true);
    expect(dpBottomUpJumpGame([1, 1, 1, 1, 1])).toBe(true);
    expect(dpBottomUpJumpGame([1, 1, 1, 10, 1])).toBe(true);
    expect(dpBottomUpJumpGame([1, 5, 2, 1, 0, 2, 0])).toBe(true);

    expect(dpBottomUpJumpGame([1, 0, 1])).toBe(false);
    expect(dpBottomUpJumpGame([3, 2, 1, 0, 4])).toBe(false);
    expect(dpBottomUpJumpGame([0, 0, 0, 0, 0])).toBe(false);
    expect(dpBottomUpJumpGame([5, 4, 3, 2, 1, 0, 0])).toBe(false);
  });
});
```

Listing 210: dpTopDownJumpGame.test.js

```
import dpTopDownJumpGame from '../dpTopDownJumpGame';

describe('dpTopDownJumpGame', () => {
  it('should solve Jump Game problem in top-down dynamic programming manner', () => {
    expect(dpTopDownJumpGame([1, 0])).toBe(true);
    expect(dpTopDownJumpGame([100, 0])).toBe(true);
    expect(dpTopDownJumpGame([2, 3, 1, 1, 4])).toBe(true);
    expect(dpTopDownJumpGame([1, 1, 1, 1, 1])).toBe(true);
    expect(dpTopDownJumpGame([1, 1, 1, 10, 1])).toBe(true);
    expect(dpTopDownJumpGame([1, 5, 2, 1, 0, 2, 0])).toBe(true);

    expect(dpTopDownJumpGame([1, 0, 1])).toBe(false);
    expect(dpTopDownJumpGame([3, 2, 1, 0, 4])).toBe(false);
    expect(dpTopDownJumpGame([0, 0, 0, 0, 0])).toBe(false);
    expect(dpTopDownJumpGame([5, 4, 3, 2, 1, 0, 0])).toBe(false);
  });
});
```

Listing 211: greedyJumpGame.test.js

```
import greedyJumpGame from '../greedyJumpGame';

describe('greedyJumpGame', () => {
  it('should solve Jump Game problem in greedy manner', () => {
    expect(greedyJumpGame([1, 0])).toBe(true);
    expect(greedyJumpGame([100, 0])).toBe(true);
    expect(greedyJumpGame([2, 3, 1, 1, 4])).toBe(true);
    expect(greedyJumpGame([1, 1, 1, 1, 1])).toBe(true);
    expect(greedyJumpGame([1, 1, 1, 10, 1])).toBe(true);
    expect(greedyJumpGame([1, 5, 2, 1, 0, 2, 0])).toBe(true);

    expect(greedyJumpGame([1, 0, 1])).toBe(false);
    expect(greedyJumpGame([3, 2, 1, 0, 4])).toBe(false);
    expect(greedyJumpGame([0, 0, 0, 0, 0])).toBe(false);
    expect(greedyJumpGame([5, 4, 3, 2, 1, 0, 0])).toBe(false);
  });
});
```

Listing 212: backtrackingJumpGame.js

```
/**
 * BACKTRACKING approach of solving Jump Game.
 *
 * This is the inefficient solution where we try every single jump
 * pattern that takes us from the first position to the last.
 * We start from the first position and jump to every index that
 * is reachable. We repeat the process until last index is reached.
 * When stuck, backtrack.
 *
 * @param {number[]} numbers - array of possible jump length.
 * @param {number} startIndex - index from where we start jumping.
 * @param {number[]} currentJumps - current jumps path.
 * @return {boolean}
 */
export default function backtrackingJumpGame(numbers, startIndex = 0, currentJumps = []) {
  if (startIndex === numbers.length - 1) {
    // We've jumped directly to last cell. This situation is a solution.
    return true;
  }

  // Check what the longest jump we could make from current position.
  // We don't need to jump beyond the array.
  const maxJumpLength = Math.min(
    numbers[startIndex], // Jump is within array.
    numbers.length - 1 - startIndex, // Jump goes beyond array.
  );

  // Let's start jumping from startIndex and see whether any
  // jump is successful and has reached the end of the array.
  for (let jumpLength = maxJumpLength; jumpLength > 0; jumpLength -= 1) {
    // Try next jump.
    const nextIndex = startIndex + jumpLength;
    currentJumps.push(nextIndex);

    const isJumpSuccessful = backtrackingJumpGame(numbers, nextIndex, currentJumps);

    // Check if current jump was successful.
    if (isJumpSuccessful) {
      return true;
    }

    // BACKTRACKING.
    // If previous jump wasn't successful then retreat and try the next one.
    currentJumps.pop();
  }

  return false;
}
```

Listing 213: dpBottomUpJumpGame.js

```
/**
 * DYNAMIC PROGRAMMING BOTTOM-UP approach of solving Jump Game.
 *
 * This comes out as an optimisation of DYNAMIC PROGRAMMING TOP-DOWN approach.
 *
 * The observation to make here is that we only ever jump to the right.
 * This means that if we start from the right of the array, every time we
 * will query a position to our right, that position has already be
 * determined as being GOOD or BAD. This means we don't need to recurse
 * anymore, as we will always hit the memo table.
 *
 * We call a position in the array a "good" one if starting at that
 * position, we can reach the last index. Otherwise, that index
 * is called a "bad" one.
 *
 * @param {number[]} numbers - array of possible jump length.
 * @return {boolean}
 */
export default function dpBottomUpJumpGame(numbers) {
  // Init cells goodness table.
  const cellsGoodness = Array(numbers.length).fill(undefined);
  // Mark the last cell as "good" one since it is where we ultimately want to get.
  cellsGoodness[cellsGoodness.length - 1] = true;

  // Go throw all cells starting from the one before the last
  // one (since the last one is "good" already) and fill cellsGoodness table.
  for (let cellIndex = numbers.length - 2; cellIndex >= 0; cellIndex -= 1) {
    const maxJumpLength = Math.min(
      numbers[cellIndex],
      numbers.length - 1 - cellIndex,
    );

    for (let jumpLength = maxJumpLength; jumpLength > 0; jumpLength -= 1) {
      const nextIndex = cellIndex + jumpLength;
      if (cellsGoodness[nextIndex] === true) {
        cellsGoodness[cellIndex] = true;
        // Once we detected that current cell is good one we don't need to
        // do further cells checking.
        break;
      }
    }
  }

  // Now, if the zero's cell is good one then we can jump from it to the end of the array.
  return cellsGoodness[0] === true;
}
```

Listing 214: dpTopDownJumpGame.js

```
/**
 * DYNAMIC PROGRAMMING TOP-DOWN approach of solving Jump Game.
 *
 * This comes out as an optimisation of BACKTRACKING approach.
 *
 * It relies on the observation that once we determine that a certain
 * index is good / bad, this result will never change. This means that
 * we can store the result and not need to recompute it every time.
 *
 * We call a position in the array a "good" one if starting at that
 * position, we can reach the last index. Otherwise, that index
 * is called a "bad" one.
 *
 * @param {number[]} numbers - array of possible jump length.
 * @param {number} startIndex - index from where we start jumping.
 * @param {number[]} currentJumps - current jumps path.
 * @param {boolean[]} cellsGoodness - holds information about whether cell is "good" or "bad"
 * @return {boolean}
 */
export default function dpTopDownJumpGame(
  numbers,
  startIndex = 0,
  currentJumps = [],
  cellsGoodness = [],
) {
  if (startIndex === numbers.length - 1) {
    // We've jumped directly to last cell. This situation is a solution.
    return true;
  }

  // Init cell goodness table if it is empty.
  // This is DYNAMIC PROGRAMMING feature.
  const currentCellsGoodness = [...cellsGoodness];
  if (!currentCellsGoodness.length) {
    numbers.forEach(() => currentCellsGoodness.push(undefined));
    // Mark the last cell as "good" one since it is where we ultimately want to get.
    currentCellsGoodness[cellsGoodness.length - 1] = true;
  }

  // Check what the longest jump we could make from current position.
  // We don't need to jump beyond the array.
  const maxJumpLength = Math.min(
    numbers[startIndex], // Jump is within array.
    numbers.length - 1 - startIndex, // Jump goes beyond array.
  );

  // Let's start jumping from startIndex and see whether any
  // jump is successful and has reached the end of the array.
  for (let jumpLength = maxJumpLength; jumpLength > 0; jumpLength -= 1) {
    // Try next jump.
    const nextIndex = startIndex + jumpLength;

    // Jump only into "good" or "unknown" cells.
    // This is top-down dynamic programming optimisation of backtracking algorithm.
    if (currentCellsGoodness[nextIndex] !== false) {
      currentJumps.push(nextIndex);

      const isJumpSuccessful = dpTopDownJumpGame(
        numbers,
        nextIndex,
        currentJumps,
        currentCellsGoodness,
      );

      // Check if current jump was successful.
      if (isJumpSuccessful) {
        return true;
      }

      // BACKTRACKING.
      // If previous jump wasn't successful then retreat and try the next one.
      currentJumps.pop();

      // Mark current cell as "bad" to avoid its deep visiting later.
      currentCellsGoodness[nextIndex] = false;
    }
  }

  return false;
```

```
}
```

Listing 215: greedyJumpGame.js

```
/**
 * GREEDY approach of solving Jump Game.
 *
 * This comes out as an optimisation of DYNAMIC PROGRAMMING BOTTOM_UP approach.
 *
 * Once we have our code in the bottom-up state, we can make one final,
 * important observation. From a given position, when we try to see if
 * we can jump to a GOOD position, we only ever use one - the first one.
 * In other words, the left-most one. If we keep track of this left-most
 * GOOD position as a separate variable, we can avoid searching for it
 * in the array. Not only that, but we can stop using the array altogether.
 *
 * We call a position in the array a "good" one if starting at that
 * position, we can reach the last index. Otherwise, that index
 * is called a "bad" one.
 *
 * @param {number[]} numbers - array of possible jump length.
 * @return {boolean}
 */
export default function greedyJumpGame(numbers) {
  // The "good" cell is a cell from which we may jump to the last cell of the numbers array.

  // The last cell in numbers array is for sure the "good" one since it is our goal to reach.
  let leftGoodPosition = numbers.length - 1;

  // Go through all numbers from right to left.
  for (let numberIndex = numbers.length - 2; numberIndex >= 0; numberIndex -= 1) {
    // If we can reach the "good" cell from the current one then for sure the current
    // one is also "good". Since after all we'll be able to reach the end of the array
    // from it.
    const maxCurrentJumpLength = numberIndex + numbers[numberIndex];
    if (maxCurrentJumpLength >= leftGoodPosition) {
      leftGoodPosition = numberIndex;
    }
  }

  // If the most left "good" position is the zero's one then we may say that it IS
  // possible jump to the end of the array from the first cell;
  return leftGoodPosition === 0;
}
```

Listing 216: knightTour.test.js

```
import knightTour from '../knightTour';

describe('knightTour', () => {
  it('should not find solution on 3x3 board', () => {
    const moves = knightTour(3);

    expect(moves.length).toBe(0);
  });

  it('should find one solution to do knight tour on 5x5 board', () => {
    const moves = knightTour(5);

    expect(moves.length).toBe(25);

    expect(moves).toEqual([
      [0, 0],
      [1, 2],
      [2, 0],
      [0, 1],
      [1, 3],
      [3, 4],
      [2, 2],
      [4, 1],
      [3, 3],
      [1, 4],
      [0, 2],
      [1, 0],
      [3, 1],
      [4, 3],
      [2, 4],
      [0, 3],
      [1, 1],
      [3, 0],
      [4, 2],
      [2, 1],
      [4, 0],
      [3, 2],
      [4, 4],
      [2, 3],
      [0, 4],
    ]);
  });
});
```

Listing 217: knightTour.js

```
/**
 * @param {number[][]} chessboard
 * @param {number[]} position
 * @return {number[][]}
 */
function getPossibleMoves(chessboard, position) {
  // Generate all knight moves (even those that go beyond the board).
  const possibleMoves = [
    [position[0] - 1, position[1] - 2],
    [position[0] - 2, position[1] - 1],
    [position[0] + 1, position[1] - 2],
    [position[0] + 2, position[1] - 1],
    [position[0] - 2, position[1] + 1],
    [position[0] - 1, position[1] + 2],
    [position[0] + 1, position[1] + 2],
    [position[0] + 2, position[1] + 1],
  ];

  // Filter out all moves that go beyond the board.
  return possibleMoves.filter((move) => {
    const boardSize = chessboard.length;
    return move[0] >= 0 && move[1] >= 0 && move[0] < boardSize && move[1] < boardSize;
  });
}

/**
 * @param {number[][]} chessboard
 * @param {number[]} move
 * @return {boolean}
 */
function isMoveAllowed(chessboard, move) {
  return chessboard[move[0]][move[1]] !== 1;
}

/**
 * @param {number[][]} chessboard
 * @param {number[][]} moves
 * @return {boolean}
 */
function isBoardCompletelyVisited(chessboard, moves) {
  const totalPossibleMovesCount = chessboard.length ** 2;
  const existingMovesCount = moves.length;

  return totalPossibleMovesCount === existingMovesCount;
}

/**
 * @param {number[][]} chessboard
 * @param {number[][]} moves
 * @return {boolean}
 */
function knightTourRecursive(chessboard, moves) {
  const currentChessboard = chessboard;

  // If board has been completely visited then we've found a solution.
  if (isBoardCompletelyVisited(currentChessboard, moves)) {
    return true;
  }

  // Get next possible knight moves.
  const lastMove = moves[moves.length - 1];
  const possibleMoves = getPossibleMoves(currentChessboard, lastMove);

  // Try to do next possible moves.
  for (let moveIndex = 0; moveIndex < possibleMoves.length; moveIndex += 1) {
    const currentMove = possibleMoves[moveIndex];

    // Check if current move is allowed. We aren't allowed to go to
    // the same cells twice.
    if (isMoveAllowed(currentChessboard, currentMove)) {
      // Actually do the move.
      moves.push(currentMove);
      currentChessboard[currentMove[0]][currentMove[1]] = 1;

      // If further moves starting from current are successful then
      // return true meaning the solution is found.
      if (knightTourRecursive(currentChessboard, moves)) {
        return true;
      }
```

```
      // BACKTRACKING.
      // If current move was unsuccessful then step back and try to do another move.
      moves.pop();
      currentChessboard[currentMove[0]][currentMove[1]] = 0;
    }
  }

  // Return false if we haven't found solution.
  return false;
}

/**
 * @param {number} chessboardSize
 * @return {number[][]}
 */
export default function knightTour(chessboardSize) {
  // Init chessboard.
  const chessboard = Array(chessboardSize).fill(null).map(() => Array(chessboardSize).fill(0));

  // Init moves array.
  const moves = [];

  // Do first move and place the knight to the 0x0 cell.
  const firstMove = [0, 0];
  moves.push(firstMove);
  chessboard[firstMove[0]][firstMove[0]] = 1;

  // Recursively try to do the next move.
  const solutionWasFound = knightTourRecursive(chessboard, moves);

  return solutionWasFound ? moves : [];
}
```

Listing 218: nQueens.test.js

```
import nQueens from '../nQueens';

describe('nQueens', () => {
  it('should not hae solution for 3 queens', () => {
    const solutions = nQueens(3);
    expect(solutions.length).toBe(0);
  });

  it('should solve n-queens problem for 4 queens', () => {
    const solutions = nQueens(4);
    expect(solutions.length).toBe(2);

    // First solution.
    expect(solutions[0][0].toString()).toBe('0,1');
    expect(solutions[0][1].toString()).toBe('1,3');
    expect(solutions[0][2].toString()).toBe('2,0');
    expect(solutions[0][3].toString()).toBe('3,2');

    // Second solution (mirrored).
    expect(solutions[1][0].toString()).toBe('0,2');
    expect(solutions[1][1].toString()).toBe('1,0');
    expect(solutions[1][2].toString()).toBe('2,3');
    expect(solutions[1][3].toString()).toBe('3,1');
  });

  it('should solve n-queens problem for 6 queens', () => {
    const solutions = nQueens(6);
    expect(solutions.length).toBe(4);

    // First solution.
    expect(solutions[0][0].toString()).toBe('0,1');
    expect(solutions[0][1].toString()).toBe('1,3');
    expect(solutions[0][2].toString()).toBe('2,5');
    expect(solutions[0][3].toString()).toBe('3,0');
    expect(solutions[0][4].toString()).toBe('4,2');
    expect(solutions[0][5].toString()).toBe('5,4');
  });
});
```

Listing 219: nQueensBitwise.test.js

```
import nQueensBitwise from '../nQueensBitwise';

describe('nQueensBitwise', () => {
  it('should have solutions for 4 to N queens', () => {
    expect(nQueensBitwise(4)).toBe(2);
    expect(nQueensBitwise(5)).toBe(10);
    expect(nQueensBitwise(6)).toBe(4);
    expect(nQueensBitwise(7)).toBe(40);
    expect(nQueensBitwise(8)).toBe(92);
    expect(nQueensBitwise(9)).toBe(352);
    expect(nQueensBitwise(10)).toBe(724);
    expect(nQueensBitwise(11)).toBe(2680);
  });
});
```

Listing 220: QueensPosition.test.js

```
import QueenPosition from '../QueenPosition';

describe('QueenPosition', () => {
  it('should store queen position on chessboard', () => {
    const position1 = new QueenPosition(0, 0);
    const position2 = new QueenPosition(2, 1);

    expect(position2.columnIndex).toBe(1);
    expect(position2.rowIndex).toBe(2);
    expect(position1.leftDiagonal).toBe(0);
    expect(position1.rightDiagonal).toBe(0);
    expect(position2.leftDiagonal).toBe(1);
    expect(position2.rightDiagonal).toBe(3);
    expect(position2.toString()).toBe('2,1');
  });
});
```

Listing 221: nQueens.js

```
import QueenPosition from './QueenPosition';

/**
 * @param {QueenPosition[]} queensPositions
 * @param {number} rowIndex
 * @param {number} columnIndex
 * @return {boolean}
 */
function isSafe(queensPositions, rowIndex, columnIndex) {
  // New position to which the Queen is going to be placed.
  const newQueenPosition = new QueenPosition(rowIndex, columnIndex);

  // Check if new queen position conflicts with any other queens.
  for (let queenIndex = 0; queenIndex < queensPositions.length; queenIndex += 1) {
    const currentQueenPosition = queensPositions[queenIndex];

    if (
      // Check if queen has been already placed.
      currentQueenPosition
      && (
        // Check if there are any queen on the same column.
        newQueenPosition.columnIndex === currentQueenPosition.columnIndex
        // Check if there are any queen on the same row.
        || newQueenPosition.rowIndex === currentQueenPosition.rowIndex
        // Check if there are any queen on the same left diagonal.
        || newQueenPosition.leftDiagonal === currentQueenPosition.leftDiagonal
        // Check if there are any queen on the same right diagonal.
        || newQueenPosition.rightDiagonal === currentQueenPosition.rightDiagonal
      )
    ) {
      // Can't place queen into current position since there are other queens that
      // are threatening it.
      return false;
    }
  }

  // Looks like we're safe.
  return true;
}

/**
 * @param {QueenPosition[][]} solutions
 * @param {QueenPosition[]} previousQueensPositions
 * @param {number} queensCount
 * @param {number} rowIndex
 * @return {boolean}
 */
function nQueensRecursive(solutions, previousQueensPositions, queensCount, rowIndex) {
  // Clone positions array.
  const queensPositions = [...previousQueensPositions].map((queenPosition) => {
    return !queenPosition ? queenPosition : new QueenPosition(
      queenPosition.rowIndex,
      queenPosition.columnIndex,
    );
  });

  if (rowIndex === queensCount) {
    // We've successfully reached the end of the board.
    // Store solution to the list of solutions.
    solutions.push(queensPositions);

    // Solution found.
    return true;
  }

  // Let's try to put queen at row rowIndex into its safe column position.
  for (let columnIndex = 0; columnIndex < queensCount; columnIndex += 1) {
    if (isSafe(queensPositions, rowIndex, columnIndex)) {
      // Place current queen to its current position.
      queensPositions[rowIndex] = new QueenPosition(rowIndex, columnIndex);

      // Try to place all other queens as well.
      nQueensRecursive(solutions, queensPositions, queensCount, rowIndex + 1);

      // BACKTRACKING.
      // Remove the queen from the row to avoid isSafe() returning false.
      queensPositions[rowIndex] = null;
    }
  }
```

```
  return false;
}


/**
 * @param {number} queensCount
 * @return {QueenPosition[][]}
 */
export default function nQueens(queensCount) {
  // Init NxN chessboard with zeros.
  // const chessboard = Array(queensCount).fill(null).map(() => Array(queensCount).fill(0));

  // This array will hold positions or coordinates of each of
  // N queens in form of [rowIndex, columnIndex].
  const queensPositions = Array(queensCount).fill(null);

  /** @var {QueenPosition[][]} solutions */
  const solutions = [];

  // Solve problem recursively.
  nQueensRecursive(solutions, queensPositions, queensCount, 0);

  return solutions;
}
```

Listing 222: nQueensBitwise.js

```javascript
/**
 * Checks all possible board configurations.
 *
 * @param {number} boardSize - Size of the squared chess board.
 * @param {number} leftDiagonal - Sequence of N bits that show whether the corresponding location
 * on the current row is "available" (no other queens are threatening from left diagonal).
 * @param {number} column - Sequence of N bits that show whether the corresponding location
 * on the current row is "available" (no other queens are threatening from columns).
 * @param {number} rightDiagonal - Sequence of N bits that show whether the corresponding location
 * on the current row is "available" (no other queens are threatening from right diagonal).
 * @param {number} solutionsCount - Keeps track of the number of valid solutions.
 * @return {number} - Number of possible solutions.
 */
function nQueensBitwiseRecursive(
  boardSize,
  leftDiagonal = 0,
  column = 0,
  rightDiagonal = 0,
  solutionsCount = 0,
) {
  // Keeps track of the number of valid solutions.
  let currentSolutionsCount = solutionsCount;

  // Helps to identify valid solutions.
  // isDone simply has a bit sequence with 1 for every entry up to the Nth. For example,
  // when N=5, done will equal 11111. The "isDone" variable simply allows us to not worry about any
  // bits beyond the Nth.
  const isDone = (2 ** boardSize) - 1;

  // All columns are occupied (i.e. 0b1111 for boardSize = 4), so the solution must be complete.
  // Since the algorithm never places a queen illegally (ie. when it can attack or be attacked),
  // we know that if all the columns have been filled, we must have a valid solution.
  if (column === isDone) {
    return currentSolutionsCount + 1;
  }

  // Gets a bit sequence with "1"s wherever there is an open "slot".
  // All that's happening here is we're taking col, ld, and rd, and if any of the columns are
  // "under attack", we mark that column as 0 in poss, basically meaning "we can't put a queen in
  // this column". Thus all bits position in poss that are '1's are available for placing
  // queen there.
  let availablePositions = ~(leftDiagonal | rightDiagonal | column);

  // Loops as long as there is a valid place to put another queen.
  // For N=4 the isDone=0b1111. Then if availablePositions=0b0000 (which would mean that all places
  // are under threatening) we must stop trying to place a queen.
  while (availablePositions & isDone) {
    // firstAvailablePosition just stores the first non-zero bit (ie. the first available location).
    // So if firstAvailablePosition was 0010, it would mean the 3rd column of the current row.
    // And that would be the position will be placing our next queen.
    //
    // For example:
    // availablePositions = 0b01100
    // firstAvailablePosition = 100
    const firstAvailablePosition = availablePositions & -availablePositions;

    // This line just marks that position in the current row as being "taken" by flipping that
    // column in availablePositions to zero. This way, when the while loop continues, we'll know
    // not to try that location again.
    //
    // For example:
    // availablePositions = 0b0100
    // firstAvailablePosition = 0b10
    // 0b0110 - 0b10 = 0b0100
    availablePositions -= firstAvailablePosition;

    /*
     * The operators >> 1 and 1 << simply move all the bits in a bit sequence one digit to the
     * right or left, respectively. So calling (rd|bit)<<1 simply says: combine rd and bit with
     * an OR operation, then move everything in the result to the left by one digit.
     *
     * More specifically, if rd is 0001 (meaning that the top-right-to-bottom-left diagonal through
     * column 4 of the current row is occupied), and bit is 0100 (meaning that we are planning to
     * place a queen in column 2 of the current row), (rd|bit) results in 0101 (meaning that after
     * we place a queen in column 2 of the current row, the second and the fourth
     * top-right-to-bottom-left diagonals will be occupied).
     *
     * Now, if add in the << operator, we get (rd|bit)<<1, which takes the 0101 we worked out in
     * our previous bullet point, and moves everything to the left by one. The result, therefore,
```

```
     * is 1010.
     */
    currentSolutionsCount += nQueensBitwiseRecursive(
      boardSize,
      (leftDiagonal | firstAvailablePosition) >> 1,
      column | firstAvailablePosition,
      (rightDiagonal | firstAvailablePosition) << 1,
      solutionsCount,
    );
  }

  return currentSolutionsCount;
}

/**
 * @param {number} boardSize - Size of the squared chess board.
 * @return {number} - Number of possible solutions.
 * @see http://gregtrowbridge.com/a-bitwise-solution-to-the-n-queens-problem-in-javascript/
 */
export default function nQueensBitwise(boardSize) {
  return nQueensBitwiseRecursive(boardSize);
}
```

Listing 223: QueenPosition.js

```
/**
 * Class that represents queen position on the chessboard.
 */
export default class QueenPosition {
  /**
   * @param {number} rowIndex
   * @param {number} columnIndex
   */
  constructor(rowIndex, columnIndex) {
    this.rowIndex = rowIndex;
    this.columnIndex = columnIndex;
  }

  /**
   * @return {number}
   */
  get leftDiagonal() {
    // Each position on the same left (\) diagonal has the same difference of
    // rowIndex and columnIndex. This fact may be used to quickly check if two
    // positions (queens) are on the same left diagonal.
    // @see https://youtu.be/xouin83ebxE?t=1m59s
    return this.rowIndex - this.columnIndex;
  }

  /**
   * @return {number}
   */
  get rightDiagonal() {
    // Each position on the same right diagonal (/) has the same
    // sum of rowIndex and columnIndex. This fact may be used to quickly
    // check if two positions (queens) are on the same right diagonal.
    // @see https://youtu.be/xouin83ebxE?t=1m59s
    return this.rowIndex + this.columnIndex;
  }

  toString() {
    return `${this.rowIndex},${this.columnIndex}`;
  }
}
```

Listing 224: bfRainTerraces.test.js

```
import bfRainTerraces from '../bfRainTerraces';

describe('bfRainTerraces', () => {
  it('should find the amount of water collected after raining', () => {
    expect(bfRainTerraces([1])).toBe(0);
    expect(bfRainTerraces([1, 0])).toBe(0);
    expect(bfRainTerraces([0, 1])).toBe(0);
    expect(bfRainTerraces([0, 1, 0])).toBe(0);
    expect(bfRainTerraces([0, 1, 0, 0])).toBe(0);
    expect(bfRainTerraces([0, 1, 0, 0, 1, 0])).toBe(2);
    expect(bfRainTerraces([0, 2, 0, 0, 1, 0])).toBe(2);
    expect(bfRainTerraces([2, 0, 2])).toBe(2);
    expect(bfRainTerraces([2, 0, 5])).toBe(2);
    expect(bfRainTerraces([3, 0, 0, 2, 0, 4])).toBe(10);
    expect(bfRainTerraces([0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1])).toBe(6);
    expect(bfRainTerraces([1, 1, 1, 1, 1])).toBe(0);
    expect(bfRainTerraces([1, 2, 3, 4, 5])).toBe(0);
    expect(bfRainTerraces([4, 1, 3, 1, 2, 1, 2, 1])).toBe(4);
    expect(bfRainTerraces([0, 2, 4, 3, 4, 2, 4, 0, 8, 7, 0])).toBe(7);
  });
});
```

Listing 225: dpRainTerraces.test.js

```
import dpRainTerraces from '../dpRainTerraces';

describe('dpRainTerraces', () => {
  it('should find the amount of water collected after raining', () => {
    expect(dpRainTerraces([1])).toBe(0);
    expect(dpRainTerraces([1, 0])).toBe(0);
    expect(dpRainTerraces([0, 1])).toBe(0);
    expect(dpRainTerraces([0, 1, 0])).toBe(0);
    expect(dpRainTerraces([0, 1, 0, 0])).toBe(0);
    expect(dpRainTerraces([0, 1, 0, 0, 1, 0])).toBe(2);
    expect(dpRainTerraces([0, 2, 0, 0, 1, 0])).toBe(2);
    expect(dpRainTerraces([2, 0, 2])).toBe(2);
    expect(dpRainTerraces([2, 0, 5])).toBe(2);
    expect(dpRainTerraces([3, 0, 0, 2, 0, 4])).toBe(10);
    expect(dpRainTerraces([0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1])).toBe(6);
    expect(dpRainTerraces([1, 1, 1, 1, 1])).toBe(0);
    expect(dpRainTerraces([1, 2, 3, 4, 5])).toBe(0);
    expect(dpRainTerraces([4, 1, 3, 1, 2, 1, 2, 1])).toBe(4);
    expect(dpRainTerraces([0, 2, 4, 3, 4, 2, 4, 0, 8, 7, 0])).toBe(7);
  });
});
```

Listing 226: bfRainTerraces.js

```
/**
 * BRUTE FORCE approach of solving Trapping Rain Water problem.
 *
 * @param {number[]} terraces
 * @return {number}
 */
export default function bfRainTerraces(terraces) {
  let waterAmount = 0;

  for (let terraceIndex = 0; terraceIndex < terraces.length; terraceIndex += 1) {
    // Get left most high terrace.
    let leftHighestLevel = 0;
    for (let leftIndex = terraceIndex - 1; leftIndex >= 0; leftIndex -= 1) {
      leftHighestLevel = Math.max(leftHighestLevel, terraces[leftIndex]);
    }

    // Get right most high terrace.
    let rightHighestLevel = 0;
    for (let rightIndex = terraceIndex + 1; rightIndex < terraces.length; rightIndex += 1) {
      rightHighestLevel = Math.max(rightHighestLevel, terraces[rightIndex]);
    }

    // Add current terrace water amount.
    const terraceBoundaryLevel = Math.min(leftHighestLevel, rightHighestLevel);
    if (terraceBoundaryLevel > terraces[terraceIndex]) {
      // Terrace will be able to store the water if the lowest of two left and right highest
      // terraces are still higher than the current one.
      waterAmount += Math.min(leftHighestLevel, rightHighestLevel) - terraces[terraceIndex];
    }
  }

  return waterAmount;
}
```

Listing 227: dpRainTerraces.js

```
/**
 * DYNAMIC PROGRAMMING approach of solving Trapping Rain Water problem.
 *
 * @param {number[]} terraces
 * @return {number}
 */
export default function dpRainTerraces(terraces) {
  let waterAmount = 0;

  // Init arrays that will keep the list of left and right maximum levels for specific positions.
  const leftMaxLevels = new Array(terraces.length).fill(0);
  const rightMaxLevels = new Array(terraces.length).fill(0);

  // Calculate the highest terrace level from the LEFT relative to the current terrace.
  [leftMaxLevels[0]] = terraces;
  for (let terraceIndex = 1; terraceIndex < terraces.length; terraceIndex += 1) {
    leftMaxLevels[terraceIndex] = Math.max(
      terraces[terraceIndex],
      leftMaxLevels[terraceIndex - 1],
    );
  }

  // Calculate the highest terrace level from the RIGHT relative to the current terrace.
  rightMaxLevels[terraces.length - 1] = terraces[terraces.length - 1];
  for (let terraceIndex = terraces.length - 2; terraceIndex >= 0; terraceIndex -= 1) {
    rightMaxLevels[terraceIndex] = Math.max(
      terraces[terraceIndex],
      rightMaxLevels[terraceIndex + 1],
    );
  }

  // Not let's go through all terraces one by one and calculate how much water
  // each terrace may accumulate based on previously calculated values.
  for (let terraceIndex = 0; terraceIndex < terraces.length; terraceIndex += 1) {
    // Pick the lowest from the left/right highest terraces.
    const currentTerraceBoundary = Math.min(
      leftMaxLevels[terraceIndex],
      rightMaxLevels[terraceIndex],
    );

    if (currentTerraceBoundary > terraces[terraceIndex]) {
      waterAmount += currentTerraceBoundary - terraces[terraceIndex];
    }
  }

  return waterAmount;
}
```

Listing 228: recursiveStaircaseBF.test.js

```
import recursiveStaircaseBF from '../recursiveStaircaseBF';

describe('recursiveStaircaseBF', () => {
  it('should calculate number of variants using Brute Force solution', () => {
    expect(recursiveStaircaseBF(-1)).toBe(0);
    expect(recursiveStaircaseBF(0)).toBe(0);
    expect(recursiveStaircaseBF(1)).toBe(1);
    expect(recursiveStaircaseBF(2)).toBe(2);
    expect(recursiveStaircaseBF(3)).toBe(3);
    expect(recursiveStaircaseBF(4)).toBe(5);
    expect(recursiveStaircaseBF(5)).toBe(8);
    expect(recursiveStaircaseBF(6)).toBe(13);
    expect(recursiveStaircaseBF(7)).toBe(21);
    expect(recursiveStaircaseBF(8)).toBe(34);
    expect(recursiveStaircaseBF(9)).toBe(55);
    expect(recursiveStaircaseBF(10)).toBe(89);
  });
});
```

Listing 229: recursiveStaircaseDP.test.js

```
import recursiveStaircaseDP from '../recursiveStaircaseDP';

describe('recursiveStaircaseDP', () => {
  it('should calculate number of variants using Dynamic Programming solution', () => {
    expect(recursiveStaircaseDP(-1)).toBe(0);
    expect(recursiveStaircaseDP(0)).toBe(0);
    expect(recursiveStaircaseDP(1)).toBe(1);
    expect(recursiveStaircaseDP(2)).toBe(2);
    expect(recursiveStaircaseDP(3)).toBe(3);
    expect(recursiveStaircaseDP(4)).toBe(5);
    expect(recursiveStaircaseDP(5)).toBe(8);
    expect(recursiveStaircaseDP(6)).toBe(13);
    expect(recursiveStaircaseDP(7)).toBe(21);
    expect(recursiveStaircaseDP(8)).toBe(34);
    expect(recursiveStaircaseDP(9)).toBe(55);
    expect(recursiveStaircaseDP(10)).toBe(89);
  });
});
```

Listing 230: recursiveStaircaseIT.test.js

```
import recursiveStaircaseIT from '../recursiveStaircaseIT';

describe('recursiveStaircaseIT', () => {
  it('should calculate number of variants using Iterative solution', () => {
    expect(recursiveStaircaseIT(-1)).toBe(0);
    expect(recursiveStaircaseIT(0)).toBe(0);
    expect(recursiveStaircaseIT(1)).toBe(1);
    expect(recursiveStaircaseIT(2)).toBe(2);
    expect(recursiveStaircaseIT(3)).toBe(3);
    expect(recursiveStaircaseIT(4)).toBe(5);
    expect(recursiveStaircaseIT(5)).toBe(8);
    expect(recursiveStaircaseIT(6)).toBe(13);
    expect(recursiveStaircaseIT(7)).toBe(21);
    expect(recursiveStaircaseIT(8)).toBe(34);
    expect(recursiveStaircaseIT(9)).toBe(55);
    expect(recursiveStaircaseIT(10)).toBe(89);
  });
});
```

Listing 231: recursiveStaircaseMEM.test.js

```
import recursiveStaircaseMEM from '../recursiveStaircaseMEM';

describe('recursiveStaircaseMEM', () => {
  it('should calculate number of variants using Brute Force with Memoization', () => {
    expect(recursiveStaircaseMEM(-1)).toBe(0);
    expect(recursiveStaircaseMEM(0)).toBe(0);
    expect(recursiveStaircaseMEM(1)).toBe(1);
    expect(recursiveStaircaseMEM(2)).toBe(2);
    expect(recursiveStaircaseMEM(3)).toBe(3);
    expect(recursiveStaircaseMEM(4)).toBe(5);
    expect(recursiveStaircaseMEM(5)).toBe(8);
    expect(recursiveStaircaseMEM(6)).toBe(13);
    expect(recursiveStaircaseMEM(7)).toBe(21);
    expect(recursiveStaircaseMEM(8)).toBe(34);
    expect(recursiveStaircaseMEM(9)).toBe(55);
    expect(recursiveStaircaseMEM(10)).toBe(89);
  });
});
```

Listing 232: recursiveStaircaseBF.js

```
/**
 * Recursive Staircase Problem (Brute Force Solution).
 *
 * @param {number} stairsNum - Number of stairs to climb on.
 * @return {number} - Number of ways to climb a staircase.
 */
export default function recursiveStaircaseBF(stairsNum) {
  if (stairsNum <= 0) {
    // There is no way to go down - you climb the stairs only upwards.
    // Also if you're standing on the ground floor that you don't need to do any further steps.
    return 0;
  }

  if (stairsNum === 1) {
    // There is only one way to go to the first step.
    return 1;
  }

  if (stairsNum === 2) {
    // There are two ways to get to the second steps: (1 + 1) or (2).
    return 2;
  }

  // Sum up how many steps we need to take after doing one step up with the number of
  // steps we need to take after doing two steps up.
  return recursiveStaircaseBF(stairsNum - 1) + recursiveStaircaseBF(stairsNum - 2);
}
```

Listing 233: recursiveStaircaseDP.js

```
/**
 * Recursive Staircase Problem (Dynamic Programming Solution).
 *
 * @param {number} stairsNum - Number of stairs to climb on.
 * @return {number} - Number of ways to climb a staircase.
 */
export default function recursiveStaircaseDP(stairsNum) {
  if (stairsNum < 0) {
    // There is no way to go down - you climb the stairs only upwards.
    return 0;
  }

  // Init the steps vector that will hold all possible ways to get to the corresponding step.
  const steps = new Array(stairsNum + 1).fill(0);

  // Init the number of ways to get to the 0th, 1st and 2nd steps.
  steps[0] = 0;
  steps[1] = 1;
  steps[2] = 2;

  if (stairsNum <= 2) {
    // Return the number of ways to get to the 0th or 1st or 2nd steps.
    return steps[stairsNum];
  }

  // Calculate every next step based on two previous ones.
  for (let currentStep = 3; currentStep <= stairsNum; currentStep += 1) {
    steps[currentStep] = steps[currentStep - 1] + steps[currentStep - 2];
  }

  // Return possible ways to get to the requested step.
  return steps[stairsNum];
}
```

Listing 234: recursiveStaircaseIT.js

```
/**
 * Recursive Staircase Problem (Iterative Solution).
 *
 * @param {number} stairsNum - Number of stairs to climb on.
 * @return {number} - Number of ways to climb a staircase.
 */
export default function recursiveStaircaseIT(stairsNum) {
  if (stairsNum <= 0) {
    // There is no way to go down - you climb the stairs only upwards.
    // Also you don't need to do anything to stay on the 0th step.
    return 0;
  }

  // Init the number of ways to get to the 0th, 1st and 2nd steps.
  const steps = [1, 2];

  if (stairsNum <= 2) {
    // Return the number of possible ways of how to get to the 1st or 2nd steps.
    return steps[stairsNum - 1];
  }

  // Calculate the number of ways to get to the n'th step based on previous ones.
  // Comparing to Dynamic Programming solution we don't store info for all the steps but
  // rather for two previous ones only.
  for (let currentStep = 3; currentStep <= stairsNum; currentStep += 1) {
    [steps[0], steps[1]] = [steps[1], steps[0] + steps[1]];
  }

  // Return possible ways to get to the requested step.
  return steps[1];
}
```

Listing 235: recursiveStaircaseMEM.js

```
/**
 * Recursive Staircase Problem (Recursive Solution With Memoization).
 *
 * @param {number} totalStairs - Number of stairs to climb on.
 * @return {number} - Number of ways to climb a staircase.
 */
export default function recursiveStaircaseMEM(totalStairs) {
  // Memo table that will hold all recursively calculated results to avoid calculating them
  // over and over again.
  const memo = [];

  // Recursive closure.
  const getSteps = (stairsNum) => {
    if (stairsNum <= 0) {
      // There is no way to go down - you climb the stairs only upwards.
      // Also if you're standing on the ground floor that you don't need to do any further steps.
      return 0;
    }

    if (stairsNum === 1) {
      // There is only one way to go to the first step.
      return 1;
    }

    if (stairsNum === 2) {
      // There are two ways to get to the second steps: (1 + 1) or (2).
      return 2;
    }

    // Avoid recursion for the steps that we've calculated recently.
    if (memo[stairsNum]) {
      return memo[stairsNum];
    }

    // Sum up how many steps we need to take after doing one step up with the number of
    // steps we need to take after doing two steps up.
    memo[stairsNum] = getSteps(stairsNum - 1) + getSteps(stairsNum - 2);

    return memo[stairsNum];
  };

  // Return possible ways to get to the requested step.
  return getSteps(totalStairs);
}
```

Listing 236: squareMatrixRotation.test.js

```
import squareMatrixRotation from '../squareMatrixRotation';

describe('squareMatrixRotation', () => {
  it('should rotate matrix #0 in-place', () => {
    const matrix = [[1]];

    const rotatedMatrix = [[1]];

    expect(squareMatrixRotation(matrix)).toEqual(rotatedMatrix);
  });

  it('should rotate matrix #1 in-place', () => {
    const matrix = [
      [1, 2],
      [3, 4],
    ];

    const rotatedMatrix = [
      [3, 1],
      [4, 2],
    ];

    expect(squareMatrixRotation(matrix)).toEqual(rotatedMatrix);
  });

  it('should rotate matrix #2 in-place', () => {
    const matrix = [
      [1, 2, 3],
      [4, 5, 6],
      [7, 8, 9],
    ];

    const rotatedMatrix = [
      [7, 4, 1],
      [8, 5, 2],
      [9, 6, 3],
    ];

    expect(squareMatrixRotation(matrix)).toEqual(rotatedMatrix);
  });

  it('should rotate matrix #3 in-place', () => {
    const matrix = [
      [5, 1, 9, 11],
      [2, 4, 8, 10],
      [13, 3, 6, 7],
      [15, 14, 12, 16],
    ];

    const rotatedMatrix = [
      [15, 13, 2, 5],
      [14, 3, 4, 1],
      [12, 6, 8, 9],
      [16, 7, 10, 11],
    ];

    expect(squareMatrixRotation(matrix)).toEqual(rotatedMatrix);
  });
});
```

Listing 237: squareMatrixRotation.js

```
/**
 * @param {*[][]} originalMatrix
 * @return {*[][]}
 */
export default function squareMatrixRotation(originalMatrix) {
  const matrix = originalMatrix.slice();

  // Do top-right/bottom-left diagonal reflection of the matrix.
  for (let rowIndex = 0; rowIndex < matrix.length; rowIndex += 1) {
    for (let columnIndex = rowIndex + 1; columnIndex < matrix.length; columnIndex += 1) {
      // Swap elements.
      [
        matrix[columnIndex][rowIndex],
        matrix[rowIndex][columnIndex],
      ] = [
        matrix[rowIndex][columnIndex],
        matrix[columnIndex][rowIndex],
      ];
    }
  }

  // Do horizontal reflection of the matrix.
  for (let rowIndex = 0; rowIndex < matrix.length; rowIndex += 1) {
    for (let columnIndex = 0; columnIndex < matrix.length / 2; columnIndex += 1) {
      // Swap elements.
      [
        matrix[rowIndex][matrix.length - columnIndex - 1],
        matrix[rowIndex][columnIndex],
      ] = [
        matrix[rowIndex][columnIndex],
        matrix[rowIndex][matrix.length - columnIndex - 1],
      ];
    }
  }

  return matrix;
}
```

Listing 238: btUniquePaths.test.js

```
import btUniquePaths from '../btUniquePaths';

describe('btUniquePaths', () => {
  it('should find the number of unique paths on board', () => {
    expect(btUniquePaths(3, 2)).toBe(3);
    expect(btUniquePaths(7, 3)).toBe(28);
    expect(btUniquePaths(3, 7)).toBe(28);
    expect(btUniquePaths(10, 10)).toBe(48620);
    expect(btUniquePaths(100, 1)).toBe(1);
    expect(btUniquePaths(1, 100)).toBe(1);
  });
});
```

Listing 239: dpUniquePaths.test.js

```
import dpUniquePaths from '../dpUniquePaths';

describe('dpUniquePaths', () => {
  it('should find the number of unique paths on board', () => {
    expect(dpUniquePaths(3, 2)).toBe(3);
    expect(dpUniquePaths(7, 3)).toBe(28);
    expect(dpUniquePaths(3, 7)).toBe(28);
    expect(dpUniquePaths(10, 10)).toBe(48620);
    expect(dpUniquePaths(100, 1)).toBe(1);
    expect(dpUniquePaths(1, 100)).toBe(1);
  });
});
```

Listing 240: uniquePaths.test.js

```
import uniquePaths from '../uniquePaths';

describe('uniquePaths', () => {
  it('should find the number of unique paths on board', () => {
    expect(uniquePaths(3, 2)).toBe(3);
    expect(uniquePaths(7, 3)).toBe(28);
    expect(uniquePaths(3, 7)).toBe(28);
    expect(uniquePaths(10, 10)).toBe(48620);
    expect(uniquePaths(100, 1)).toBe(1);
    expect(uniquePaths(1, 100)).toBe(1);
  });
});
```

Listing 241: btUniquePaths.js

```
/**
 * BACKTRACKING approach of solving Unique Paths problem.
 *
 * @param {number} width - Width of the board.
 * @param {number} height - Height of the board.
 * @param {number[][]} steps - The steps that have been already made.
 * @param {number} uniqueSteps - Total number of unique steps.
 * @return {number} - Number of unique paths.
 */
export default function btUniquePaths(width, height, steps = [[0, 0]], uniqueSteps = 0) {
  // Fetch current position on board.
  const currentPos = steps[steps.length - 1];

  // Check if we've reached the end.
  if (currentPos[0] === width - 1 && currentPos[1] === height - 1) {
    // In case if we've reached the end let's increase total
    // number of unique steps.
    return uniqueSteps + 1;
  }

  // Let's calculate how many unique path we will have
  // by going right and by going down.
  let rightUniqueSteps = 0;
  let downUniqueSteps = 0;

  // Do right step if possible.
  if (currentPos[0] < width - 1) {
    steps.push([
      currentPos[0] + 1,
      currentPos[1],
    ]);

    // Calculate how many unique paths we'll get by moving right.
    rightUniqueSteps = btUniquePaths(width, height, steps, uniqueSteps);

    // BACKTRACK and try another move.
    steps.pop();
  }

  // Do down step if possible.
  if (currentPos[1] < height - 1) {
    steps.push([
      currentPos[0],
      currentPos[1] + 1,
    ]);

    // Calculate how many unique paths we'll get by moving down.
    downUniqueSteps = btUniquePaths(width, height, steps, uniqueSteps);

    // BACKTRACK and try another move.
    steps.pop();
  }

  // Total amount of unique steps will be equal to total amount of
  // unique steps by going right plus total amount of unique steps
  // by going down.
  return rightUniqueSteps + downUniqueSteps;
}
```

Listing 242: dpUniquePaths.js

```
/**
 * DYNAMIC PROGRAMMING approach of solving Unique Paths problem.
 *
 * @param {number} width - Width of the board.
 * @param {number} height - Height of the board.
 * @return {number} - Number of unique paths.
 */
export default function dpUniquePaths(width, height) {
  // Init board.
  const board = Array(height).fill(null).map(() => {
    return Array(width).fill(0);
  });

  // Base case.
  // There is only one way of getting to board[0][any] and
  // there is also only one way of getting to board[any][0].
  // This is because we have a restriction of moving right
  // and down only.
  for (let rowIndex = 0; rowIndex < height; rowIndex += 1) {
    for (let columnIndex = 0; columnIndex < width; columnIndex += 1) {
      if (rowIndex === 0 || columnIndex === 0) {
        board[rowIndex][columnIndex] = 1;
      }
    }
  }

  // Now, since we have this restriction of moving only to the right
  // and down we might say that number of unique paths to the current
  // cell is a sum of numbers of unique paths to the cell above the
  // current one and to the cell to the left of current one.
  for (let rowIndex = 1; rowIndex < height; rowIndex += 1) {
    for (let columnIndex = 1; columnIndex < width; columnIndex += 1) {
      const uniquesFromTop = board[rowIndex - 1][columnIndex];
      const uniquesFromLeft = board[rowIndex][columnIndex - 1];
      board[rowIndex][columnIndex] = uniquesFromTop + uniquesFromLeft;
    }
  }

  return board[height - 1][width - 1];
}
```

Listing 243: uniquePaths.js

```
import pascalTriangle from '../../math/pascal-triangle/pascalTriangle';

/**
 * @param {number} width
 * @param {number} height
 * @return {number}
 */
export default function uniquePaths(width, height) {
  const pascalLine = width + height - 2;
  const pascalLinePosition = Math.min(width, height) - 1;

  return pascalTriangle(pascalLine)[pascalLinePosition];
}
```

Listing 244: BloomFilter.test.js

```
import BloomFilter from '../BloomFilter';

describe('BloomFilter', () => {
  let bloomFilter;
  const people = [
    'Bruce Wayne',
    'Clark Kent',
    'Barry Allen',
  ];

  beforeEach(() => {
    bloomFilter = new BloomFilter();
  });

  it('should have methods named "insert" and "mayContain"', () => {
    expect(typeof bloomFilter.insert).toBe('function');
    expect(typeof bloomFilter.mayContain).toBe('function');
  });

  it('should create a new filter store with the appropriate methods', () => {
    const store = bloomFilter.createStore(18);
    expect(typeof store.getValue).toBe('function');
    expect(typeof store.setValue).toBe('function');
  });

  it('should hash deterministically with all 3 hash functions', () => {
    const str1 = 'apple';

    expect(bloomFilter.hash1(str1)).toEqual(bloomFilter.hash1(str1));
    expect(bloomFilter.hash2(str1)).toEqual(bloomFilter.hash2(str1));
    expect(bloomFilter.hash3(str1)).toEqual(bloomFilter.hash3(str1));

    expect(bloomFilter.hash1(str1)).toBe(14);
    expect(bloomFilter.hash2(str1)).toBe(43);
    expect(bloomFilter.hash3(str1)).toBe(10);

    const str2 = 'orange';

    expect(bloomFilter.hash1(str2)).toEqual(bloomFilter.hash1(str2));
    expect(bloomFilter.hash2(str2)).toEqual(bloomFilter.hash2(str2));
    expect(bloomFilter.hash3(str2)).toEqual(bloomFilter.hash3(str2));

    expect(bloomFilter.hash1(str2)).toBe(0);
    expect(bloomFilter.hash2(str2)).toBe(61);
    expect(bloomFilter.hash3(str2)).toBe(10);
  });

  it('should create an array with 3 hash values', () => {
    expect(bloomFilter.getHashValues('abc').length).toBe(3);
    expect(bloomFilter.getHashValues('abc')).toEqual([66, 63, 54]);
  });

  it('should insert strings correctly and return true when checking for inserted values', () => {
    people.forEach(person => bloomFilter.insert(person));

    expect(bloomFilter.mayContain('Bruce Wayne')).toBe(true);
    expect(bloomFilter.mayContain('Clark Kent')).toBe(true);
    expect(bloomFilter.mayContain('Barry Allen')).toBe(true);

    expect(bloomFilter.mayContain('Tony Stark')).toBe(false);
  });
});
```

Listing 245: BloomFilter.js

```
export default class BloomFilter {
  /**
   * @param {number} size - the size of the storage.
   */
  constructor(size = 100) {
    // Bloom filter size directly affects the likelihood of false positives.
    // The bigger the size the lower the likelihood of false positives.
    this.size = size;
    this.storage = this.createStore(size);
  }

  /**
   * @param {string} item
   */
  insert(item) {
    const hashValues = this.getHashValues(item);

    // Set each hashValue index to true.
    hashValues.forEach(val => this.storage.setValue(val));
  }

  /**
   * @param {string} item
   * @return {boolean}
   */
  mayContain(item) {
    const hashValues = this.getHashValues(item);

    for (let hashIndex = 0; hashIndex < hashValues.length; hashIndex += 1) {
      if (!this.storage.getValue(hashValues[hashIndex])) {
        // We know that the item was definitely not inserted.
        return false;
      }
    }

    // The item may or may not have been inserted.
    return true;
  }

  /**
   * Creates the data store for our filter.
   * We use this method to generate the store in order to
   * encapsulate the data itself and only provide access
   * to the necessary methods.
   *
   * @param {number} size
   * @return {Object}
   */
  createStore(size) {
    const storage = [];

    // Initialize all indexes to false
    for (let storageCellIndex = 0; storageCellIndex < size; storageCellIndex += 1) {
      storage.push(false);
    }

    const storageInterface = {
      getValue(index) {
        return storage[index];
      },
      setValue(index) {
        storage[index] = true;
      },
    };

    return storageInterface;
  }

  /**
   * @param {string} item
   * @return {number}
   */
  hash1(item) {
    let hash = 0;

    for (let charIndex = 0; charIndex < item.length; charIndex += 1) {
      const char = item.charCodeAt(charIndex);
      hash = (hash << 5) + hash + char;
      hash &= hash; // Convert to 32bit integer
```

```
      hash = Math.abs(hash);
    }

    return hash % this.size;
  }

  /**
   * @param {string} item
   * @return {number}
   */
  hash2(item) {
    let hash = 5381;

    for (let charIndex = 0; charIndex < item.length; charIndex += 1) {
      const char = item.charCodeAt(charIndex);
      hash = (hash << 5) + hash + char; /* hash * 33 + c */
    }

    return Math.abs(hash % this.size);
  }

  /**
   * @param {string} item
   * @return {number}
   */
  hash3(item) {
    let hash = 0;

    for (let charIndex = 0; charIndex < item.length; charIndex += 1) {
      const char = item.charCodeAt(charIndex);
      hash = (hash << 5) - hash;
      hash += char;
      hash &= hash; // Convert to 32bit integer
    }

    return Math.abs(hash % this.size);
  }

  /**
   * Runs all 3 hash functions on the input and returns an array of results.
   *
   * @param {string} item
   * @return {number[]}
   */
  getHashValues(item) {
    return [
      this.hash1(item),
      this.hash2(item),
      this.hash3(item),
    ];
  }
}
```

Listing 246: DisjointSet.test.js

```
import DisjointSet from '../DisjointSet';

describe('DisjointSet', () => {
  it('should throw error when trying to union and check not existing sets', () => {
    function mergeNotExistingSets() {
      const disjointSet = new DisjointSet();

      disjointSet.union('A', 'B');
    }

    function checkNotExistingSets() {
      const disjointSet = new DisjointSet();

      disjointSet.inSameSet('A', 'B');
    }

    expect(mergeNotExistingSets).toThrow();
    expect(checkNotExistingSets).toThrow();
  });

  it('should do basic manipulations on disjoint set', () => {
    const disjointSet = new DisjointSet();

    expect(disjointSet.find('A')).toBeNull();
    expect(disjointSet.find('B')).toBeNull();

    disjointSet.makeSet('A');

    expect(disjointSet.find('A')).toBe('A');
    expect(disjointSet.find('B')).toBeNull();

    disjointSet.makeSet('B');

    expect(disjointSet.find('A')).toBe('A');
    expect(disjointSet.find('B')).toBe('B');

    disjointSet.makeSet('C');

    expect(disjointSet.inSameSet('A', 'B')).toBe(false);

    disjointSet.union('A', 'B');

    expect(disjointSet.find('A')).toBe('A');
    expect(disjointSet.find('B')).toBe('A');
    expect(disjointSet.inSameSet('A', 'B')).toBe(true);
    expect(disjointSet.inSameSet('B', 'A')).toBe(true);
    expect(disjointSet.inSameSet('A', 'C')).toBe(false);

    disjointSet.union('A', 'A');

    disjointSet.union('B', 'C');

    expect(disjointSet.find('A')).toBe('A');
    expect(disjointSet.find('B')).toBe('A');
    expect(disjointSet.find('C')).toBe('A');

    expect(disjointSet.inSameSet('A', 'B')).toBe(true);
    expect(disjointSet.inSameSet('B', 'C')).toBe(true);
    expect(disjointSet.inSameSet('A', 'C')).toBe(true);

    disjointSet
      .makeSet('E')
      .makeSet('F')
      .makeSet('G')
      .makeSet('H')
      .makeSet('I');

    disjointSet
      .union('E', 'F')
      .union('F', 'G')
      .union('G', 'H')
      .union('H', 'I');

    expect(disjointSet.inSameSet('A', 'I')).toBe(false);
    expect(disjointSet.inSameSet('E', 'I')).toBe(true);

    disjointSet.union('I', 'C');

    expect(disjointSet.find('I')).toBe('E');
```

```
    expect(disjointSet.inSameSet('A', 'I')).toBe(true);
  });

  it('should union smaller set with bigger one making bigger one to be new root', () => {
    const disjointSet = new DisjointSet();

    disjointSet
      .makeSet('A')
      .makeSet('B')
      .makeSet('C')
      .union('B', 'C')
      .union('A', 'C');

    expect(disjointSet.find('A')).toBe('B');
  });

  it('should do basic manipulations on disjoint set with custom key extractor', () => {
    const keyExtractor = value => value.key;

    const disjointSet = new DisjointSet(keyExtractor);

    const itemA = { key: 'A', value: 1 };
    const itemB = { key: 'B', value: 2 };
    const itemC = { key: 'C', value: 3 };

    expect(disjointSet.find(itemA)).toBeNull();
    expect(disjointSet.find(itemB)).toBeNull();

    disjointSet.makeSet(itemA);

    expect(disjointSet.find(itemA)).toBe('A');
    expect(disjointSet.find(itemB)).toBeNull();

    disjointSet.makeSet(itemB);

    expect(disjointSet.find(itemA)).toBe('A');
    expect(disjointSet.find(itemB)).toBe('B');

    disjointSet.makeSet(itemC);

    expect(disjointSet.inSameSet(itemA, itemB)).toBe(false);

    disjointSet.union(itemA, itemB);

    expect(disjointSet.find(itemA)).toBe('A');
    expect(disjointSet.find(itemB)).toBe('A');
    expect(disjointSet.inSameSet(itemA, itemB)).toBe(true);
    expect(disjointSet.inSameSet(itemB, itemA)).toBe(true);
    expect(disjointSet.inSameSet(itemA, itemC)).toBe(false);

    disjointSet.union(itemA, itemC);

    expect(disjointSet.find(itemA)).toBe('A');
    expect(disjointSet.find(itemB)).toBe('A');
    expect(disjointSet.find(itemC)).toBe('A');

    expect(disjointSet.inSameSet(itemA, itemB)).toBe(true);
    expect(disjointSet.inSameSet(itemB, itemC)).toBe(true);
    expect(disjointSet.inSameSet(itemA, itemC)).toBe(true);
  });
});
```

Listing 247: DisjointSetItem.test.js

```
import DisjointSetItem from '../DisjointSetItem';

describe('DisjointSetItem', () => {
  it('should do basic manipulation with disjoint set item', () => {
    const itemA = new DisjointSetItem('A');
    const itemB = new DisjointSetItem('B');
    const itemC = new DisjointSetItem('C');
    const itemD = new DisjointSetItem('D');

    expect(itemA.getRank()).toBe(0);
    expect(itemA.getChildren()).toEqual([]);
    expect(itemA.getKey()).toBe('A');
    expect(itemA.getRoot()).toEqual(itemA);
    expect(itemA.isRoot()).toBe(true);
    expect(itemB.isRoot()).toBe(true);

    itemA.addChild(itemB);
    itemD.setParent(itemC);

    expect(itemA.getRank()).toBe(1);
    expect(itemC.getRank()).toBe(1);

    expect(itemB.getRank()).toBe(0);
    expect(itemD.getRank()).toBe(0);

    expect(itemA.getChildren().length).toBe(1);
    expect(itemC.getChildren().length).toBe(1);

    expect(itemA.getChildren()[0]).toEqual(itemB);
    expect(itemC.getChildren()[0]).toEqual(itemD);

    expect(itemB.getChildren().length).toBe(0);
    expect(itemD.getChildren().length).toBe(0);

    expect(itemA.getRoot()).toEqual(itemA);
    expect(itemB.getRoot()).toEqual(itemA);

    expect(itemC.getRoot()).toEqual(itemC);
    expect(itemD.getRoot()).toEqual(itemC);

    expect(itemA.isRoot()).toBe(true);
    expect(itemB.isRoot()).toBe(false);
    expect(itemC.isRoot()).toBe(true);
    expect(itemD.isRoot()).toBe(false);

    itemA.addChild(itemC);

    expect(itemA.isRoot()).toBe(true);
    expect(itemB.isRoot()).toBe(false);
    expect(itemC.isRoot()).toBe(false);
    expect(itemD.isRoot()).toBe(false);

    expect(itemA.getRank()).toEqual(3);
    expect(itemB.getRank()).toEqual(0);
    expect(itemC.getRank()).toEqual(1);
  });

  it('should do basic manipulation with disjoint set item with custom key extractor', () => {
    const keyExtractor = (value) => {
      return value.key;
    };

    const itemA = new DisjointSetItem({ key: 'A', value: 1 }, keyExtractor);
    const itemB = new DisjointSetItem({ key: 'B', value: 2 }, keyExtractor);
    const itemC = new DisjointSetItem({ key: 'C', value: 3 }, keyExtractor);
    const itemD = new DisjointSetItem({ key: 'D', value: 4 }, keyExtractor);

    expect(itemA.getRank()).toBe(0);
    expect(itemA.getChildren()).toEqual([]);
    expect(itemA.getKey()).toBe('A');
    expect(itemA.getRoot()).toEqual(itemA);
    expect(itemA.isRoot()).toBe(true);
    expect(itemB.isRoot()).toBe(true);

    itemA.addChild(itemB);
    itemD.setParent(itemC);

    expect(itemA.getRank()).toBe(1);
    expect(itemC.getRank()).toBe(1);
```

```
    expect(itemB.getRank()).toBe(0);
    expect(itemD.getRank()).toBe(0);

    expect(itemA.getChildren().length).toBe(1);
    expect(itemC.getChildren().length).toBe(1);

    expect(itemA.getChildren()[0]).toEqual(itemB);
    expect(itemC.getChildren()[0]).toEqual(itemD);

    expect(itemB.getChildren().length).toBe(0);
    expect(itemD.getChildren().length).toBe(0);

    expect(itemA.getRoot()).toEqual(itemA);
    expect(itemB.getRoot()).toEqual(itemA);

    expect(itemC.getRoot()).toEqual(itemC);
    expect(itemD.getRoot()).toEqual(itemC);

    expect(itemA.isRoot()).toBe(true);
    expect(itemB.isRoot()).toBe(false);
    expect(itemC.isRoot()).toBe(true);
    expect(itemD.isRoot()).toBe(false);

    itemA.addChild(itemC);

    expect(itemA.isRoot()).toBe(true);
    expect(itemB.isRoot()).toBe(false);
    expect(itemC.isRoot()).toBe(false);
    expect(itemD.isRoot()).toBe(false);

    expect(itemA.getRank()).toEqual(3);
    expect(itemB.getRank()).toEqual(0);
    expect(itemC.getRank()).toEqual(1);
  });
});
```

Listing 248: DisjointSet.js

```
import DisjointSetItem from './DisjointSetItem';

export default class DisjointSet {
  /**
   * @param {function(value: *)} [keyCallback]
   */
  constructor(keyCallback) {
    this.keyCallback = keyCallback;
    this.items = {};
  }

  /**
   * @param {*} itemValue
   * @return {DisjointSet}
   */
  makeSet(itemValue) {
    const disjointSetItem = new DisjointSetItem(itemValue, this.keyCallback);

    if (!this.items[disjointSetItem.getKey()]) {
      // Add new item only in case if it not presented yet.
      this.items[disjointSetItem.getKey()] = disjointSetItem;
    }

    return this;
  }

  /**
   * Find set representation node.
   *
   * @param {*} itemValue
   * @return {(string|null)}
   */
  find(itemValue) {
    const templateDisjointItem = new DisjointSetItem(itemValue, this.keyCallback);

    // Try to find item itself;
    const requiredDisjointItem = this.items[templateDisjointItem.getKey()];

    if (!requiredDisjointItem) {
      return null;
    }

    return requiredDisjointItem.getRoot().getKey();
  }

  /**
   * Union by rank.
   *
   * @param {*} valueA
   * @param {*} valueB
   * @return {DisjointSet}
   */
  union(valueA, valueB) {
    const rootKeyA = this.find(valueA);
    const rootKeyB = this.find(valueB);

    if (rootKeyA === null || rootKeyB === null) {
      throw new Error('One or two values are not in sets');
    }

    if (rootKeyA === rootKeyB) {
      // In case if both elements are already in the same set then just return its key.
      return this;
    }

    const rootA = this.items[rootKeyA];
    const rootB = this.items[rootKeyB];

    if (rootA.getRank() < rootB.getRank()) {
      // If rootB's tree is bigger then make rootB to be a new root.
      rootB.addChild(rootA);

      return this;
    }

    // If rootA's tree is bigger then make rootA to be a new root.
    rootA.addChild(rootB);

    return this;
```

```
  }

  /**
   * @param {*} valueA
   * @param {*} valueB
   * @return {boolean}
   */
  inSameSet(valueA, valueB) {
    const rootKeyA = this.find(valueA);
    const rootKeyB = this.find(valueB);

    if (rootKeyA === null || rootKeyB === null) {
      throw new Error('One or two values are not in sets');
    }

    return rootKeyA === rootKeyB;
  }
}
```

Listing 249: DisjointSetItem.js

```
export default class DisjointSetItem {
  /**
   * @param {*} value
   * @param {function(value: *)} [keyCallback]
   */
  constructor(value, keyCallback) {
    this.value = value;
    this.keyCallback = keyCallback;
    /** @var {DisjointSetItem} this.parent */
    this.parent = null;
    this.children = {};
  }

  /**
   * @return {*}
   */
  getKey() {
    // Allow user to define custom key generator.
    if (this.keyCallback) {
      return this.keyCallback(this.value);
    }

    // Otherwise use value as a key by default.
    return this.value;
  }

  /**
   * @return {DisjointSetItem}
   */
  getRoot() {
    return this.isRoot() ? this : this.parent.getRoot();
  }

  /**
   * @return {boolean}
   */
  isRoot() {
    return this.parent === null;
  }

  /**
   * Rank basically means the number of all ancestors.
   *
   * @return {number}
   */
  getRank() {
    if (this.getChildren().length === 0) {
      return 0;
    }

    let rank = 0;

    /** @var {DisjointSetItem} child */
    this.getChildren().forEach((child) => {
      // Count child itself.
      rank += 1;

      // Also add all children of current child.
      rank += child.getRank();
    });

    return rank;
  }

  /**
   * @return {DisjointSetItem[]}
   */
  getChildren() {
    return Object.values(this.children);
  }

  /**
   * @param {DisjointSetItem} parentItem
   * @param {boolean} forceSettingParentChild
   * @return {DisjointSetItem}
   */
  setParent(parentItem, forceSettingParentChild = true) {
    this.parent = parentItem;
    if (forceSettingParentChild) {
```

```
      parentItem.addChild(this);
    }

    return this;
  }

  /**
   * @param {DisjointSetItem} childItem
   * @return {DisjointSetItem}
   */
  addChild(childItem) {
    this.children[childItem.getKey()] = childItem;
    childItem.setParent(this, false);

    return this;
  }
}
```

Listing 250: DoublyLinkedList.test.js

```
import DoublyLinkedList from '../DoublyLinkedList';

describe('DoublyLinkedList', () => {
  it('should create empty linked list', () => {
    const linkedList = new DoublyLinkedList();
    expect(linkedList.toString()).toBe('');
  });

  it('should append node to linked list', () => {
    const linkedList = new DoublyLinkedList();

    expect(linkedList.head).toBeNull();
    expect(linkedList.tail).toBeNull();

    linkedList.append(1);
    linkedList.append(2);

    expect(linkedList.head.next.value).toBe(2);
    expect(linkedList.tail.previous.value).toBe(1);
    expect(linkedList.toString()).toBe('1,2');
  });

  it('should prepend node to linked list', () => {
    const linkedList = new DoublyLinkedList();

    linkedList.prepend(2);
    expect(linkedList.head.toString()).toBe('2');
    expect(linkedList.tail.toString()).toBe('2');

    linkedList.append(1);
    linkedList.prepend(3);

    expect(linkedList.head.next.next.previous).toBe(linkedList.head.next);
    expect(linkedList.tail.previous.next).toBe(linkedList.tail);
    expect(linkedList.tail.previous.value).toBe(2);
    expect(linkedList.toString()).toBe('3,2,1');
  });

  it('should create linked list from array', () => {
    const linkedList = new DoublyLinkedList();
    linkedList.fromArray([1, 1, 2, 3, 3, 3, 4, 5]);

    expect(linkedList.toString()).toBe('1,1,2,3,3,3,4,5');
  });

  it('should delete node by value from linked list', () => {
    const linkedList = new DoublyLinkedList();

    expect(linkedList.delete(5)).toBeNull();

    linkedList.append(1);
    linkedList.append(1);
    linkedList.append(2);
    linkedList.append(3);
    linkedList.append(3);
    linkedList.append(3);
    linkedList.append(4);
    linkedList.append(5);

    expect(linkedList.head.toString()).toBe('1');
    expect(linkedList.tail.toString()).toBe('5');

    const deletedNode = linkedList.delete(3);
    expect(deletedNode.value).toBe(3);
    expect(linkedList.tail.previous.previous.value).toBe(2);
    expect(linkedList.toString()).toBe('1,1,2,4,5');

    linkedList.delete(3);
    expect(linkedList.toString()).toBe('1,1,2,4,5');

    linkedList.delete(1);
    expect(linkedList.toString()).toBe('2,4,5');

    expect(linkedList.head.toString()).toBe('2');
    expect(linkedList.head.next.next).toBe(linkedList.tail);
    expect(linkedList.tail.previous.previous).toBe(linkedList.head);
    expect(linkedList.tail.toString()).toBe('5');

    linkedList.delete(5);
```

```
    expect(linkedList.toString()).toBe('2,4');

    expect(linkedList.head.toString()).toBe('2');
    expect(linkedList.tail.toString()).toBe('4');

    linkedList.delete(4);
    expect(linkedList.toString()).toBe('2');

    expect(linkedList.head.toString()).toBe('2');
    expect(linkedList.tail.toString()).toBe('2');
    expect(linkedList.head).toBe(linkedList.tail);

    linkedList.delete(2);
    expect(linkedList.toString()).toBe('');
  });

  it('should delete linked list tail', () => {
    const linkedList = new DoublyLinkedList();

    expect(linkedList.deleteTail()).toBeNull();

    linkedList.append(1);
    linkedList.append(2);
    linkedList.append(3);

    expect(linkedList.head.toString()).toBe('1');
    expect(linkedList.tail.toString()).toBe('3');

    const deletedNode1 = linkedList.deleteTail();

    expect(deletedNode1.value).toBe(3);
    expect(linkedList.toString()).toBe('1,2');
    expect(linkedList.head.toString()).toBe('1');
    expect(linkedList.tail.toString()).toBe('2');

    const deletedNode2 = linkedList.deleteTail();

    expect(deletedNode2.value).toBe(2);
    expect(linkedList.toString()).toBe('1');
    expect(linkedList.head.toString()).toBe('1');
    expect(linkedList.tail.toString()).toBe('1');

    const deletedNode3 = linkedList.deleteTail();

    expect(deletedNode3.value).toBe(1);
    expect(linkedList.toString()).toBe('');
    expect(linkedList.head).toBeNull();
    expect(linkedList.tail).toBeNull();
  });

  it('should delete linked list head', () => {
    const linkedList = new DoublyLinkedList();

    expect(linkedList.deleteHead()).toBeNull();

    linkedList.append(1);
    linkedList.append(2);

    expect(linkedList.head.toString()).toBe('1');
    expect(linkedList.tail.toString()).toBe('2');

    const deletedNode1 = linkedList.deleteHead();

    expect(deletedNode1.value).toBe(1);
    expect(linkedList.head.previous).toBeNull();
    expect(linkedList.toString()).toBe('2');
    expect(linkedList.head.toString()).toBe('2');
    expect(linkedList.tail.toString()).toBe('2');

    const deletedNode2 = linkedList.deleteHead();

    expect(deletedNode2.value).toBe(2);
    expect(linkedList.toString()).toBe('');
    expect(linkedList.head).toBeNull();
    expect(linkedList.tail).toBeNull();
  });

  it('should be possible to store objects in the list and to print them out', () => {
    const linkedList = new DoublyLinkedList();

    const nodeValue1 = { value: 1, key: 'key1' };
    const nodeValue2 = { value: 2, key: 'key2' };
```

```
    linkedList
      .append(nodeValue1)
      .prepend(nodeValue2);

    const nodeStringifier = value => `${value.key}:${value.value}`;

    expect(linkedList.toString(nodeStringifier)).toBe('key2:2,key1:1');
  });

  it('should find node by value', () => {
    const linkedList = new DoublyLinkedList();

    expect(linkedList.find({ value: 5 })).toBeNull();

    linkedList.append(1);
    expect(linkedList.find({ value: 1 })).toBeDefined();

    linkedList
      .append(2)
      .append(3);

    const node = linkedList.find({ value: 2 });

    expect(node.value).toBe(2);
    expect(linkedList.find({ value: 5 })).toBeNull();
  });

  it('should find node by callback', () => {
    const linkedList = new DoublyLinkedList();

    linkedList
      .append({ value: 1, key: 'test1' })
      .append({ value: 2, key: 'test2' })
      .append({ value: 3, key: 'test3' });

    const node = linkedList.find({ callback: value => value.key === 'test2' });

    expect(node).toBeDefined();
    expect(node.value.value).toBe(2);
    expect(node.value.key).toBe('test2');
    expect(linkedList.find({ callback: value => value.key === 'test5' })).toBeNull();
  });

  it('should find node by means of custom compare function', () => {
    const comparatorFunction = (a, b) => {
      if (a.customValue === b.customValue) {
        return 0;
      }

      return a.customValue < b.customValue ? -1 : 1;
    };

    const linkedList = new DoublyLinkedList(comparatorFunction);

    linkedList
      .append({ value: 1, customValue: 'test1' })
      .append({ value: 2, customValue: 'test2' })
      .append({ value: 3, customValue: 'test3' });

    const node = linkedList.find({
      value: { value: 2, customValue: 'test2' },
    });

    expect(node).toBeDefined();
    expect(node.value.value).toBe(2);
    expect(node.value.customValue).toBe('test2');
    expect(linkedList.find({ value: 2, customValue: 'test5' })).toBeNull();
  });

  it('should reverse linked list', () => {
    const linkedList = new DoublyLinkedList();

    // Add test values to linked list.
    linkedList
      .append(1)
      .append(2)
      .append(3)
      .append(4);

    expect(linkedList.toString()).toBe('1,2,3,4');
    expect(linkedList.head.value).toBe(1);
```

```
    expect(linkedList.tail.value).toBe(4);

    // Reverse linked list.
    linkedList.reverse();

    expect(linkedList.toString()).toBe('4,3,2,1');

    expect(linkedList.head.previous).toBeNull();
    expect(linkedList.head.value).toBe(4);
    expect(linkedList.head.next.value).toBe(3);
    expect(linkedList.head.next.next.value).toBe(2);
    expect(linkedList.head.next.next.next.value).toBe(1);

    expect(linkedList.tail.next).toBeNull();
    expect(linkedList.tail.value).toBe(1);
    expect(linkedList.tail.previous.value).toBe(2);
    expect(linkedList.tail.previous.previous.value).toBe(3);
    expect(linkedList.tail.previous.previous.previous.value).toBe(4);

    // Reverse linked list back to initial state.
    linkedList.reverse();

    expect(linkedList.toString()).toBe('1,2,3,4');

    expect(linkedList.head.previous).toBeNull();
    expect(linkedList.head.value).toBe(1);
    expect(linkedList.head.next.value).toBe(2);
    expect(linkedList.head.next.next.value).toBe(3);
    expect(linkedList.head.next.next.next.value).toBe(4);

    expect(linkedList.tail.next).toBeNull();
    expect(linkedList.tail.value).toBe(4);
    expect(linkedList.tail.previous.value).toBe(3);
    expect(linkedList.tail.previous.previous.value).toBe(2);
    expect(linkedList.tail.previous.previous.previous.value).toBe(1);
  });
});
```

Listing 251: DoublyLinkedListNode.test.js

```
import DoublyLinkedListNode from '../DoublyLinkedListNode';

describe('DoublyLinkedListNode', () => {
  it('should create list node with value', () => {
    const node = new DoublyLinkedListNode(1);

    expect(node.value).toBe(1);
    expect(node.next).toBeNull();
    expect(node.previous).toBeNull();
  });

  it('should create list node with object as a value', () => {
    const nodeValue = { value: 1, key: 'test' };
    const node = new DoublyLinkedListNode(nodeValue);

    expect(node.value.value).toBe(1);
    expect(node.value.key).toBe('test');
    expect(node.next).toBeNull();
    expect(node.previous).toBeNull();
  });

  it('should link nodes together', () => {
    const node2 = new DoublyLinkedListNode(2);
    const node1 = new DoublyLinkedListNode(1, node2);
    const node3 = new DoublyLinkedListNode(10, node1, node2);

    expect(node1.next).toBeDefined();
    expect(node1.previous).toBeNull();
    expect(node2.next).toBeNull();
    expect(node2.previous).toBeNull();
    expect(node3.next).toBeDefined();
    expect(node3.previous).toBeDefined();
    expect(node1.value).toBe(1);
    expect(node1.next.value).toBe(2);
    expect(node3.next.value).toBe(1);
    expect(node3.previous.value).toBe(2);
  });

  it('should convert node to string', () => {
    const node = new DoublyLinkedListNode(1);

    expect(node.toString()).toBe('1');

    node.value = 'string value';
    expect(node.toString()).toBe('string value');
  });

  it('should convert node to string with custom stringifier', () => {
    const nodeValue = { value: 1, key: 'test' };
    const node = new DoublyLinkedListNode(nodeValue);
    const toStringCallback = value => `value: ${value.value}, key: ${value.key}`;

    expect(node.toString(toStringCallback)).toBe('value: 1, key: test');
  });
});
```

Listing 252: DoublyLinkedList.js

```
import DoublyLinkedListNode from './DoublyLinkedListNode';
import Comparator from '../../utils/comparator/Comparator';

export default class DoublyLinkedList {
  /**
   * @param {Function} [comparatorFunction]
   */
  constructor(comparatorFunction) {
    /** @var DoublyLinkedListNode */
    this.head = null;

    /** @var DoublyLinkedListNode */
    this.tail = null;

    this.compare = new Comparator(comparatorFunction);
  }

  /**
   * @param {*} value
   * @return {DoublyLinkedList}
   */
  prepend(value) {
    // Make new node to be a head.
    const newNode = new DoublyLinkedListNode(value, this.head);

    // If there is head, then it won't be head anymore.
    // Therefore, make its previous reference to be new node (new head).
    // Then mark the new node as head.
    if (this.head) {
      this.head.previous = newNode;
    }
    this.head = newNode;

    // If there is no tail yet let's make new node a tail.
    if (!this.tail) {
      this.tail = newNode;
    }

    return this;
  }

  /**
   * @param {*} value
   * @return {DoublyLinkedList}
   */
  append(value) {
    const newNode = new DoublyLinkedListNode(value);

    // If there is no head yet let's make new node a head.
    if (!this.head) {
      this.head = newNode;
      this.tail = newNode;

      return this;
    }

    // Attach new node to the end of linked list.
    this.tail.next = newNode;

    // Attach current tail to the new node's previous reference.
    newNode.previous = this.tail;

    // Set new node to be the tail of linked list.
    this.tail = newNode;

    return this;
  }

  /**
   * @param {*} value
   * @return {DoublyLinkedListNode}
   */
  delete(value) {
    if (!this.head) {
      return null;
    }

    let deletedNode = null;
    let currentNode = this.head;
```

```
  while (currentNode) {
    if (this.compare.equal(currentNode.value, value)) {
      deletedNode = currentNode;

      if (deletedNode === this.head) {
        // If HEAD is going to be deleted...

        // Set head to second node, which will become new head.
        this.head = deletedNode.next;

        // Set new head's previous to null.
        if (this.head) {
          this.head.previous = null;
        }

        // If all the nodes in list has same value that is passed as argument
        // then all nodes will get deleted, therefore tail needs to be updated.
        if (deletedNode === this.tail) {
          this.tail = null;
        }
      } else if (deletedNode === this.tail) {
        // If TAIL is going to be deleted...

        // Set tail to second last node, which will become new tail.
        this.tail = deletedNode.previous;
        this.tail.next = null;
      } else {
        // If MIDDLE node is going to be deleted...
        const previousNode = deletedNode.previous;
        const nextNode = deletedNode.next;

        previousNode.next = nextNode;
        nextNode.previous = previousNode;
      }
    }

    currentNode = currentNode.next;
  }

  return deletedNode;
}

/**
 * @param {Object} findParams
 * @param {*} findParams.value
 * @param {function} [findParams.callback]
 * @return {DoublyLinkedListNode}
 */
find({ value = undefined, callback = undefined }) {
  if (!this.head) {
    return null;
  }

  let currentNode = this.head;

  while (currentNode) {
    // If callback is specified then try to find node by callback.
    if (callback && callback(currentNode.value)) {
      return currentNode;
    }

    // If value is specified then try to compare by value..
    if (value !== undefined && this.compare.equal(currentNode.value, value)) {
      return currentNode;
    }

    currentNode = currentNode.next;
  }

  return null;
}

/**
 * @return {DoublyLinkedListNode}
 */
deleteTail() {
  if (!this.tail) {
    // No tail to delete.
    return null;
  }
```

```
  if (this.head === this.tail) {
    // There is only one node in linked list.
    const deletedTail = this.tail;
    this.head = null;
    this.tail = null;

    return deletedTail;
  }

  // If there are many nodes in linked list...
  const deletedTail = this.tail;

  this.tail = this.tail.previous;
  this.tail.next = null;

  return deletedTail;
}

/**
 * @return {DoublyLinkedListNode}
 */
deleteHead() {
  if (!this.head) {
    return null;
  }

  const deletedHead = this.head;

  if (this.head.next) {
    this.head = this.head.next;
    this.head.previous = null;
  } else {
    this.head = null;
    this.tail = null;
  }

  return deletedHead;
}

/**
 * @return {DoublyLinkedListNode[]}
 */
toArray() {
  const nodes = [];

  let currentNode = this.head;
  while (currentNode) {
    nodes.push(currentNode);
    currentNode = currentNode.next;
  }

  return nodes;
}

/**
 * @param {*[]} values - Array of values that need to be converted to linked list.
 * @return {DoublyLinkedList}
 */
fromArray(values) {
  values.forEach(value => this.append(value));

  return this;
}

/**
 * @param {function} [callback]
 * @return {string}
 */
toString(callback) {
  return this.toArray().map(node => node.toString(callback)).toString();
}

/**
 * Reverse a linked list.
 * @returns {DoublyLinkedList}
 */
reverse() {
  let currNode = this.head;
  let prevNode = null;
  let nextNode = null;

  while (currNode) {
```

```
      // Store next node.
      nextNode = currNode.next;
      prevNode = currNode.previous;

      // Change next node of the current node so it would link to previous node.
      currNode.next = prevNode;
      currNode.previous = nextNode;

      // Move prevNode and currNode nodes one step forward.
      prevNode = currNode;
      currNode = nextNode;
    }

    // Reset head and tail.
    this.tail = this.head;
    this.head = prevNode;

    return this;
  }
}
```

Listing 253: DoublyLinkedListNode.js

```
export default class DoublyLinkedListNode {
  constructor(value, next = null, previous = null) {
    this.value = value;
    this.next = next;
    this.previous = previous;
  }

  toString(callback) {
    return callback ? callback(this.value) : `${this.value}`;
  }
}
```

Listing 254: Graph.test.js

```
import Graph from '../Graph';
import GraphVertex from '../GraphVertex';
import GraphEdge from '../GraphEdge';

describe('Graph', () => {
  it('should add vertices to graph', () => {
    const graph = new Graph();

    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');

    graph
      .addVertex(vertexA)
      .addVertex(vertexB);

    expect(graph.toString()).toBe('A,B');
    expect(graph.getVertexByKey(vertexA.getKey())).toEqual(vertexA);
    expect(graph.getVertexByKey(vertexB.getKey())).toEqual(vertexB);
  });

  it('should add edges to undirected graph', () => {
    const graph = new Graph();

    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');

    const edgeAB = new GraphEdge(vertexA, vertexB);

    graph.addEdge(edgeAB);

    expect(graph.getAllVertices().length).toBe(2);
    expect(graph.getAllVertices()[0]).toEqual(vertexA);
    expect(graph.getAllVertices()[1]).toEqual(vertexB);

    const graphVertexA = graph.getVertexByKey(vertexA.getKey());
    const graphVertexB = graph.getVertexByKey(vertexB.getKey());

    expect(graph.toString()).toBe('A,B');
    expect(graphVertexA).toBeDefined();
    expect(graphVertexB).toBeDefined();

    expect(graph.getVertexByKey('not existing')).toBeUndefined();

    expect(graphVertexA.getNeighbors().length).toBe(1);
    expect(graphVertexA.getNeighbors()[0]).toEqual(vertexB);
    expect(graphVertexA.getNeighbors()[0]).toEqual(graphVertexB);

    expect(graphVertexB.getNeighbors().length).toBe(1);
    expect(graphVertexB.getNeighbors()[0]).toEqual(vertexA);
    expect(graphVertexB.getNeighbors()[0]).toEqual(graphVertexA);
  });

  it('should add edges to directed graph', () => {
    const graph = new Graph(true);

    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');

    const edgeAB = new GraphEdge(vertexA, vertexB);

    graph.addEdge(edgeAB);

    const graphVertexA = graph.getVertexByKey(vertexA.getKey());
    const graphVertexB = graph.getVertexByKey(vertexB.getKey());

    expect(graph.toString()).toBe('A,B');
    expect(graphVertexA).toBeDefined();
    expect(graphVertexB).toBeDefined();

    expect(graphVertexA.getNeighbors().length).toBe(1);
    expect(graphVertexA.getNeighbors()[0]).toEqual(vertexB);
    expect(graphVertexA.getNeighbors()[0]).toEqual(graphVertexB);

    expect(graphVertexB.getNeighbors().length).toBe(0);
  });

  it('should find edge by vertices in undirected graph', () => {
    const graph = new Graph();
```

```
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');

    const edgeAB = new GraphEdge(vertexA, vertexB, 10);

    graph.addEdge(edgeAB);

    const graphEdgeAB = graph.findEdge(vertexA, vertexB);
    const graphEdgeBA = graph.findEdge(vertexB, vertexA);
    const graphEdgeAC = graph.findEdge(vertexA, vertexC);
    const graphEdgeCA = graph.findEdge(vertexC, vertexA);

    expect(graphEdgeAC).toBeNull();
    expect(graphEdgeCA).toBeNull();
    expect(graphEdgeAB).toEqual(edgeAB);
    expect(graphEdgeBA).toEqual(edgeAB);
    expect(graphEdgeAB.weight).toBe(10);
  });

  it('should find edge by vertices in directed graph', () => {
    const graph = new Graph(true);

    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');

    const edgeAB = new GraphEdge(vertexA, vertexB, 10);

    graph.addEdge(edgeAB);

    const graphEdgeAB = graph.findEdge(vertexA, vertexB);
    const graphEdgeBA = graph.findEdge(vertexB, vertexA);
    const graphEdgeAC = graph.findEdge(vertexA, vertexC);
    const graphEdgeCA = graph.findEdge(vertexC, vertexA);

    expect(graphEdgeAC).toBeNull();
    expect(graphEdgeCA).toBeNull();
    expect(graphEdgeBA).toBeNull();
    expect(graphEdgeAB).toEqual(edgeAB);
    expect(graphEdgeAB.weight).toBe(10);
  });

  it('should return vertex neighbors', () => {
    const graph = new Graph(true);

    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeAC = new GraphEdge(vertexA, vertexC);

    graph
      .addEdge(edgeAB)
      .addEdge(edgeAC);

    const neighbors = graph.getNeighbors(vertexA);

    expect(neighbors.length).toBe(2);
    expect(neighbors[0]).toEqual(vertexB);
    expect(neighbors[1]).toEqual(vertexC);
  });

  it('should throw an error when trying to add edge twice', () => {
    function addSameEdgeTwice() {
      const graph = new Graph(true);

      const vertexA = new GraphVertex('A');
      const vertexB = new GraphVertex('B');

      const edgeAB = new GraphEdge(vertexA, vertexB);

      graph
        .addEdge(edgeAB)
        .addEdge(edgeAB);
    }

    expect(addSameEdgeTwice).toThrow();
  });

  it('should return the list of all added edges', () => {
```

```
  const graph = new Graph(true);

  const vertexA = new GraphVertex('A');
  const vertexB = new GraphVertex('B');
  const vertexC = new GraphVertex('C');

  const edgeAB = new GraphEdge(vertexA, vertexB);
  const edgeBC = new GraphEdge(vertexB, vertexC);

  graph
    .addEdge(edgeAB)
    .addEdge(edgeBC);

  const edges = graph.getAllEdges();

  expect(edges.length).toBe(2);
  expect(edges[0]).toEqual(edgeAB);
  expect(edges[1]).toEqual(edgeBC);
});

it('should calculate total graph weight for default graph', () => {
  const graph = new Graph();

  const vertexA = new GraphVertex('A');
  const vertexB = new GraphVertex('B');
  const vertexC = new GraphVertex('C');
  const vertexD = new GraphVertex('D');

  const edgeAB = new GraphEdge(vertexA, vertexB);
  const edgeBC = new GraphEdge(vertexB, vertexC);
  const edgeCD = new GraphEdge(vertexC, vertexD);
  const edgeAD = new GraphEdge(vertexA, vertexD);

  graph
    .addEdge(edgeAB)
    .addEdge(edgeBC)
    .addEdge(edgeCD)
    .addEdge(edgeAD);

  expect(graph.getWeight()).toBe(0);
});

it('should calculate total graph weight for weighted graph', () => {
  const graph = new Graph();

  const vertexA = new GraphVertex('A');
  const vertexB = new GraphVertex('B');
  const vertexC = new GraphVertex('C');
  const vertexD = new GraphVertex('D');

  const edgeAB = new GraphEdge(vertexA, vertexB, 1);
  const edgeBC = new GraphEdge(vertexB, vertexC, 2);
  const edgeCD = new GraphEdge(vertexC, vertexD, 3);
  const edgeAD = new GraphEdge(vertexA, vertexD, 4);

  graph
    .addEdge(edgeAB)
    .addEdge(edgeBC)
    .addEdge(edgeCD)
    .addEdge(edgeAD);

  expect(graph.getWeight()).toBe(10);
});

it('should be possible to delete edges from graph', () => {
  const graph = new Graph();

  const vertexA = new GraphVertex('A');
  const vertexB = new GraphVertex('B');
  const vertexC = new GraphVertex('C');

  const edgeAB = new GraphEdge(vertexA, vertexB);
  const edgeBC = new GraphEdge(vertexB, vertexC);
  const edgeAC = new GraphEdge(vertexA, vertexC);

  graph
    .addEdge(edgeAB)
    .addEdge(edgeBC)
    .addEdge(edgeAC);

  expect(graph.getAllEdges().length).toBe(3);
```

```
  graph.deleteEdge(edgeAB);

  expect(graph.getAllEdges().length).toBe(2);
  expect(graph.getAllEdges()[0].getKey()).toBe(edgeBC.getKey());
  expect(graph.getAllEdges()[1].getKey()).toBe(edgeAC.getKey());
});

it('should should throw an error when trying to delete not existing edge', () => {
  function deleteNotExistingEdge() {
    const graph = new Graph();

    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeBC = new GraphEdge(vertexB, vertexC);

    graph.addEdge(edgeAB);
    graph.deleteEdge(edgeBC);
  }

  expect(deleteNotExistingEdge).toThrowError();
});

it('should be possible to reverse graph', () => {
  const vertexA = new GraphVertex('A');
  const vertexB = new GraphVertex('B');
  const vertexC = new GraphVertex('C');
  const vertexD = new GraphVertex('D');

  const edgeAB = new GraphEdge(vertexA, vertexB);
  const edgeAC = new GraphEdge(vertexA, vertexC);
  const edgeCD = new GraphEdge(vertexC, vertexD);

  const graph = new Graph(true);
  graph
    .addEdge(edgeAB)
    .addEdge(edgeAC)
    .addEdge(edgeCD);

  expect(graph.toString()).toBe('A,B,C,D');
  expect(graph.getAllEdges().length).toBe(3);
  expect(graph.getNeighbors(vertexA).length).toBe(2);
  expect(graph.getNeighbors(vertexA)[0].getKey()).toBe(vertexB.getKey());
  expect(graph.getNeighbors(vertexA)[1].getKey()).toBe(vertexC.getKey());
  expect(graph.getNeighbors(vertexB).length).toBe(0);
  expect(graph.getNeighbors(vertexC).length).toBe(1);
  expect(graph.getNeighbors(vertexC)[0].getKey()).toBe(vertexD.getKey());
  expect(graph.getNeighbors(vertexD).length).toBe(0);

  graph.reverse();

  expect(graph.toString()).toBe('A,B,C,D');
  expect(graph.getAllEdges().length).toBe(3);
  expect(graph.getNeighbors(vertexA).length).toBe(0);
  expect(graph.getNeighbors(vertexB).length).toBe(1);
  expect(graph.getNeighbors(vertexB)[0].getKey()).toBe(vertexA.getKey());
  expect(graph.getNeighbors(vertexC).length).toBe(1);
  expect(graph.getNeighbors(vertexC)[0].getKey()).toBe(vertexA.getKey());
  expect(graph.getNeighbors(vertexD).length).toBe(1);
  expect(graph.getNeighbors(vertexD)[0].getKey()).toBe(vertexC.getKey());
});

it('should return vertices indices', () => {
  const vertexA = new GraphVertex('A');
  const vertexB = new GraphVertex('B');
  const vertexC = new GraphVertex('C');
  const vertexD = new GraphVertex('D');

  const edgeAB = new GraphEdge(vertexA, vertexB);
  const edgeBC = new GraphEdge(vertexB, vertexC);
  const edgeCD = new GraphEdge(vertexC, vertexD);
  const edgeBD = new GraphEdge(vertexB, vertexD);

  const graph = new Graph();
  graph
    .addEdge(edgeAB)
    .addEdge(edgeBC)
    .addEdge(edgeCD)
    .addEdge(edgeBD);
```

```
    const verticesIndices = graph.getVerticesIndices();
    expect(verticesIndices).toEqual({
      A: 0,
      B: 1,
      C: 2,
      D: 3,
    });
  });

  it('should generate adjacency matrix for undirected graph', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeBC = new GraphEdge(vertexB, vertexC);
    const edgeCD = new GraphEdge(vertexC, vertexD);
    const edgeBD = new GraphEdge(vertexB, vertexD);

    const graph = new Graph();
    graph
      .addEdge(edgeAB)
      .addEdge(edgeBC)
      .addEdge(edgeCD)
      .addEdge(edgeBD);

    const adjacencyMatrix = graph.getAdjacencyMatrix();
    expect(adjacencyMatrix).toEqual([
      [Infinity, 0, Infinity, Infinity],
      [0, Infinity, 0, 0],
      [Infinity, 0, Infinity, 0],
      [Infinity, 0, 0, Infinity],
    ]);
  });

  it('should generate adjacency matrix for directed graph', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');
    const vertexD = new GraphVertex('D');

    const edgeAB = new GraphEdge(vertexA, vertexB, 2);
    const edgeBC = new GraphEdge(vertexB, vertexC, 1);
    const edgeCD = new GraphEdge(vertexC, vertexD, 5);
    const edgeBD = new GraphEdge(vertexB, vertexD, 7);

    const graph = new Graph(true);
    graph
      .addEdge(edgeAB)
      .addEdge(edgeBC)
      .addEdge(edgeCD)
      .addEdge(edgeBD);

    const adjacencyMatrix = graph.getAdjacencyMatrix();
    expect(adjacencyMatrix).toEqual([
      [Infinity, 2, Infinity, Infinity],
      [Infinity, Infinity, 1, 7],
      [Infinity, Infinity, Infinity, 5],
      [Infinity, Infinity, Infinity, Infinity],
    ]);
  });
});
```

Listing 255: GraphEdge.test.js

```
import GraphEdge from '../GraphEdge';
import GraphVertex from '../GraphVertex';

describe('GraphEdge', () => {
  it('should create graph edge with default weight', () => {
    const startVertex = new GraphVertex('A');
    const endVertex = new GraphVertex('B');
    const edge = new GraphEdge(startVertex, endVertex);

    expect(edge.getKey()).toBe('A_B');
    expect(edge.toString()).toBe('A_B');
    expect(edge.startVertex).toEqual(startVertex);
    expect(edge.endVertex).toEqual(endVertex);
    expect(edge.weight).toEqual(0);
  });

  it('should create graph edge with predefined weight', () => {
    const startVertex = new GraphVertex('A');
    const endVertex = new GraphVertex('B');
    const edge = new GraphEdge(startVertex, endVertex, 10);

    expect(edge.startVertex).toEqual(startVertex);
    expect(edge.endVertex).toEqual(endVertex);
    expect(edge.weight).toEqual(10);
  });

  it('should be possible to do edge reverse', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const edge = new GraphEdge(vertexA, vertexB, 10);

    expect(edge.startVertex).toEqual(vertexA);
    expect(edge.endVertex).toEqual(vertexB);
    expect(edge.weight).toEqual(10);

    edge.reverse();

    expect(edge.startVertex).toEqual(vertexB);
    expect(edge.endVertex).toEqual(vertexA);
    expect(edge.weight).toEqual(10);
  });
});
```

Listing 256: GraphVertex.test.js

```
import GraphVertex from '../GraphVertex';
import GraphEdge from '../GraphEdge';

describe('GraphVertex', () => {
  it('should throw an error when trying to create vertex without value', () => {
    let vertex = null;

    function createEmptyVertex() {
      vertex = new GraphVertex();
    }

    expect(vertex).toBeNull();
    expect(createEmptyVertex).toThrow();
  });

  it('should create graph vertex', () => {
    const vertex = new GraphVertex('A');

    expect(vertex).toBeDefined();
    expect(vertex.value).toBe('A');
    expect(vertex.toString()).toBe('A');
    expect(vertex.getKey()).toBe('A');
    expect(vertex.edges.toString()).toBe('');
    expect(vertex.getEdges()).toEqual([]);
  });

  it('should add edges to vertex and check if it exists', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    vertexA.addEdge(edgeAB);

    expect(vertexA.hasEdge(edgeAB)).toBe(true);
    expect(vertexB.hasEdge(edgeAB)).toBe(false);
    expect(vertexA.getEdges().length).toBe(1);
    expect(vertexA.getEdges()[0].toString()).toBe('A_B');
  });

  it('should delete edges from vertex', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeAC = new GraphEdge(vertexA, vertexC);
    vertexA
      .addEdge(edgeAB)
      .addEdge(edgeAC);

    expect(vertexA.hasEdge(edgeAB)).toBe(true);
    expect(vertexB.hasEdge(edgeAB)).toBe(false);

    expect(vertexA.hasEdge(edgeAC)).toBe(true);
    expect(vertexC.hasEdge(edgeAC)).toBe(false);

    expect(vertexA.getEdges().length).toBe(2);

    expect(vertexA.getEdges()[0].toString()).toBe('A_B');
    expect(vertexA.getEdges()[1].toString()).toBe('A_C');

    vertexA.deleteEdge(edgeAB);
    expect(vertexA.hasEdge(edgeAB)).toBe(false);
    expect(vertexA.hasEdge(edgeAC)).toBe(true);
    expect(vertexA.getEdges()[0].toString()).toBe('A_C');

    vertexA.deleteEdge(edgeAC);
    expect(vertexA.hasEdge(edgeAB)).toBe(false);
    expect(vertexA.hasEdge(edgeAC)).toBe(false);
    expect(vertexA.getEdges().length).toBe(0);
  });

  it('should delete all edges from vertex', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeAC = new GraphEdge(vertexA, vertexC);
```

```
    vertexA
      .addEdge(edgeAB)
      .addEdge(edgeAC);

    expect(vertexA.hasEdge(edgeAB)).toBe(true);
    expect(vertexB.hasEdge(edgeAB)).toBe(false);

    expect(vertexA.hasEdge(edgeAC)).toBe(true);
    expect(vertexC.hasEdge(edgeAC)).toBe(false);

    expect(vertexA.getEdges().length).toBe(2);

    vertexA.deleteAllEdges();

    expect(vertexA.hasEdge(edgeAB)).toBe(false);
    expect(vertexB.hasEdge(edgeAB)).toBe(false);

    expect(vertexA.hasEdge(edgeAC)).toBe(false);
    expect(vertexC.hasEdge(edgeAC)).toBe(false);

    expect(vertexA.getEdges().length).toBe(0);
  });

  it('should return vertex neighbors in case if current node is start one', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    const edgeAC = new GraphEdge(vertexA, vertexC);
    vertexA
      .addEdge(edgeAB)
      .addEdge(edgeAC);

    expect(vertexB.getNeighbors()).toEqual([]);

    const neighbors = vertexA.getNeighbors();

    expect(neighbors.length).toBe(2);
    expect(neighbors[0]).toEqual(vertexB);
    expect(neighbors[1]).toEqual(vertexC);
  });

  it('should return vertex neighbors in case if current node is end one', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');

    const edgeBA = new GraphEdge(vertexB, vertexA);
    const edgeCA = new GraphEdge(vertexC, vertexA);
    vertexA
      .addEdge(edgeBA)
      .addEdge(edgeCA);

    expect(vertexB.getNeighbors()).toEqual([]);

    const neighbors = vertexA.getNeighbors();

    expect(neighbors.length).toBe(2);
    expect(neighbors[0]).toEqual(vertexB);
    expect(neighbors[1]).toEqual(vertexC);
  });

  it('should check if vertex has specific neighbor', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    vertexA.addEdge(edgeAB);

    expect(vertexA.hasNeighbor(vertexB)).toBe(true);
    expect(vertexA.hasNeighbor(vertexC)).toBe(false);
  });

  it('should edge by vertex', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');
    const vertexC = new GraphVertex('C');

    const edgeAB = new GraphEdge(vertexA, vertexB);
    vertexA.addEdge(edgeAB);
```

```
    expect(vertexA.findEdge(vertexB)).toEqual(edgeAB);
    expect(vertexA.findEdge(vertexC)).toBeNull();
  });

  it('should calculate vertex degree', () => {
    const vertexA = new GraphVertex('A');
    const vertexB = new GraphVertex('B');

    expect(vertexA.getDegree()).toBe(0);

    const edgeAB = new GraphEdge(vertexA, vertexB);
    vertexA.addEdge(edgeAB);

    expect(vertexA.getDegree()).toBe(1);

    const edgeBA = new GraphEdge(vertexB, vertexA);
    vertexA.addEdge(edgeBA);

    expect(vertexA.getDegree()).toBe(2);

    vertexA.addEdge(edgeAB);
    expect(vertexA.getDegree()).toBe(3);

    expect(vertexA.getEdges().length).toEqual(3);
  });
});
```

Listing 257: Graph.js

```
export default class Graph {
  /**
   * @param {boolean} isDirected
   */
  constructor(isDirected = false) {
    this.vertices = {};
    this.edges = {};
    this.isDirected = isDirected;
  }

  /**
   * @param {GraphVertex} newVertex
   * @returns {Graph}
   */
  addVertex(newVertex) {
    this.vertices[newVertex.getKey()] = newVertex;

    return this;
  }

  /**
   * @param {string} vertexKey
   * @returns GraphVertex
   */
  getVertexByKey(vertexKey) {
    return this.vertices[vertexKey];
  }

  /**
   * @param {GraphVertex} vertex
   * @returns {GraphVertex[]}
   */
  getNeighbors(vertex) {
    return vertex.getNeighbors();
  }

  /**
   * @return {GraphVertex[]}
   */
  getAllVertices() {
    return Object.values(this.vertices);
  }

  /**
   * @return {GraphEdge[]}
   */
  getAllEdges() {
    return Object.values(this.edges);
  }

  /**
   * @param {GraphEdge} edge
   * @returns {Graph}
   */
  addEdge(edge) {
    // Try to find and end start vertices.
    let startVertex = this.getVertexByKey(edge.startVertex.getKey());
    let endVertex = this.getVertexByKey(edge.endVertex.getKey());

    // Insert start vertex if it wasn't inserted.
    if (!startVertex) {
      this.addVertex(edge.startVertex);
      startVertex = this.getVertexByKey(edge.startVertex.getKey());
    }

    // Insert end vertex if it wasn't inserted.
    if (!endVertex) {
      this.addVertex(edge.endVertex);
      endVertex = this.getVertexByKey(edge.endVertex.getKey());
    }

    // Check if edge has been already added.
    if (this.edges[edge.getKey()]) {
      throw new Error('Edge has already been added before');
    } else {
      this.edges[edge.getKey()] = edge;
    }

    // Add edge to the vertices.
```

```
    if (this.isDirected) {
      // If graph IS directed then add the edge only to start vertex.
      startVertex.addEdge(edge);
    } else {
      // If graph ISN'T directed then add the edge to both vertices.
      startVertex.addEdge(edge);
      endVertex.addEdge(edge);
    }

    return this;
  }

  /**
   * @param {GraphEdge} edge
   */
  deleteEdge(edge) {
    // Delete edge from the list of edges.
    if (this.edges[edge.getKey()]) {
      delete this.edges[edge.getKey()];
    } else {
      throw new Error('Edge not found in graph');
    }

    // Try to find and end start vertices and delete edge from them.
    const startVertex = this.getVertexByKey(edge.startVertex.getKey());
    const endVertex = this.getVertexByKey(edge.endVertex.getKey());

    startVertex.deleteEdge(edge);
    endVertex.deleteEdge(edge);
  }

  /**
   * @param {GraphVertex} startVertex
   * @param {GraphVertex} endVertex
   * @return {(GraphEdge|null)}
   */
  findEdge(startVertex, endVertex) {
    const vertex = this.getVertexByKey(startVertex.getKey());

    if (!vertex) {
      return null;
    }

    return vertex.findEdge(endVertex);
  }

  /**
   * @return {number}
   */
  getWeight() {
    return this.getAllEdges().reduce((weight, graphEdge) => {
      return weight + graphEdge.weight;
    }, 0);
  }

  /**
   * Reverse all the edges in directed graph.
   * @return {Graph}
   */
  reverse() {
    /** @param {GraphEdge} edge */
    this.getAllEdges().forEach((edge) => {
      // Delete straight edge from graph and from vertices.
      this.deleteEdge(edge);

      // Reverse the edge.
      edge.reverse();

      // Add reversed edge back to the graph and its vertices.
      this.addEdge(edge);
    });

    return this;
  }

  /**
   * @return {object}
   */
  getVerticesIndices() {
    const verticesIndices = {};
    this.getAllVertices().forEach((vertex, index) => {
      verticesIndices[vertex.getKey()] = index;
```

```
    });

    return verticesIndices;
  }

  /**
   * @return {*[][]}
   */
  getAdjacencyMatrix() {
    const vertices = this.getAllVertices();
    const verticesIndices = this.getVerticesIndices();

    // Init matrix with infinities meaning that there is no ways of
    // getting from one vertex to another yet.
    const adjacencyMatrix = Array(vertices.length).fill(null).map(() => {
      return Array(vertices.length).fill(Infinity);
    });

    // Fill the columns.
    vertices.forEach((vertex, vertexIndex) => {
      vertex.getNeighbors().forEach((neighbor) => {
        const neighborIndex = verticesIndices[neighbor.getKey()];
        adjacencyMatrix[vertexIndex][neighborIndex] = this.findEdge(vertex, neighbor).weight;
      });
    });

    return adjacencyMatrix;
  }

  /**
   * @return {string}
   */
  toString() {
    return Object.keys(this.vertices).toString();
  }
}
```

Listing 258: GraphEdge.js

```
export default class GraphEdge {
  /**
   * @param {GraphVertex} startVertex
   * @param {GraphVertex} endVertex
   * @param {number} [weight=1]
   */
  constructor(startVertex, endVertex, weight = 0) {
    this.startVertex = startVertex;
    this.endVertex = endVertex;
    this.weight = weight;
  }

  /**
   * @return {string}
   */
  getKey() {
    const startVertexKey = this.startVertex.getKey();
    const endVertexKey = this.endVertex.getKey();

    return `${startVertexKey}_${endVertexKey}`;
  }

  /**
   * @return {GraphEdge}
   */
  reverse() {
    const tmp = this.startVertex;
    this.startVertex = this.endVertex;
    this.endVertex = tmp;

    return this;
  }

  /**
   * @return {string}
   */
  toString() {
    return this.getKey();
  }
}
```

Listing 259: GraphVertex.js

```
import LinkedList from '../linked-list/LinkedList';

export default class GraphVertex {
  /**
   * @param {*} value
   */
  constructor(value) {
    if (value === undefined) {
      throw new Error('Graph vertex must have a value');
    }

    /**
     * @param {GraphEdge} edgeA
     * @param {GraphEdge} edgeB
     */
    const edgeComparator = (edgeA, edgeB) => {
      if (edgeA.getKey() === edgeB.getKey()) {
        return 0;
      }

      return edgeA.getKey() < edgeB.getKey() ? -1 : 1;
    };

    // Normally you would store string value like vertex name.
    // But generally it may be any object as well
    this.value = value;
    this.edges = new LinkedList(edgeComparator);
  }

  /**
   * @param {GraphEdge} edge
   * @returns {GraphVertex}
   */
  addEdge(edge) {
    this.edges.append(edge);

    return this;
  }

  /**
   * @param {GraphEdge} edge
   */
  deleteEdge(edge) {
    this.edges.delete(edge);
  }

  /**
   * @returns {GraphVertex[]}
   */
  getNeighbors() {
    const edges = this.edges.toArray();

    /** @param {LinkedListNode} node */
    const neighborsConverter = (node) => {
      return node.value.startVertex === this ? node.value.endVertex : node.value.startVertex;
    };

    // Return either start or end vertex.
    // For undirected graphs it is possible that current vertex will be the end one.
    return edges.map(neighborsConverter);
  }

  /**
   * @return {GraphEdge[]}
   */
  getEdges() {
    return this.edges.toArray().map(linkedListNode => linkedListNode.value);
  }

  /**
   * @return {number}
   */
  getDegree() {
    return this.edges.toArray().length;
  }

  /**
   * @param {GraphEdge} requiredEdge
   * @returns {boolean}
```

```
   */
  hasEdge(requiredEdge) {
    const edgeNode = this.edges.find({
      callback: edge => edge === requiredEdge,
    });

    return !!edgeNode;
  }

  /**
   * @param {GraphVertex} vertex
   * @returns {boolean}
   */
  hasNeighbor(vertex) {
    const vertexNode = this.edges.find({
      callback: edge => edge.startVertex === vertex || edge.endVertex === vertex,
    });

    return !!vertexNode;
  }

  /**
   * @param {GraphVertex} vertex
   * @returns {(GraphEdge|null)}
   */
  findEdge(vertex) {
    const edgeFinder = (edge) => {
      return edge.startVertex === vertex || edge.endVertex === vertex;
    };

    const edge = this.edges.find({ callback: edgeFinder });

    return edge ? edge.value : null;
  }

  /**
   * @returns {string}
   */
  getKey() {
    return this.value;
  }

  /**
   * @return {GraphVertex}
   */
  deleteAllEdges() {
    this.getEdges().forEach(edge => this.deleteEdge(edge));

    return this;
  }

  /**
   * @param {function} [callback]
   * @returns {string}
   */
  toString(callback) {
    return callback ? callback(this.value) : `${this.value}`;
  }
}
```

Listing 260: HashTable.test.js

```
import HashTable from '../HashTable';

describe('HashTable', () => {
  it('should create hash table of certain size', () => {
    const defaultHashTable = new HashTable();
    expect(defaultHashTable.buckets.length).toBe(32);

    const biggerHashTable = new HashTable(64);
    expect(biggerHashTable.buckets.length).toBe(64);
  });

  it('should generate proper hash for specified keys', () => {
    const hashTable = new HashTable();

    expect(hashTable.hash('a')).toBe(1);
    expect(hashTable.hash('b')).toBe(2);
    expect(hashTable.hash('abc')).toBe(6);
  });

  it('should set, read and delete data with collisions', () => {
    const hashTable = new HashTable(3);

    expect(hashTable.hash('a')).toBe(1);
    expect(hashTable.hash('b')).toBe(2);
    expect(hashTable.hash('c')).toBe(0);
    expect(hashTable.hash('d')).toBe(1);

    hashTable.set('a', 'sky-old');
    hashTable.set('a', 'sky');
    hashTable.set('b', 'sea');
    hashTable.set('c', 'earth');
    hashTable.set('d', 'ocean');

    expect(hashTable.has('x')).toBe(false);
    expect(hashTable.has('b')).toBe(true);
    expect(hashTable.has('c')).toBe(true);

    const stringifier = value => `${value.key}:${value.value}`;

    expect(hashTable.buckets[0].toString(stringifier)).toBe('c:earth');
    expect(hashTable.buckets[1].toString(stringifier)).toBe('a:sky,d:ocean');
    expect(hashTable.buckets[2].toString(stringifier)).toBe('b:sea');

    expect(hashTable.get('a')).toBe('sky');
    expect(hashTable.get('d')).toBe('ocean');
    expect(hashTable.get('x')).not.toBeDefined();

    hashTable.delete('a');

    expect(hashTable.delete('not-existing')).toBeNull();

    expect(hashTable.get('a')).not.toBeDefined();
    expect(hashTable.get('d')).toBe('ocean');

    hashTable.set('d', 'ocean-new');
    expect(hashTable.get('d')).toBe('ocean-new');
  });

  it('should be possible to add objects to hash table', () => {
    const hashTable = new HashTable();

    hashTable.set('objectKey', { prop1: 'a', prop2: 'b' });

    const object = hashTable.get('objectKey');
    expect(object).toBeDefined();
    expect(object.prop1).toBe('a');
    expect(object.prop2).toBe('b');
  });

  it('should track actual keys', () => {
    const hashTable = new HashTable(3);

    hashTable.set('a', 'sky-old');
    hashTable.set('a', 'sky');
    hashTable.set('b', 'sea');
    hashTable.set('c', 'earth');
    hashTable.set('d', 'ocean');

    expect(hashTable.getKeys()).toEqual(['a', 'b', 'c', 'd']);
```

```
    expect(hashTable.has('a')).toBe(true);
    expect(hashTable.has('x')).toBe(false);

    hashTable.delete('a');

    expect(hashTable.has('a')).toBe(false);
    expect(hashTable.has('b')).toBe(true);
    expect(hashTable.has('x')).toBe(false);
  });
});
```

Listing 261: HashTable.js

```
import LinkedList from '../linked-list/LinkedList';

// Hash table size directly affects on the number of collisions.
// The bigger the hash table size the less collisions you'll get.
// For demonstrating purposes hash table size is small to show how collisions
// are being handled.
const defaultHashTableSize = 32;

export default class HashTable {
  /**
   * @param {number} hashTableSize
   */
  constructor(hashTableSize = defaultHashTableSize) {
    // Create hash table of certain size and fill each bucket with empty linked list.
    this.buckets = Array(hashTableSize).fill(null).map(() => new LinkedList());

    // Just to keep track of all actual keys in a fast way.
    this.keys = {};
  }

  /**
   * Converts key string to hash number.
   *
   * @param {string} key
   * @return {number}
   */
  hash(key) {
    // For simplicity reasons we will just use character codes sum of all characters of the key
    // to calculate the hash.
    //
    // But you may also use more sophisticated approaches like polynomial string hash to reduce the
    // number of collisions:
    //
    // hash = charCodeAt(0) * PRIME^(n-1) + charCodeAt(1) * PRIME^(n-2) + ... + charCodeAt(n-1)
    //
    // where charCodeAt(i) is the i-th character code of the key, n is the length of the key and
    // PRIME is just any prime number like 31.
    const hash = Array.from(key).reduce(
      (hashAccumulator, keySymbol) => (hashAccumulator + keySymbol.charCodeAt(0)),
      0,
    );

    // Reduce hash number so it would fit hash table size.
    return hash % this.buckets.length;
  }

  /**
   * @param {string} key
   * @param {*} value
   */
  set(key, value) {
    const keyHash = this.hash(key);
    this.keys[key] = keyHash;
    const bucketLinkedList = this.buckets[keyHash];
    const node = bucketLinkedList.find({ callback: nodeValue => nodeValue.key === key });

    if (!node) {
      // Insert new node.
      bucketLinkedList.append({ key, value });
    } else {
      // Update value of existing node.
      node.value.value = value;
    }
  }

  /**
   * @param {string} key
   * @return {*}
   */
  delete(key) {
    const keyHash = this.hash(key);
    delete this.keys[key];
    const bucketLinkedList = this.buckets[keyHash];
    const node = bucketLinkedList.find({ callback: nodeValue => nodeValue.key === key });

    if (node) {
      return bucketLinkedList.delete(node.value);
    }
```

```
    return null;
  }

  /**
   * @param {string} key
   * @return {*}
   */
  get(key) {
    const bucketLinkedList = this.buckets[this.hash(key)];
    const node = bucketLinkedList.find({ callback: nodeValue => nodeValue.key === key });

    return node ? node.value.value : undefined;
  }

  /**
   * @param {string} key
   * @return {boolean}
   */
  has(key) {
    return Object.hasOwnProperty.call(this.keys, key);
  }

  /**
   * @return {string[]}
   */
  getKeys() {
    return Object.keys(this.keys);
  }
}
```

Listing 262: Heap.test.js

```
import Heap from '../Heap';

describe('Heap', () => {
  it('should not allow to create instance of the Heap directly', () => {
    const instantiateHeap = () => {
      const heap = new Heap();
      heap.add(5);
    };

    expect(instantiateHeap).toThrow();
  });
});
```

Listing 263: MaxHeap.test.js

```
import MaxHeap from '../MaxHeap';
import Comparator from '../../../utils/comparator/Comparator';

describe('MaxHeap', () => {
  it('should create an empty max heap', () => {
    const maxHeap = new MaxHeap();

    expect(maxHeap).toBeDefined();
    expect(maxHeap.peek()).toBeNull();
    expect(maxHeap.isEmpty()).toBe(true);
  });

  it('should add items to the heap and heapify it up', () => {
    const maxHeap = new MaxHeap();

    maxHeap.add(5);
    expect(maxHeap.isEmpty()).toBe(false);
    expect(maxHeap.peek()).toBe(5);
    expect(maxHeap.toString()).toBe('5');

    maxHeap.add(3);
    expect(maxHeap.peek()).toBe(5);
    expect(maxHeap.toString()).toBe('5,3');

    maxHeap.add(10);
    expect(maxHeap.peek()).toBe(10);
    expect(maxHeap.toString()).toBe('10,3,5');

    maxHeap.add(1);
    expect(maxHeap.peek()).toBe(10);
    expect(maxHeap.toString()).toBe('10,3,5,1');

    maxHeap.add(1);
    expect(maxHeap.peek()).toBe(10);
    expect(maxHeap.toString()).toBe('10,3,5,1,1');

    expect(maxHeap.poll()).toBe(10);
    expect(maxHeap.toString()).toBe('5,3,1,1');

    expect(maxHeap.poll()).toBe(5);
    expect(maxHeap.toString()).toBe('3,1,1');

    expect(maxHeap.poll()).toBe(3);
    expect(maxHeap.toString()).toBe('1,1');
  });

  it('should poll items from the heap and heapify it down', () => {
    const maxHeap = new MaxHeap();

    maxHeap.add(5);
    maxHeap.add(3);
    maxHeap.add(10);
    maxHeap.add(11);
    maxHeap.add(1);

    expect(maxHeap.toString()).toBe('11,10,5,3,1');

    expect(maxHeap.poll()).toBe(11);
    expect(maxHeap.toString()).toBe('10,3,5,1');

    expect(maxHeap.poll()).toBe(10);
    expect(maxHeap.toString()).toBe('5,3,1');

    expect(maxHeap.poll()).toBe(5);
    expect(maxHeap.toString()).toBe('3,1');

    expect(maxHeap.poll()).toBe(3);
    expect(maxHeap.toString()).toBe('1');

    expect(maxHeap.poll()).toBe(1);
    expect(maxHeap.toString()).toBe('');

    expect(maxHeap.poll()).toBeNull();
    expect(maxHeap.toString()).toBe('');
  });

  it('should heapify down through the right branch as well', () => {
    const maxHeap = new MaxHeap();
```

```
    maxHeap.add(3);
    maxHeap.add(12);
    maxHeap.add(10);

    expect(maxHeap.toString()).toBe('12,3,10');

    maxHeap.add(11);
    expect(maxHeap.toString()).toBe('12,11,10,3');

    expect(maxHeap.poll()).toBe(12);
    expect(maxHeap.toString()).toBe('11,3,10');
  });

  it('should be possible to find item indices in heap', () => {
    const maxHeap = new MaxHeap();

    maxHeap.add(3);
    maxHeap.add(12);
    maxHeap.add(10);
    maxHeap.add(11);
    maxHeap.add(11);

    expect(maxHeap.toString()).toBe('12,11,10,3,11');

    expect(maxHeap.find(5)).toEqual([]);
    expect(maxHeap.find(12)).toEqual([0]);
    expect(maxHeap.find(11)).toEqual([1, 4]);
  });

  it('should be possible to remove items from heap with heapify down', () => {
    const maxHeap = new MaxHeap();

    maxHeap.add(3);
    maxHeap.add(12);
    maxHeap.add(10);
    maxHeap.add(11);
    maxHeap.add(11);

    expect(maxHeap.toString()).toBe('12,11,10,3,11');

    expect(maxHeap.remove(12).toString()).toEqual('11,11,10,3');
    expect(maxHeap.remove(12).peek()).toEqual(11);
    expect(maxHeap.remove(11).toString()).toEqual('10,3');
    expect(maxHeap.remove(10).peek()).toEqual(3);
  });

  it('should be possible to remove items from heap with heapify up', () => {
    const maxHeap = new MaxHeap();

    maxHeap.add(3);
    maxHeap.add(10);
    maxHeap.add(5);
    maxHeap.add(6);
    maxHeap.add(7);
    maxHeap.add(4);
    maxHeap.add(6);
    maxHeap.add(8);
    maxHeap.add(2);
    maxHeap.add(1);

    expect(maxHeap.toString()).toBe('10,8,6,7,6,4,5,3,2,1');
    expect(maxHeap.remove(4).toString()).toEqual('10,8,6,7,6,1,5,3,2');
    expect(maxHeap.remove(3).toString()).toEqual('10,8,6,7,6,1,5,2');
    expect(maxHeap.remove(5).toString()).toEqual('10,8,6,7,6,1,2');
    expect(maxHeap.remove(10).toString()).toEqual('8,7,6,2,6,1');
    expect(maxHeap.remove(6).toString()).toEqual('8,7,1,2');
    expect(maxHeap.remove(2).toString()).toEqual('8,7,1');
    expect(maxHeap.remove(1).toString()).toEqual('8,7');
    expect(maxHeap.remove(7).toString()).toEqual('8');
    expect(maxHeap.remove(8).toString()).toEqual('');
  });

  it('should be possible to remove items from heap with custom finding comparator', () => {
    const maxHeap = new MaxHeap();
    maxHeap.add('a');
    maxHeap.add('bb');
    maxHeap.add('ccc');
    maxHeap.add('dddd');

    expect(maxHeap.toString()).toBe('dddd,ccc,bb,a');

    const comparator = new Comparator((a, b) => {
```

```
      if (a.length === b.length) {
        return 0;
      }

      return a.length < b.length ? -1 : 1;
    });

    maxHeap.remove('hey', comparator);
    expect(maxHeap.toString()).toBe('dddd,a,bb');
  });
});
```

Listing 264: MinHeap.test.js

```
import MinHeap from '../MinHeap';
import Comparator from '../../../utils/comparator/Comparator';

describe('MinHeap', () => {
  it('should create an empty min heap', () => {
    const minHeap = new MinHeap();

    expect(minHeap).toBeDefined();
    expect(minHeap.peek()).toBeNull();
    expect(minHeap.isEmpty()).toBe(true);
  });

  it('should add items to the heap and heapify it up', () => {
    const minHeap = new MinHeap();

    minHeap.add(5);
    expect(minHeap.isEmpty()).toBe(false);
    expect(minHeap.peek()).toBe(5);
    expect(minHeap.toString()).toBe('5');

    minHeap.add(3);
    expect(minHeap.peek()).toBe(3);
    expect(minHeap.toString()).toBe('3,5');

    minHeap.add(10);
    expect(minHeap.peek()).toBe(3);
    expect(minHeap.toString()).toBe('3,5,10');

    minHeap.add(1);
    expect(minHeap.peek()).toBe(1);
    expect(minHeap.toString()).toBe('1,3,10,5');

    minHeap.add(1);
    expect(minHeap.peek()).toBe(1);
    expect(minHeap.toString()).toBe('1,1,10,5,3');

    expect(minHeap.poll()).toBe(1);
    expect(minHeap.toString()).toBe('1,3,10,5');

    expect(minHeap.poll()).toBe(1);
    expect(minHeap.toString()).toBe('3,5,10');

    expect(minHeap.poll()).toBe(3);
    expect(minHeap.toString()).toBe('5,10');
  });

  it('should poll items from the heap and heapify it down', () => {
    const minHeap = new MinHeap();

    minHeap.add(5);
    minHeap.add(3);
    minHeap.add(10);
    minHeap.add(11);
    minHeap.add(1);

    expect(minHeap.toString()).toBe('1,3,10,11,5');

    expect(minHeap.poll()).toBe(1);
    expect(minHeap.toString()).toBe('3,5,10,11');

    expect(minHeap.poll()).toBe(3);
    expect(minHeap.toString()).toBe('5,11,10');

    expect(minHeap.poll()).toBe(5);
    expect(minHeap.toString()).toBe('10,11');

    expect(minHeap.poll()).toBe(10);
    expect(minHeap.toString()).toBe('11');

    expect(minHeap.poll()).toBe(11);
    expect(minHeap.toString()).toBe('');

    expect(minHeap.poll()).toBeNull();
    expect(minHeap.toString()).toBe('');
  });

  it('should heapify down through the right branch as well', () => {
    const minHeap = new MinHeap();
```

```
    minHeap.add(3);
    minHeap.add(12);
    minHeap.add(10);

    expect(minHeap.toString()).toBe('3,12,10');

    minHeap.add(11);
    expect(minHeap.toString()).toBe('3,11,10,12');

    expect(minHeap.poll()).toBe(3);
    expect(minHeap.toString()).toBe('10,11,12');
  });

  it('should be possible to find item indices in heap', () => {
    const minHeap = new MinHeap();

    minHeap.add(3);
    minHeap.add(12);
    minHeap.add(10);
    minHeap.add(11);
    minHeap.add(11);

    expect(minHeap.toString()).toBe('3,11,10,12,11');

    expect(minHeap.find(5)).toEqual([]);
    expect(minHeap.find(3)).toEqual([0]);
    expect(minHeap.find(11)).toEqual([1, 4]);
  });

  it('should be possible to remove items from heap with heapify down', () => {
    const minHeap = new MinHeap();

    minHeap.add(3);
    minHeap.add(12);
    minHeap.add(10);
    minHeap.add(11);
    minHeap.add(11);

    expect(minHeap.toString()).toBe('3,11,10,12,11');

    expect(minHeap.remove(3).toString()).toEqual('10,11,11,12');
    expect(minHeap.remove(3).peek()).toEqual(10);
    expect(minHeap.remove(11).toString()).toEqual('10,12');
    expect(minHeap.remove(3).peek()).toEqual(10);
  });

  it('should be possible to remove items from heap with heapify up', () => {
    const minHeap = new MinHeap();

    minHeap.add(3);
    minHeap.add(10);
    minHeap.add(5);
    minHeap.add(6);
    minHeap.add(7);
    minHeap.add(4);
    minHeap.add(6);
    minHeap.add(8);
    minHeap.add(2);
    minHeap.add(1);

    expect(minHeap.toString()).toBe('1,2,4,6,3,5,6,10,8,7');
    expect(minHeap.remove(8).toString()).toEqual('1,2,4,6,3,5,6,10,7');
    expect(minHeap.remove(7).toString()).toEqual('1,2,4,6,3,5,6,10');
    expect(minHeap.remove(1).toString()).toEqual('2,3,4,6,10,5,6');
    expect(minHeap.remove(2).toString()).toEqual('3,6,4,6,10,5');
    expect(minHeap.remove(6).toString()).toEqual('3,5,4,10');
    expect(minHeap.remove(10).toString()).toEqual('3,5,4');
    expect(minHeap.remove(5).toString()).toEqual('3,4');
    expect(minHeap.remove(3).toString()).toEqual('4');
    expect(minHeap.remove(4).toString()).toEqual('');
  });

  it('should be possible to remove items from heap with custom finding comparator', () => {
    const minHeap = new MinHeap();
    minHeap.add('dddd');
    minHeap.add('ccc');
    minHeap.add('bb');
    minHeap.add('a');

    expect(minHeap.toString()).toBe('a,bb,ccc,dddd');

    const comparator = new Comparator((a, b) => {
```

```
      if (a.length === b.length) {
        return 0;
      }

      return a.length < b.length ? -1 : 1;
    });

    minHeap.remove('hey', comparator);
    expect(minHeap.toString()).toBe('a,bb,dddd');
  });

  it('should remove values from heap and correctly re-order the tree', () => {
    const minHeap = new MinHeap();

    minHeap.add(1);
    minHeap.add(2);
    minHeap.add(3);
    minHeap.add(4);
    minHeap.add(5);
    minHeap.add(6);
    minHeap.add(7);
    minHeap.add(8);
    minHeap.add(9);

    expect(minHeap.toString()).toBe('1,2,3,4,5,6,7,8,9');

    minHeap.remove(2);
    expect(minHeap.toString()).toBe('1,4,3,8,5,6,7,9');

    minHeap.remove(4);
    expect(minHeap.toString()).toBe('1,5,3,8,9,6,7');
  });
});
```

Listing 265: Heap.js

```
import Comparator from '../../utils/comparator/Comparator';

/**
 * Parent class for Min and Max Heaps.
 */
export default class Heap {
  /**
   * @constructs Heap
   * @param {Function} [comparatorFunction]
   */
  constructor(comparatorFunction) {
    if (new.target === Heap) {
      throw new TypeError('Cannot construct Heap instance directly');
    }

    // Array representation of the heap.
    this.heapContainer = [];
    this.compare = new Comparator(comparatorFunction);
  }

  /**
   * @param {number} parentIndex
   * @return {number}
   */
  getLeftChildIndex(parentIndex) {
    return (2 * parentIndex) + 1;
  }

  /**
   * @param {number} parentIndex
   * @return {number}
   */
  getRightChildIndex(parentIndex) {
    return (2 * parentIndex) + 2;
  }

  /**
   * @param {number} childIndex
   * @return {number}
   */
  getParentIndex(childIndex) {
    return Math.floor((childIndex - 1) / 2);
  }

  /**
   * @param {number} childIndex
   * @return {boolean}
   */
  hasParent(childIndex) {
    return this.getParentIndex(childIndex) >= 0;
  }

  /**
   * @param {number} parentIndex
   * @return {boolean}
   */
  hasLeftChild(parentIndex) {
    return this.getLeftChildIndex(parentIndex) < this.heapContainer.length;
  }

  /**
   * @param {number} parentIndex
   * @return {boolean}
   */
  hasRightChild(parentIndex) {
    return this.getRightChildIndex(parentIndex) < this.heapContainer.length;
  }

  /**
   * @param {number} parentIndex
   * @return {*}
   */
  leftChild(parentIndex) {
    return this.heapContainer[this.getLeftChildIndex(parentIndex)];
  }

  /**
   * @param {number} parentIndex
   * @return {*}
```

```
   */
  rightChild(parentIndex) {
    return this.heapContainer[this.getRightChildIndex(parentIndex)];
  }

  /**
   * @param {number} childIndex
   * @return {*}
   */
  parent(childIndex) {
    return this.heapContainer[this.getParentIndex(childIndex)];
  }

  /**
   * @param {number} indexOne
   * @param {number} indexTwo
   */
  swap(indexOne, indexTwo) {
    const tmp = this.heapContainer[indexTwo];
    this.heapContainer[indexTwo] = this.heapContainer[indexOne];
    this.heapContainer[indexOne] = tmp;
  }

  /**
   * @return {*}
   */
  peek() {
    if (this.heapContainer.length === 0) {
      return null;
    }

    return this.heapContainer[0];
  }

  /**
   * @return {*}
   */
  poll() {
    if (this.heapContainer.length === 0) {
      return null;
    }

    if (this.heapContainer.length === 1) {
      return this.heapContainer.pop();
    }

    const item = this.heapContainer[0];

    // Move the last element from the end to the head.
    this.heapContainer[0] = this.heapContainer.pop();
    this.heapifyDown();

    return item;
  }

  /**
   * @param {*} item
   * @return {Heap}
   */
  add(item) {
    this.heapContainer.push(item);
    this.heapifyUp();
    return this;
  }

  /**
   * @param {*} item
   * @param {Comparator} [comparator]
   * @return {Heap}
   */
  remove(item, comparator = this.compare) {
    // Find number of items to remove.
    const numberOfItemsToRemove = this.find(item, comparator).length;

    for (let iteration = 0; iteration < numberOfItemsToRemove; iteration += 1) {
      // We need to find item index to remove each time after removal since
      // indices are being changed after each heapify process.
      const indexToRemove = this.find(item, comparator).pop();

      // If we need to remove last child in the heap then just remove it.
      // There is no need to heapify the heap afterwards.
      if (indexToRemove === (this.heapContainer.length - 1)) {
```

```
        this.heapContainer.pop();
      } else {
        // Move last element in heap to the vacant (removed) position.
        this.heapContainer[indexToRemove] = this.heapContainer.pop();

        // Get parent.
        const parentItem = this.parent(indexToRemove);

        // If there is no parent or parent is in correct order with the node
        // we're going to delete then heapify down. Otherwise heapify up.
        if (
          this.hasLeftChild(indexToRemove)
          && (
            !parentItem
            || this.pairIsInCorrectOrder(parentItem, this.heapContainer[indexToRemove])
          )
        ) {
          this.heapifyDown(indexToRemove);
        } else {
          this.heapifyUp(indexToRemove);
        }
      }
    }

    return this;
  }

  /**
   * @param {*} item
   * @param {Comparator} [comparator]
   * @return {Number[]}
   */
  find(item, comparator = this.compare) {
    const foundItemIndices = [];

    for (let itemIndex = 0; itemIndex < this.heapContainer.length; itemIndex += 1) {
      if (comparator.equal(item, this.heapContainer[itemIndex])) {
        foundItemIndices.push(itemIndex);
      }
    }

    return foundItemIndices;
  }

  /**
   * @return {boolean}
   */
  isEmpty() {
    return !this.heapContainer.length;
  }

  /**
   * @return {string}
   */
  toString() {
    return this.heapContainer.toString();
  }

  /**
   * @param {number} [customStartIndex]
   */
  heapifyUp(customStartIndex) {
    // Take the last element (last in array or the bottom left in a tree)
    // in the heap container and lift it up until it is in the correct
    // order with respect to its parent element.
    let currentIndex = customStartIndex || this.heapContainer.length - 1;

    while (
      this.hasParent(currentIndex)
      && !this.pairIsInCorrectOrder(this.parent(currentIndex), this.heapContainer[currentIndex])
    ) {
      this.swap(currentIndex, this.getParentIndex(currentIndex));
      currentIndex = this.getParentIndex(currentIndex);
    }
  }

  /**
   * @param {number} [customStartIndex]
   */
  heapifyDown(customStartIndex = 0) {
    // Compare the parent element to its children and swap parent with the appropriate
    // child (smallest child for MinHeap, largest child for MaxHeap).
```

```
    // Do the same for next children after swap.
    let currentIndex = customStartIndex;
    let nextIndex = null;

    while (this.hasLeftChild(currentIndex)) {
      if (
        this.hasRightChild(currentIndex)
        && this.pairIsInCorrectOrder(this.rightChild(currentIndex), this.leftChild(currentIndex))
      ) {
        nextIndex = this.getRightChildIndex(currentIndex);
      } else {
        nextIndex = this.getLeftChildIndex(currentIndex);
      }

      if (this.pairIsInCorrectOrder(
        this.heapContainer[currentIndex],
        this.heapContainer[nextIndex],
      )) {
        break;
      }

      this.swap(currentIndex, nextIndex);
      currentIndex = nextIndex;
    }
  }

  /**
   * Checks if pair of heap elements is in correct order.
   * For MinHeap the first element must be always smaller or equal.
   * For MaxHeap the first element must be always bigger or equal.
   *
   * @param {*} firstElement
   * @param {*} secondElement
   * @return {boolean}
   */
  /* istanbul ignore next */
  pairIsInCorrectOrder(firstElement, secondElement) {
    throw new Error(`
      You have to implement heap pair comparision method
      for ${firstElement} and ${secondElement} values.
    `);
  }
}
```

Listing 266: MaxHeap.js

```
import Heap from './Heap';

export default class MaxHeap extends Heap {
  /**
   * Checks if pair of heap elements is in correct order.
   * For MinHeap the first element must be always smaller or equal.
   * For MaxHeap the first element must be always bigger or equal.
   *
   * @param {*} firstElement
   * @param {*} secondElement
   * @return {boolean}
   */
  pairIsInCorrectOrder(firstElement, secondElement) {
    return this.compare.greaterThanOrEqual(firstElement, secondElement);
  }
}
```

Listing 267: MinHeap.js

```
 import Heap from './Heap';

export default class MinHeap extends Heap {
  /**
   * Checks if pair of heap elements is in correct order.
   * For MinHeap the first element must be always smaller or equal.
   * For MaxHeap the first element must be always bigger or equal.
   *
   * @param {*} firstElement
   * @param {*} secondElement
   * @return {boolean}
   */
  pairIsInCorrectOrder(firstElement, secondElement) {
    return this.compare.lessThanOrEqual(firstElement, secondElement);
  }
}
```

Listing 268: LinkedList.test.js

```
import LinkedList from '../LinkedList';

describe('LinkedList', () => {
  it('should create empty linked list', () => {
    const linkedList = new LinkedList();
    expect(linkedList.toString()).toBe('');
  });

  it('should append node to linked list', () => {
    const linkedList = new LinkedList();

    expect(linkedList.head).toBeNull();
    expect(linkedList.tail).toBeNull();

    linkedList.append(1);
    linkedList.append(2);

    expect(linkedList.toString()).toBe('1,2');
    expect(linkedList.tail.next).toBeNull();
  });

  it('should prepend node to linked list', () => {
    const linkedList = new LinkedList();

    linkedList.prepend(2);
    expect(linkedList.head.toString()).toBe('2');
    expect(linkedList.tail.toString()).toBe('2');

    linkedList.append(1);
    linkedList.prepend(3);

    expect(linkedList.toString()).toBe('3,2,1');
  });

  it('should delete node by value from linked list', () => {
    const linkedList = new LinkedList();

    expect(linkedList.delete(5)).toBeNull();

    linkedList.append(1);
    linkedList.append(1);
    linkedList.append(2);
    linkedList.append(3);
    linkedList.append(3);
    linkedList.append(3);
    linkedList.append(4);
    linkedList.append(5);

    expect(linkedList.head.toString()).toBe('1');
    expect(linkedList.tail.toString()).toBe('5');

    const deletedNode = linkedList.delete(3);
    expect(deletedNode.value).toBe(3);
    expect(linkedList.toString()).toBe('1,1,2,4,5');

    linkedList.delete(3);
    expect(linkedList.toString()).toBe('1,1,2,4,5');

    linkedList.delete(1);
    expect(linkedList.toString()).toBe('2,4,5');

    expect(linkedList.head.toString()).toBe('2');
    expect(linkedList.tail.toString()).toBe('5');

    linkedList.delete(5);
    expect(linkedList.toString()).toBe('2,4');

    expect(linkedList.head.toString()).toBe('2');
    expect(linkedList.tail.toString()).toBe('4');

    linkedList.delete(4);
    expect(linkedList.toString()).toBe('2');

    expect(linkedList.head.toString()).toBe('2');
    expect(linkedList.tail.toString()).toBe('2');

    linkedList.delete(2);
    expect(linkedList.toString()).toBe('');
  });
```

```
it('should delete linked list tail', () => {
  const linkedList = new LinkedList();

  linkedList.append(1);
  linkedList.append(2);
  linkedList.append(3);

  expect(linkedList.head.toString()).toBe('1');
  expect(linkedList.tail.toString()).toBe('3');

  const deletedNode1 = linkedList.deleteTail();

  expect(deletedNode1.value).toBe(3);
  expect(linkedList.toString()).toBe('1,2');
  expect(linkedList.head.toString()).toBe('1');
  expect(linkedList.tail.toString()).toBe('2');

  const deletedNode2 = linkedList.deleteTail();

  expect(deletedNode2.value).toBe(2);
  expect(linkedList.toString()).toBe('1');
  expect(linkedList.head.toString()).toBe('1');
  expect(linkedList.tail.toString()).toBe('1');

  const deletedNode3 = linkedList.deleteTail();

  expect(deletedNode3.value).toBe(1);
  expect(linkedList.toString()).toBe('');
  expect(linkedList.head).toBeNull();
  expect(linkedList.tail).toBeNull();
});

it('should delete linked list head', () => {
  const linkedList = new LinkedList();

  expect(linkedList.deleteHead()).toBeNull();

  linkedList.append(1);
  linkedList.append(2);

  expect(linkedList.head.toString()).toBe('1');
  expect(linkedList.tail.toString()).toBe('2');

  const deletedNode1 = linkedList.deleteHead();

  expect(deletedNode1.value).toBe(1);
  expect(linkedList.toString()).toBe('2');
  expect(linkedList.head.toString()).toBe('2');
  expect(linkedList.tail.toString()).toBe('2');

  const deletedNode2 = linkedList.deleteHead();

  expect(deletedNode2.value).toBe(2);
  expect(linkedList.toString()).toBe('');
  expect(linkedList.head).toBeNull();
  expect(linkedList.tail).toBeNull();
});

it('should be possible to store objects in the list and to print them out', () => {
  const linkedList = new LinkedList();

  const nodeValue1 = { value: 1, key: 'key1' };
  const nodeValue2 = { value: 2, key: 'key2' };

  linkedList
    .append(nodeValue1)
    .prepend(nodeValue2);

  const nodeStringifier = value => `${value.key}:${value.value}`;

  expect(linkedList.toString(nodeStringifier)).toBe('key2:2,key1:1');
});

it('should find node by value', () => {
  const linkedList = new LinkedList();

  expect(linkedList.find({ value: 5 })).toBeNull();

  linkedList.append(1);
  expect(linkedList.find({ value: 1 })).toBeDefined();
```

```
  linkedList
    .append(2)
    .append(3);

  const node = linkedList.find({ value: 2 });

  expect(node.value).toBe(2);
  expect(linkedList.find({ value: 5 })).toBeNull();
});

it('should find node by callback', () => {
  const linkedList = new LinkedList();

  linkedList
    .append({ value: 1, key: 'test1' })
    .append({ value: 2, key: 'test2' })
    .append({ value: 3, key: 'test3' });

  const node = linkedList.find({ callback: value => value.key === 'test2' });

  expect(node).toBeDefined();
  expect(node.value.value).toBe(2);
  expect(node.value.key).toBe('test2');
  expect(linkedList.find({ callback: value => value.key === 'test5' })).toBeNull();
});

it('should create linked list from array', () => {
  const linkedList = new LinkedList();
  linkedList.fromArray([1, 1, 2, 3, 3, 3, 4, 5]);

  expect(linkedList.toString()).toBe('1,1,2,3,3,3,4,5');
});

it('should find node by means of custom compare function', () => {
  const comparatorFunction = (a, b) => {
    if (a.customValue === b.customValue) {
      return 0;
    }

    return a.customValue < b.customValue ? -1 : 1;
  };

  const linkedList = new LinkedList(comparatorFunction);

  linkedList
    .append({ value: 1, customValue: 'test1' })
    .append({ value: 2, customValue: 'test2' })
    .append({ value: 3, customValue: 'test3' });

  const node = linkedList.find({
    value: { value: 2, customValue: 'test2' },
  });

  expect(node).toBeDefined();
  expect(node.value.value).toBe(2);
  expect(node.value.customValue).toBe('test2');
  expect(linkedList.find({ value: 2, customValue: 'test5' })).toBeNull();
});

it('should reverse linked list', () => {
  const linkedList = new LinkedList();

  // Add test values to linked list.
  linkedList
    .append(1)
    .append(2)
    .append(3);

  expect(linkedList.toString()).toBe('1,2,3');
  expect(linkedList.head.value).toBe(1);
  expect(linkedList.tail.value).toBe(3);

  // Reverse linked list.
  linkedList.reverse();
  expect(linkedList.toString()).toBe('3,2,1');
  expect(linkedList.head.value).toBe(3);
  expect(linkedList.tail.value).toBe(1);

  // Reverse linked list back to initial state.
  linkedList.reverse();
  expect(linkedList.toString()).toBe('1,2,3');
  expect(linkedList.head.value).toBe(1);
```

```
    expect(linkedList.tail.value).toBe(3);
  });
});
```

Listing 269: LinkedListNode.test.js

```
import LinkedListNode from '../LinkedListNode';

describe('LinkedListNode', () => {
  it('should create list node with value', () => {
    const node = new LinkedListNode(1);

    expect(node.value).toBe(1);
    expect(node.next).toBeNull();
  });

  it('should create list node with object as a value', () => {
    const nodeValue = { value: 1, key: 'test' };
    const node = new LinkedListNode(nodeValue);

    expect(node.value.value).toBe(1);
    expect(node.value.key).toBe('test');
    expect(node.next).toBeNull();
  });

  it('should link nodes together', () => {
    const node2 = new LinkedListNode(2);
    const node1 = new LinkedListNode(1, node2);

    expect(node1.next).toBeDefined();
    expect(node2.next).toBeNull();
    expect(node1.value).toBe(1);
    expect(node1.next.value).toBe(2);
  });

  it('should convert node to string', () => {
    const node = new LinkedListNode(1);

    expect(node.toString()).toBe('1');

    node.value = 'string value';
    expect(node.toString()).toBe('string value');
  });

  it('should convert node to string with custom stringifier', () => {
    const nodeValue = { value: 1, key: 'test' };
    const node = new LinkedListNode(nodeValue);
    const toStringCallback = value => `value: ${value.value}, key: ${value.key}`;

    expect(node.toString(toStringCallback)).toBe('value: 1, key: test');
  });
});
```

Listing 270: LinkedList.js

```
import LinkedListNode from './LinkedListNode';
import Comparator from '../../utils/comparator/Comparator';

export default class LinkedList {
  /**
   * @param {Function} [comparatorFunction]
   */
  constructor(comparatorFunction) {
    /** @var LinkedListNode */
    this.head = null;

    /** @var LinkedListNode */
    this.tail = null;

    this.compare = new Comparator(comparatorFunction);
  }

  /**
   * @param {*} value
   * @return {LinkedList}
   */
  prepend(value) {
    // Make new node to be a head.
    const newNode = new LinkedListNode(value, this.head);
    this.head = newNode;

    // If there is no tail yet let's make new node a tail.
    if (!this.tail) {
      this.tail = newNode;
    }

    return this;
  }

  /**
   * @param {*} value
   * @return {LinkedList}
   */
  append(value) {
    const newNode = new LinkedListNode(value);

    // If there is no head yet let's make new node a head.
    if (!this.head) {
      this.head = newNode;
      this.tail = newNode;

      return this;
    }

    // Attach new node to the end of linked list.
    this.tail.next = newNode;
    this.tail = newNode;

    return this;
  }

  /**
   * @param {*} value
   * @return {LinkedListNode}
   */
  delete(value) {
    if (!this.head) {
      return null;
    }

    let deletedNode = null;

    // If the head must be deleted then make next node that is differ
    // from the head to be a new head.
    while (this.head && this.compare.equal(this.head.value, value)) {
      deletedNode = this.head;
      this.head = this.head.next;
    }

    let currentNode = this.head;

    if (currentNode !== null) {
      // If next node must be deleted then make next node to be a next next one.
      while (currentNode.next) {
```

```
      if (this.compare.equal(currentNode.next.value, value)) {
        deletedNode = currentNode.next;
        currentNode.next = currentNode.next.next;
      } else {
        currentNode = currentNode.next;
      }
    }
  }

  // Check if tail must be deleted.
  if (this.compare.equal(this.tail.value, value)) {
    this.tail = currentNode;
  }

  return deletedNode;
}

/**
 * @param {Object} findParams
 * @param {*} findParams.value
 * @param {function} [findParams.callback]
 * @return {LinkedListNode}
 */
find({ value = undefined, callback = undefined }) {
  if (!this.head) {
    return null;
  }

  let currentNode = this.head;

  while (currentNode) {
    // If callback is specified then try to find node by callback.
    if (callback && callback(currentNode.value)) {
      return currentNode;
    }

    // If value is specified then try to compare by value..
    if (value !== undefined && this.compare.equal(currentNode.value, value)) {
      return currentNode;
    }

    currentNode = currentNode.next;
  }

  return null;
}

/**
 * @return {LinkedListNode}
 */
deleteTail() {
  const deletedTail = this.tail;

  if (this.head === this.tail) {
    // There is only one node in linked list.
    this.head = null;
    this.tail = null;

    return deletedTail;
  }

  // If there are many nodes in linked list...

  // Rewind to the last node and delete "next" link for the node before the last one.
  let currentNode = this.head;
  while (currentNode.next) {
    if (!currentNode.next.next) {
      currentNode.next = null;
    } else {
      currentNode = currentNode.next;
    }
  }

  this.tail = currentNode;

  return deletedTail;
}

/**
 * @return {LinkedListNode}
 */
deleteHead() {
```

```
    if (!this.head) {
      return null;
    }

    const deletedHead = this.head;

    if (this.head.next) {
      this.head = this.head.next;
    } else {
      this.head = null;
      this.tail = null;
    }

    return deletedHead;
  }

  /**
   * @param {*[]} values - Array of values that need to be converted to linked list.
   * @return {LinkedList}
   */
  fromArray(values) {
    values.forEach(value => this.append(value));

    return this;
  }

  /**
   * @return {LinkedListNode[]}
   */
  toArray() {
    const nodes = [];

    let currentNode = this.head;
    while (currentNode) {
      nodes.push(currentNode);
      currentNode = currentNode.next;
    }

    return nodes;
  }

  /**
   * @param {function} [callback]
   * @return {string}
   */
  toString(callback) {
    return this.toArray().map(node => node.toString(callback)).toString();
  }

  /**
   * Reverse a linked list.
   * @returns {LinkedList}
   */
  reverse() {
    let currNode = this.head;
    let prevNode = null;
    let nextNode = null;

    while (currNode) {
      // Store next node.
      nextNode = currNode.next;

      // Change next node of the current node so it would link to previous node.
      currNode.next = prevNode;

      // Move prevNode and currNode nodes one step forward.
      prevNode = currNode;
      currNode = nextNode;
    }

    // Reset head and tail.
    this.tail = this.head;
    this.head = prevNode;

    return this;
  }
}
```

Listing 271: LinkedListNode.js

```
export default class LinkedListNode {
  constructor(value, next = null) {
    this.value = value;
    this.next = next;
  }

  toString(callback) {
    return callback ? callback(this.value) : `${this.value}`;
  }
}
```

Listing 272: PriorityQueue.test.js

```
import PriorityQueue from '../PriorityQueue';

describe('PriorityQueue', () => {
  it('should create default priority queue', () => {
    const priorityQueue = new PriorityQueue();

    expect(priorityQueue).toBeDefined();
  });

  it('should insert items to the queue and respect priorities', () => {
    const priorityQueue = new PriorityQueue();

    priorityQueue.add(10, 1);
    expect(priorityQueue.peek()).toBe(10);

    priorityQueue.add(5, 2);
    expect(priorityQueue.peek()).toBe(10);

    priorityQueue.add(100, 0);
    expect(priorityQueue.peek()).toBe(100);
  });

  it('should be possible to use objects in priority queue', () => {
    const priorityQueue = new PriorityQueue();

    const user1 = { name: 'Mike' };
    const user2 = { name: 'Bill' };
    const user3 = { name: 'Jane' };

    priorityQueue.add(user1, 1);
    expect(priorityQueue.peek()).toBe(user1);

    priorityQueue.add(user2, 2);
    expect(priorityQueue.peek()).toBe(user1);

    priorityQueue.add(user3, 0);
    expect(priorityQueue.peek()).toBe(user3);
  });

  it('should poll from queue with respect to priorities', () => {
    const priorityQueue = new PriorityQueue();

    priorityQueue.add(10, 1);
    priorityQueue.add(5, 2);
    priorityQueue.add(100, 0);
    priorityQueue.add(200, 0);

    expect(priorityQueue.poll()).toBe(100);
    expect(priorityQueue.poll()).toBe(200);
    expect(priorityQueue.poll()).toBe(10);
    expect(priorityQueue.poll()).toBe(5);
  });

  it('should be possible to change priority of head node', () => {
    const priorityQueue = new PriorityQueue();

    priorityQueue.add(10, 1);
    priorityQueue.add(5, 2);
    priorityQueue.add(100, 0);
    priorityQueue.add(200, 0);

    expect(priorityQueue.peek()).toBe(100);

    priorityQueue.changePriority(100, 10);
    priorityQueue.changePriority(10, 20);

    expect(priorityQueue.poll()).toBe(200);
    expect(priorityQueue.poll()).toBe(5);
    expect(priorityQueue.poll()).toBe(100);
    expect(priorityQueue.poll()).toBe(10);
  });

  it('should be possible to change priority of internal nodes', () => {
    const priorityQueue = new PriorityQueue();

    priorityQueue.add(10, 1);
    priorityQueue.add(5, 2);
    priorityQueue.add(100, 0);
    priorityQueue.add(200, 0);
```

```
    expect(priorityQueue.peek()).toBe(100);

    priorityQueue.changePriority(200, 10);
    priorityQueue.changePriority(10, 20);

    expect(priorityQueue.poll()).toBe(100);
    expect(priorityQueue.poll()).toBe(5);
    expect(priorityQueue.poll()).toBe(200);
    expect(priorityQueue.poll()).toBe(10);
  });

  it('should be possible to change priority along with node addition', () => {
    const priorityQueue = new PriorityQueue();

    priorityQueue.add(10, 1);
    priorityQueue.add(5, 2);
    priorityQueue.add(100, 0);
    priorityQueue.add(200, 0);

    priorityQueue.changePriority(200, 10);
    priorityQueue.changePriority(10, 20);

    priorityQueue.add(15, 15);

    expect(priorityQueue.poll()).toBe(100);
    expect(priorityQueue.poll()).toBe(5);
    expect(priorityQueue.poll()).toBe(200);
    expect(priorityQueue.poll()).toBe(15);
    expect(priorityQueue.poll()).toBe(10);
  });

  it('should be possible to search in priority queue by value', () => {
    const priorityQueue = new PriorityQueue();

    priorityQueue.add(10, 1);
    priorityQueue.add(5, 2);
    priorityQueue.add(100, 0);
    priorityQueue.add(200, 0);
    priorityQueue.add(15, 15);

    expect(priorityQueue.hasValue(70)).toBe(false);
    expect(priorityQueue.hasValue(15)).toBe(true);
  });
});
```

Listing 273: PriorityQueue.js

```
import MinHeap from '../heap/MinHeap';
import Comparator from '../../utils/comparator/Comparator';

// It is the same as min heap except that when comparing two elements
// we take into account its priority instead of the element's value.
export default class PriorityQueue extends MinHeap {
  constructor() {
    // Call MinHip constructor first.
    super();

    // Setup priorities map.
    this.priorities = new Map();

    // Use custom comparator for heap elements that will take element priority
    // instead of element value into account.
    this.compare = new Comparator(this.comparePriority.bind(this));
  }

  /**
   * Add item to the priority queue.
   * @param {*} item - item we're going to add to the queue.
   * @param {number} [priority] - items priority.
   * @return {PriorityQueue}
   */
  add(item, priority = 0) {
    this.priorities.set(item, priority);
    super.add(item);
    return this;
  }

  /**
   * Remove item from priority queue.
   * @param {*} item - item we're going to remove.
   * @param {Comparator} [customFindingComparator] - custom function for finding the item to remove
   * @return {PriorityQueue}
   */
  remove(item, customFindingComparator) {
    super.remove(item, customFindingComparator);
    this.priorities.delete(item);
    return this;
  }

  /**
   * Change priority of the item in a queue.
   * @param {*} item - item we're going to re-prioritize.
   * @param {number} priority - new item's priority.
   * @return {PriorityQueue}
   */
  changePriority(item, priority) {
    this.remove(item, new Comparator(this.compareValue));
    this.add(item, priority);
    return this;
  }

  /**
   * Find item by ite value.
   * @param {*} item
   * @return {Number[]}
   */
  findByValue(item) {
    return this.find(item, new Comparator(this.compareValue));
  }

  /**
   * Check if item already exists in a queue.
   * @param {*} item
   * @return {boolean}
   */
  hasValue(item) {
    return this.findByValue(item).length > 0;
  }

  /**
   * Compares priorities of two items.
   * @param {*} a
   * @param {*} b
   * @return {number}
   */
  comparePriority(a, b) {
```

```
    if (this.priorities.get(a) === this.priorities.get(b)) {
      return 0;
    }
    return this.priorities.get(a) < this.priorities.get(b) ? -1 : 1;
  }

  /**
   * Compares values of two items.
   * @param {*} a
   * @param {*} b
   * @return {number}
   */
  compareValue(a, b) {
    if (a === b) {
      return 0;
    }
    return a < b ? -1 : 1;
  }
}
```

Listing 274: Queue.test.js

```
import Queue from '../Queue';

describe('Queue', () => {
  it('should create empty queue', () => {
    const queue = new Queue();
    expect(queue).not.toBeNull();
    expect(queue.linkedList).not.toBeNull();
  });

  it('should enqueue data to queue', () => {
    const queue = new Queue();

    queue.enqueue(1);
    queue.enqueue(2);

    expect(queue.toString()).toBe('1,2');
  });

  it('should be possible to enqueue/dequeue objects', () => {
    const queue = new Queue();

    queue.enqueue({ value: 'test1', key: 'key1' });
    queue.enqueue({ value: 'test2', key: 'key2' });

    const stringifier = value => `${value.key}:${value.value}`;

    expect(queue.toString(stringifier)).toBe('key1:test1,key2:test2');
    expect(queue.dequeue().value).toBe('test1');
    expect(queue.dequeue().value).toBe('test2');
  });

  it('should peek data from queue', () => {
    const queue = new Queue();

    expect(queue.peek()).toBeNull();

    queue.enqueue(1);
    queue.enqueue(2);

    expect(queue.peek()).toBe(1);
    expect(queue.peek()).toBe(1);
  });

  it('should check if queue is empty', () => {
    const queue = new Queue();

    expect(queue.isEmpty()).toBe(true);

    queue.enqueue(1);

    expect(queue.isEmpty()).toBe(false);
  });

  it('should dequeue from queue in FIFO order', () => {
    const queue = new Queue();

    queue.enqueue(1);
    queue.enqueue(2);

    expect(queue.dequeue()).toBe(1);
    expect(queue.dequeue()).toBe(2);
    expect(queue.dequeue()).toBeNull();
    expect(queue.isEmpty()).toBe(true);
  });
});
```

Listing 275: Queue.js

```
import LinkedList from '../linked-list/LinkedList';

export default class Queue {
  constructor() {
    // We're going to implement Queue based on LinkedList since the two
    // structures are quite similar. Namely, they both operate mostly on
    // the elements at the beginning and the end. Compare enqueue/dequeue
    // operations of Queue with append/deleteHead operations of LinkedList.
    this.linkedList = new LinkedList();
  }

  /**
   * @return {boolean}
   */
  isEmpty() {
    return !this.linkedList.head;
  }

  /**
   * Read the element at the front of the queue without removing it.
   * @return {*}
   */
  peek() {
    if (!this.linkedList.head) {
      return null;
    }

    return this.linkedList.head.value;
  }

  /**
   * Add a new element to the end of the queue (the tail of the linked list).
   * This element will be processed after all elements ahead of it.
   * @param {*} value
   */
  enqueue(value) {
    this.linkedList.append(value);
  }

  /**
   * Remove the element at the front of the queue (the head of the linked list).
   * If the queue is empty, return null.
   * @return {*}
   */
  dequeue() {
    const removedHead = this.linkedList.deleteHead();
    return removedHead ? removedHead.value : null;
  }

  /**
   * @param [callback]
   * @return {string}
   */
  toString(callback) {
    // Return string representation of the queue's linked list.
    return this.linkedList.toString(callback);
  }
}
```

Listing 276: Stack.test.js

```
import Stack from '../Stack';

describe('Stack', () => {
  it('should create empty stack', () => {
    const stack = new Stack();
    expect(stack).not.toBeNull();
    expect(stack.linkedList).not.toBeNull();
  });

  it('should stack data to stack', () => {
    const stack = new Stack();

    stack.push(1);
    stack.push(2);

    expect(stack.toString()).toBe('2,1');
  });

  it('should peek data from stack', () => {
    const stack = new Stack();

    expect(stack.peek()).toBeNull();

    stack.push(1);
    stack.push(2);

    expect(stack.peek()).toBe(2);
    expect(stack.peek()).toBe(2);
  });

  it('should check if stack is empty', () => {
    const stack = new Stack();

    expect(stack.isEmpty()).toBe(true);

    stack.push(1);

    expect(stack.isEmpty()).toBe(false);
  });

  it('should pop data from stack', () => {
    const stack = new Stack();

    stack.push(1);
    stack.push(2);

    expect(stack.pop()).toBe(2);
    expect(stack.pop()).toBe(1);
    expect(stack.pop()).toBeNull();
    expect(stack.isEmpty()).toBe(true);
  });

  it('should be possible to push/pop objects', () => {
    const stack = new Stack();

    stack.push({ value: 'test1', key: 'key1' });
    stack.push({ value: 'test2', key: 'key2' });

    const stringifier = value => `${value.key}:${value.value}`;

    expect(stack.toString(stringifier)).toBe('key2:test2,key1:test1');
    expect(stack.pop().value).toBe('test2');
    expect(stack.pop().value).toBe('test1');
  });

  it('should be possible to convert stack to array', () => {
    const stack = new Stack();

    expect(stack.peek()).toBeNull();

    stack.push(1);
    stack.push(2);
    stack.push(3);

    expect(stack.toArray()).toEqual([3, 2, 1]);
  });
});
```

Listing 277: Stack.js

```
import LinkedList from '../linked-list/LinkedList';

export default class Stack {
  constructor() {
    // We're going to implement Stack based on LinkedList since these
    // structures are quite similar. Compare push/pop operations of the Stack
    // with prepend/deleteHead operations of LinkedList.
    this.linkedList = new LinkedList();
  }

  /**
   * @return {boolean}
   */
  isEmpty() {
    // The stack is empty if its linked list doesn't have a head.
    return !this.linkedList.head;
  }

  /**
   * @return {*}
   */
  peek() {
    if (this.isEmpty()) {
      // If the linked list is empty then there is nothing to peek from.
      return null;
    }

    // Just read the value from the start of linked list without deleting it.
    return this.linkedList.head.value;
  }

  /**
   * @param {*} value
   */
  push(value) {
    // Pushing means to lay the value on top of the stack. Therefore let's just add
    // the new value at the start of the linked list.
    this.linkedList.prepend(value);
  }

  /**
   * @return {*}
   */
  pop() {
    // Let's try to delete the first node (the head) from the linked list.
    // If there is no head (the linked list is empty) just return null.
    const removedHead = this.linkedList.deleteHead();
    return removedHead ? removedHead.value : null;
  }

  /**
   * @return {*[]}
   */
  toArray() {
    return this.linkedList
      .toArray()
      .map(linkedListNode => linkedListNode.value);
  }

  /**
   * @param {function} [callback]
   * @return {string}
   */
  toString(callback) {
    return this.linkedList.toString(callback);
  }
}
```

Listing 278: BinaryTreeNode.test.js

```
import BinaryTreeNode from '../BinaryTreeNode';

describe('BinaryTreeNode', () => {
  it('should create node', () => {
    const node = new BinaryTreeNode();

    expect(node).toBeDefined();

    expect(node.value).toBeNull();
    expect(node.left).toBeNull();
    expect(node.right).toBeNull();

    const leftNode = new BinaryTreeNode(1);
    const rightNode = new BinaryTreeNode(3);
    const rootNode = new BinaryTreeNode(2);

    rootNode
      .setLeft(leftNode)
      .setRight(rightNode);

    expect(rootNode.value).toBe(2);
    expect(rootNode.left.value).toBe(1);
    expect(rootNode.right.value).toBe(3);
  });

  it('should set parent', () => {
    const leftNode = new BinaryTreeNode(1);
    const rightNode = new BinaryTreeNode(3);
    const rootNode = new BinaryTreeNode(2);

    rootNode
      .setLeft(leftNode)
      .setRight(rightNode);

    expect(rootNode.parent).toBeNull();
    expect(rootNode.left.parent.value).toBe(2);
    expect(rootNode.right.parent.value).toBe(2);
    expect(rootNode.right.parent).toEqual(rootNode);
  });

  it('should traverse node', () => {
    const leftNode = new BinaryTreeNode(1);
    const rightNode = new BinaryTreeNode(3);
    const rootNode = new BinaryTreeNode(2);

    rootNode
      .setLeft(leftNode)
      .setRight(rightNode);

    expect(rootNode.traverseInOrder()).toEqual([1, 2, 3]);

    expect(rootNode.toString()).toBe('1,2,3');
  });

  it('should remove child node', () => {
    const leftNode = new BinaryTreeNode(1);
    const rightNode = new BinaryTreeNode(3);
    const rootNode = new BinaryTreeNode(2);

    rootNode
      .setLeft(leftNode)
      .setRight(rightNode);

    expect(rootNode.traverseInOrder()).toEqual([1, 2, 3]);

    expect(rootNode.removeChild(rootNode.left)).toBe(true);
    expect(rootNode.traverseInOrder()).toEqual([2, 3]);

    expect(rootNode.removeChild(rootNode.right)).toBe(true);
    expect(rootNode.traverseInOrder()).toEqual([2]);

    expect(rootNode.removeChild(rootNode.right)).toBe(false);
    expect(rootNode.traverseInOrder()).toEqual([2]);
  });

  it('should replace child node', () => {
    const leftNode = new BinaryTreeNode(1);
    const rightNode = new BinaryTreeNode(3);
    const rootNode = new BinaryTreeNode(2);
```

```
    rootNode
      .setLeft(leftNode)
      .setRight(rightNode);

    expect(rootNode.traverseInOrder()).toEqual([1, 2, 3]);

    const replacementNode = new BinaryTreeNode(5);
    rightNode.setRight(replacementNode);

    expect(rootNode.traverseInOrder()).toEqual([1, 2, 3, 5]);

    expect(rootNode.replaceChild(rootNode.right, rootNode.right.right)).toBe(true);
    expect(rootNode.right.value).toBe(5);
    expect(rootNode.right.right).toBeNull();
    expect(rootNode.traverseInOrder()).toEqual([1, 2, 5]);

    expect(rootNode.replaceChild(rootNode.right, rootNode.right.right)).toBe(false);
    expect(rootNode.traverseInOrder()).toEqual([1, 2, 5]);

    expect(rootNode.replaceChild(rootNode.right, replacementNode)).toBe(true);
    expect(rootNode.traverseInOrder()).toEqual([1, 2, 5]);

    expect(rootNode.replaceChild(rootNode.left, replacementNode)).toBe(true);
    expect(rootNode.traverseInOrder()).toEqual([5, 2, 5]);

    expect(rootNode.replaceChild(new BinaryTreeNode(), new BinaryTreeNode())).toBe(false);
  });

  it('should calculate node height', () => {
    const root = new BinaryTreeNode(1);
    const left = new BinaryTreeNode(3);
    const right = new BinaryTreeNode(2);
    const grandLeft = new BinaryTreeNode(5);
    const grandRight = new BinaryTreeNode(6);
    const grandGrandLeft = new BinaryTreeNode(7);

    expect(root.height).toBe(0);
    expect(root.balanceFactor).toBe(0);

    root
      .setLeft(left)
      .setRight(right);

    expect(root.height).toBe(1);
    expect(left.height).toBe(0);
    expect(root.balanceFactor).toBe(0);

    left
      .setLeft(grandLeft)
      .setRight(grandRight);

    expect(root.height).toBe(2);
    expect(left.height).toBe(1);
    expect(grandLeft.height).toBe(0);
    expect(grandRight.height).toBe(0);
    expect(root.balanceFactor).toBe(1);

    grandLeft.setLeft(grandGrandLeft);

    expect(root.height).toBe(3);
    expect(left.height).toBe(2);
    expect(grandLeft.height).toBe(1);
    expect(grandRight.height).toBe(0);
    expect(grandGrandLeft.height).toBe(0);
    expect(root.balanceFactor).toBe(2);
  });

  it('should calculate node height for right nodes as well', () => {
    const root = new BinaryTreeNode(1);
    const right = new BinaryTreeNode(2);

    root.setRight(right);

    expect(root.height).toBe(1);
    expect(right.height).toBe(0);
    expect(root.balanceFactor).toBe(-1);
  });

  it('should set null for left and right node', () => {
    const root = new BinaryTreeNode(2);
    const left = new BinaryTreeNode(1);
```

```
    const right = new BinaryTreeNode(3);

    root.setLeft(left);
    root.setRight(right);

    expect(root.left.value).toBe(1);
    expect(root.right.value).toBe(3);

    root.setLeft(null);
    root.setRight(null);

    expect(root.left).toBeNull();
    expect(root.right).toBeNull();
  });

  it('should be possible to create node with object as a value', () => {
    const obj1 = { key: 'object_1', toString: () => 'object_1' };
    const obj2 = { key: 'object_2' };

    const node1 = new BinaryTreeNode(obj1);
    const node2 = new BinaryTreeNode(obj2);

    node1.setLeft(node2);

    expect(node1.value).toEqual(obj1);
    expect(node2.value).toEqual(obj2);
    expect(node1.left.value).toEqual(obj2);

    node1.removeChild(node2);

    expect(node1.value).toEqual(obj1);
    expect(node2.value).toEqual(obj2);
    expect(node1.left).toBeNull();

    expect(node1.toString()).toBe('object_1');
    expect(node2.toString()).toBe('[object Object]');
  });

  it('should be possible to attach meta information to the node', () => {
    const redNode = new BinaryTreeNode(1);
    const blackNode = new BinaryTreeNode(2);

    redNode.meta.set('color', 'red');
    blackNode.meta.set('color', 'black');

    expect(redNode.meta.get('color')).toBe('red');
    expect(blackNode.meta.get('color')).toBe('black');
  });

  it('should detect right uncle', () => {
    const grandParent = new BinaryTreeNode('grand-parent');
    const parent = new BinaryTreeNode('parent');
    const uncle = new BinaryTreeNode('uncle');
    const child = new BinaryTreeNode('child');

    expect(grandParent.uncle).not.toBeDefined();
    expect(parent.uncle).not.toBeDefined();

    grandParent.setLeft(parent);

    expect(parent.uncle).not.toBeDefined();
    expect(child.uncle).not.toBeDefined();

    parent.setLeft(child);

    expect(child.uncle).not.toBeDefined();

    grandParent.setRight(uncle);

    expect(parent.uncle).not.toBeDefined();
    expect(child.uncle).toBeDefined();
    expect(child.uncle).toEqual(uncle);
  });

  it('should detect left uncle', () => {
    const grandParent = new BinaryTreeNode('grand-parent');
    const parent = new BinaryTreeNode('parent');
    const uncle = new BinaryTreeNode('uncle');
    const child = new BinaryTreeNode('child');

    expect(grandParent.uncle).not.toBeDefined();
    expect(parent.uncle).not.toBeDefined();
```

```
    grandParent.setRight(parent);

    expect(parent.uncle).not.toBeDefined();
    expect(child.uncle).not.toBeDefined();

    parent.setRight(child);

    expect(child.uncle).not.toBeDefined();

    grandParent.setLeft(uncle);

    expect(parent.uncle).not.toBeDefined();
    expect(child.uncle).toBeDefined();
    expect(child.uncle).toEqual(uncle);
  });

  it('should be possible to set node values', () => {
    const node = new BinaryTreeNode('initial_value');

    expect(node.value).toBe('initial_value');

    node.setValue('new_value');

    expect(node.value).toBe('new_value');
  });

  it('should be possible to copy node', () => {
    const root = new BinaryTreeNode('root');
    const left = new BinaryTreeNode('left');
    const right = new BinaryTreeNode('right');

    root
      .setLeft(left)
      .setRight(right);

    expect(root.toString()).toBe('left,root,right');

    const newRoot = new BinaryTreeNode('new_root');
    const newLeft = new BinaryTreeNode('new_left');
    const newRight = new BinaryTreeNode('new_right');

    newRoot
      .setLeft(newLeft)
      .setRight(newRight);

    expect(newRoot.toString()).toBe('new_left,new_root,new_right');

    BinaryTreeNode.copyNode(root, newRoot);

    expect(root.toString()).toBe('left,root,right');
    expect(newRoot.toString()).toBe('left,root,right');
  });
});
```

Listing 279: AvlTRee.test.js

```
import AvlTree from '../AvlTree';

describe('AvlTree', () => {
  it('should do simple left-left rotation', () => {
    const tree = new AvlTree();

    tree.insert(4);
    tree.insert(3);
    tree.insert(2);

    expect(tree.toString()).toBe('2,3,4');
    expect(tree.root.value).toBe(3);
    expect(tree.root.height).toBe(1);

    tree.insert(1);

    expect(tree.toString()).toBe('1,2,3,4');
    expect(tree.root.value).toBe(3);
    expect(tree.root.height).toBe(2);

    tree.insert(0);

    expect(tree.toString()).toBe('0,1,2,3,4');
    expect(tree.root.value).toBe(3);
    expect(tree.root.left.value).toBe(1);
    expect(tree.root.height).toBe(2);
  });

  it('should do complex left-left rotation', () => {
    const tree = new AvlTree();

    tree.insert(30);
    tree.insert(20);
    tree.insert(40);
    tree.insert(10);

    expect(tree.root.value).toBe(30);
    expect(tree.root.height).toBe(2);
    expect(tree.toString()).toBe('10,20,30,40');

    tree.insert(25);
    expect(tree.root.value).toBe(30);
    expect(tree.root.height).toBe(2);
    expect(tree.toString()).toBe('10,20,25,30,40');

    tree.insert(5);
    expect(tree.root.value).toBe(20);
    expect(tree.root.height).toBe(2);
    expect(tree.toString()).toBe('5,10,20,25,30,40');
  });

  it('should do simple right-right rotation', () => {
    const tree = new AvlTree();

    tree.insert(2);
    tree.insert(3);
    tree.insert(4);

    expect(tree.toString()).toBe('2,3,4');
    expect(tree.root.value).toBe(3);
    expect(tree.root.height).toBe(1);

    tree.insert(5);

    expect(tree.toString()).toBe('2,3,4,5');
    expect(tree.root.value).toBe(3);
    expect(tree.root.height).toBe(2);

    tree.insert(6);

    expect(tree.toString()).toBe('2,3,4,5,6');
    expect(tree.root.value).toBe(3);
    expect(tree.root.right.value).toBe(5);
    expect(tree.root.height).toBe(2);
  });

  it('should do complex right-right rotation', () => {
    const tree = new AvlTree();
```

```
    tree.insert(30);
    tree.insert(20);
    tree.insert(40);
    tree.insert(50);

    expect(tree.root.value).toBe(30);
    expect(tree.root.height).toBe(2);
    expect(tree.toString()).toBe('20,30,40,50');

    tree.insert(35);
    expect(tree.root.value).toBe(30);
    expect(tree.root.height).toBe(2);
    expect(tree.toString()).toBe('20,30,35,40,50');

    tree.insert(55);
    expect(tree.root.value).toBe(40);
    expect(tree.root.height).toBe(2);
    expect(tree.toString()).toBe('20,30,35,40,50,55');
  });

  it('should do left-right rotation', () => {
    const tree = new AvlTree();

    tree.insert(30);
    tree.insert(20);
    tree.insert(25);

    expect(tree.root.height).toBe(1);
    expect(tree.root.value).toBe(25);
    expect(tree.toString()).toBe('20,25,30');
  });

  it('should do right-left rotation', () => {
    const tree = new AvlTree();

    tree.insert(30);
    tree.insert(40);
    tree.insert(35);

    expect(tree.root.height).toBe(1);
    expect(tree.root.value).toBe(35);
    expect(tree.toString()).toBe('30,35,40');
  });

  it('should create balanced tree: case #1', () => {
    // @see: https://www.youtube.com/watch?v=rbg7Qf8GkQ4&t=839s
    const tree = new AvlTree();

    tree.insert(1);
    tree.insert(2);
    tree.insert(3);

    expect(tree.root.value).toBe(2);
    expect(tree.root.height).toBe(1);
    expect(tree.toString()).toBe('1,2,3');

    tree.insert(6);

    expect(tree.root.value).toBe(2);
    expect(tree.root.height).toBe(2);
    expect(tree.toString()).toBe('1,2,3,6');

    tree.insert(15);

    expect(tree.root.value).toBe(2);
    expect(tree.root.height).toBe(2);
    expect(tree.toString()).toBe('1,2,3,6,15');

    tree.insert(-2);

    expect(tree.root.value).toBe(2);
    expect(tree.root.height).toBe(2);
    expect(tree.toString()).toBe('-2,1,2,3,6,15');

    tree.insert(-5);

    expect(tree.root.value).toBe(2);
    expect(tree.root.height).toBe(2);
    expect(tree.toString()).toBe('-5,-2,1,2,3,6,15');

    tree.insert(-8);
```

```
    expect(tree.root.value).toBe(2);
    expect(tree.root.height).toBe(3);
    expect(tree.toString()).toBe('-8,-5,-2,1,2,3,6,15');
  });

  it('should create balanced tree: case #2', () => {
    // @see https://www.youtube.com/watch?v=7m94k2Qhg68
    const tree = new AvlTree();

    tree.insert(43);
    tree.insert(18);
    tree.insert(22);
    tree.insert(9);
    tree.insert(21);
    tree.insert(6);

    expect(tree.root.value).toBe(18);
    expect(tree.root.height).toBe(2);
    expect(tree.toString()).toBe('6,9,18,21,22,43');

    tree.insert(8);

    expect(tree.root.value).toBe(18);
    expect(tree.root.height).toBe(2);
    expect(tree.toString()).toBe('6,8,9,18,21,22,43');
  });

  it('should do left right rotation and keeping left right node safe', () => {
    const tree = new AvlTree();

    tree.insert(30);
    tree.insert(15);
    tree.insert(40);
    tree.insert(10);
    tree.insert(18);
    tree.insert(35);
    tree.insert(45);
    tree.insert(5);
    tree.insert(12);

    expect(tree.toString()).toBe('5,10,12,15,18,30,35,40,45');
    expect(tree.root.height).toBe(3);

    tree.insert(11);

    expect(tree.toString()).toBe('5,10,11,12,15,18,30,35,40,45');
    expect(tree.root.height).toBe(3);
  });

  it('should do left right rotation and keeping left right node safe', () => {
    const tree = new AvlTree();

    tree.insert(30);
    tree.insert(15);
    tree.insert(40);
    tree.insert(10);
    tree.insert(18);
    tree.insert(35);
    tree.insert(45);
    tree.insert(42);
    tree.insert(47);

    expect(tree.toString()).toBe('10,15,18,30,35,40,42,45,47');
    expect(tree.root.height).toBe(3);

    tree.insert(43);

    expect(tree.toString()).toBe('10,15,18,30,35,40,42,43,45,47');
    expect(tree.root.height).toBe(3);
  });

  it('should remove values from the tree with right-right rotation', () => {
    const tree = new AvlTree();

    tree.insert(10);
    tree.insert(20);
    tree.insert(30);
    tree.insert(40);

    expect(tree.toString()).toBe('10,20,30,40');

    tree.remove(10);
```

```
    expect(tree.toString()).toBe('20,30,40');
    expect(tree.root.value).toBe(30);
    expect(tree.root.left.value).toBe(20);
    expect(tree.root.right.value).toBe(40);
    expect(tree.root.balanceFactor).toBe(0);
  });

  it('should remove values from the tree with left-left rotation', () => {
    const tree = new AvlTree();

    tree.insert(10);
    tree.insert(20);
    tree.insert(30);
    tree.insert(5);

    expect(tree.toString()).toBe('5,10,20,30');

    tree.remove(30);

    expect(tree.toString()).toBe('5,10,20');
    expect(tree.root.value).toBe(10);
    expect(tree.root.left.value).toBe(5);
    expect(tree.root.right.value).toBe(20);
    expect(tree.root.balanceFactor).toBe(0);
  });

  it('should keep balance after removal', () => {
    const tree = new AvlTree();

    tree.insert(1);
    tree.insert(2);
    tree.insert(3);
    tree.insert(4);
    tree.insert(5);
    tree.insert(6);
    tree.insert(7);
    tree.insert(8);
    tree.insert(9);

    expect(tree.toString()).toBe('1,2,3,4,5,6,7,8,9');
    expect(tree.root.value).toBe(4);
    expect(tree.root.height).toBe(3);
    expect(tree.root.balanceFactor).toBe(-1);

    tree.remove(8);

    expect(tree.root.value).toBe(4);
    expect(tree.root.balanceFactor).toBe(-1);

    tree.remove(9);

    expect(tree.contains(8)).toBeFalsy();
    expect(tree.contains(9)).toBeFalsy();
    expect(tree.toString()).toBe('1,2,3,4,5,6,7');
    expect(tree.root.value).toBe(4);
    expect(tree.root.height).toBe(2);
    expect(tree.root.balanceFactor).toBe(0);
  });
});
```

Listing 280: AvlTree.js

```
import BinarySearchTree from '../binary-search-tree/BinarySearchTree';

export default class AvlTree extends BinarySearchTree {
  /**
   * @param {*} value
   */
  insert(value) {
    // Do the normal BST insert.
    super.insert(value);

    // Let's move up to the root and check balance factors along the way.
    let currentNode = this.root.find(value);
    while (currentNode) {
      this.balance(currentNode);
      currentNode = currentNode.parent;
    }
  }

  /**
   * @param {*} value
   * @return {boolean}
   */
  remove(value) {
    // Do standard BST removal.
    super.remove(value);

    // Balance the tree starting from the root node.
    this.balance(this.root);
  }

  /**
   * @param {BinarySearchTreeNode} node
   */
  balance(node) {
    // If balance factor is not OK then try to balance the node.
    if (node.balanceFactor > 1) {
      // Left rotation.
      if (node.left.balanceFactor > 0) {
        // Left-Left rotation
        this.rotateLeftLeft(node);
      } else if (node.left.balanceFactor < 0) {
        // Left-Right rotation.
        this.rotateLeftRight(node);
      }
    } else if (node.balanceFactor < -1) {
      // Right rotation.
      if (node.right.balanceFactor < 0) {
        // Right-Right rotation
        this.rotateRightRight(node);
      } else if (node.right.balanceFactor > 0) {
        // Right-Left rotation.
        this.rotateRightLeft(node);
      }
    }
  }

  /**
   * @param {BinarySearchTreeNode} rootNode
   */
  rotateLeftLeft(rootNode) {
    // Detach left node from root node.
    const leftNode = rootNode.left;
    rootNode.setLeft(null);

    // Make left node to be a child of rootNode's parent.
    if (rootNode.parent) {
      rootNode.parent.setLeft(leftNode);
    } else if (rootNode === this.root) {
      // If root node is root then make left node to be a new root.
      this.root = leftNode;
    }

    // If left node has a right child then detach it and
    // attach it as a left child for rootNode.
    if (leftNode.right) {
      rootNode.setLeft(leftNode.right);
    }

    // Attach rootNode to the right of leftNode.
```

```
    leftNode.setRight(rootNode);
  }

  /**
   * @param {BinarySearchTreeNode} rootNode
   */
  rotateLeftRight(rootNode) {
    // Detach left node from rootNode since it is going to be replaced.
    const leftNode = rootNode.left;
    rootNode.setLeft(null);

    // Detach right node from leftNode.
    const leftRightNode = leftNode.right;
    leftNode.setRight(null);

    // Preserve leftRightNode's left subtree.
    if (leftRightNode.left) {
      leftNode.setRight(leftRightNode.left);
      leftRightNode.setLeft(null);
    }

    // Attach leftRightNode to the rootNode.
    rootNode.setLeft(leftRightNode);

    // Attach leftNode as left node for leftRight node.
    leftRightNode.setLeft(leftNode);

    // Do left-left rotation.
    this.rotateLeftLeft(rootNode);
  }

  /**
   * @param {BinarySearchTreeNode} rootNode
   */
  rotateRightLeft(rootNode) {
    // Detach right node from rootNode since it is going to be replaced.
    const rightNode = rootNode.right;
    rootNode.setRight(null);

    // Detach left node from rightNode.
    const rightLeftNode = rightNode.left;
    rightNode.setLeft(null);

    if (rightLeftNode.right) {
      rightNode.setLeft(rightLeftNode.right);
      rightLeftNode.setRight(null);
    }

    // Attach rightLeftNode to the rootNode.
    rootNode.setRight(rightLeftNode);

    // Attach rightNode as right node for rightLeft node.
    rightLeftNode.setRight(rightNode);

    // Do right-right rotation.
    this.rotateRightRight(rootNode);
  }

  /**
   * @param {BinarySearchTreeNode} rootNode
   */
  rotateRightRight(rootNode) {
    // Detach right node from root node.
    const rightNode = rootNode.right;
    rootNode.setRight(null);

    // Make right node to be a child of rootNode's parent.
    if (rootNode.parent) {
      rootNode.parent.setRight(rightNode);
    } else if (rootNode === this.root) {
      // If root node is root then make right node to be a new root.
      this.root = rightNode;
    }

    // If right node has a left child then detach it and
    // attach it as a right child for rootNode.
    if (rightNode.left) {
      rootNode.setRight(rightNode.left);
    }

    // Attach rootNode to the left of rightNode.
    rightNode.setLeft(rootNode);
```

```
  }
}
```

Listing 281: BinarySearchTree.test.js

```
import BinarySearchTree from '../BinarySearchTree';

describe('BinarySearchTree', () => {
  it('should create binary search tree', () => {
    const bst = new BinarySearchTree();

    expect(bst).toBeDefined();
    expect(bst.root).toBeDefined();
    expect(bst.root.value).toBeNull();
    expect(bst.root.left).toBeNull();
    expect(bst.root.right).toBeNull();
  });

  it('should insert values', () => {
    const bst = new BinarySearchTree();

    const insertedNode1 = bst.insert(10);
    const insertedNode2 = bst.insert(20);
    bst.insert(5);

    expect(bst.toString()).toBe('5,10,20');
    expect(insertedNode1.value).toBe(10);
    expect(insertedNode2.value).toBe(20);
  });

  it('should check if value exists', () => {
    const bst = new BinarySearchTree();

    bst.insert(10);
    bst.insert(20);
    bst.insert(5);

    expect(bst.contains(20)).toBe(true);
    expect(bst.contains(40)).toBe(false);
  });

  it('should remove nodes', () => {
    const bst = new BinarySearchTree();

    bst.insert(10);
    bst.insert(20);
    bst.insert(5);

    expect(bst.toString()).toBe('5,10,20');

    const removed1 = bst.remove(5);
    expect(bst.toString()).toBe('10,20');
    expect(removed1).toBe(true);

    const removed2 = bst.remove(20);
    expect(bst.toString()).toBe('10');
    expect(removed2).toBe(true);
  });

  it('should insert object values', () => {
    const nodeValueCompareFunction = (a, b) => {
      const normalizedA = a || { value: null };
      const normalizedB = b || { value: null };

      if (normalizedA.value === normalizedB.value) {
        return 0;
      }

      return normalizedA.value < normalizedB.value ? -1 : 1;
    };

    const obj1 = { key: 'obj1', value: 1, toString: () => 'obj1' };
    const obj2 = { key: 'obj2', value: 2, toString: () => 'obj2' };
    const obj3 = { key: 'obj3', value: 3, toString: () => 'obj3' };

    const bst = new BinarySearchTree(nodeValueCompareFunction);

    bst.insert(obj2);
    bst.insert(obj3);
    bst.insert(obj1);

    expect(bst.toString()).toBe('obj1,obj2,obj3');
  });
```

```
  it('should be traversed to sorted array', () => {
    const bst = new BinarySearchTree();

    bst.insert(10);
    bst.insert(-10);
    bst.insert(20);
    bst.insert(-20);
    bst.insert(25);
    bst.insert(6);

    expect(bst.toString()).toBe('-20,-10,6,10,20,25');
    expect(bst.root.height).toBe(2);

    bst.insert(4);

    expect(bst.toString()).toBe('-20,-10,4,6,10,20,25');
    expect(bst.root.height).toBe(3);
  });
});
```

Listing 282: BinarySearchTreeNode.test.js

```
import BinarySearchTreeNode from '../BinarySearchTreeNode';

describe('BinarySearchTreeNode', () => {
  it('should create binary search tree', () => {
    const bstNode = new BinarySearchTreeNode(2);

    expect(bstNode.value).toBe(2);
    expect(bstNode.left).toBeNull();
    expect(bstNode.right).toBeNull();
  });

  it('should insert in itself if it is empty', () => {
    const bstNode = new BinarySearchTreeNode();
    bstNode.insert(1);

    expect(bstNode.value).toBe(1);
    expect(bstNode.left).toBeNull();
    expect(bstNode.right).toBeNull();
  });

  it('should insert nodes in correct order', () => {
    const bstNode = new BinarySearchTreeNode(2);
    const insertedNode1 = bstNode.insert(1);

    expect(insertedNode1.value).toBe(1);
    expect(bstNode.toString()).toBe('1,2');
    expect(bstNode.contains(1)).toBe(true);
    expect(bstNode.contains(3)).toBe(false);

    const insertedNode2 = bstNode.insert(3);

    expect(insertedNode2.value).toBe(3);
    expect(bstNode.toString()).toBe('1,2,3');
    expect(bstNode.contains(3)).toBe(true);
    expect(bstNode.contains(4)).toBe(false);

    bstNode.insert(7);

    expect(bstNode.toString()).toBe('1,2,3,7');
    expect(bstNode.contains(7)).toBe(true);
    expect(bstNode.contains(8)).toBe(false);

    bstNode.insert(4);

    expect(bstNode.toString()).toBe('1,2,3,4,7');
    expect(bstNode.contains(4)).toBe(true);
    expect(bstNode.contains(8)).toBe(false);

    bstNode.insert(6);

    expect(bstNode.toString()).toBe('1,2,3,4,6,7');
    expect(bstNode.contains(6)).toBe(true);
    expect(bstNode.contains(8)).toBe(false);
  });

  it('should not insert duplicates', () => {
    const bstNode = new BinarySearchTreeNode(2);
    bstNode.insert(1);

    expect(bstNode.toString()).toBe('1,2');
    expect(bstNode.contains(1)).toBe(true);
    expect(bstNode.contains(3)).toBe(false);

    bstNode.insert(1);

    expect(bstNode.toString()).toBe('1,2');
    expect(bstNode.contains(1)).toBe(true);
    expect(bstNode.contains(3)).toBe(false);
  });

  it('should find min node', () => {
    const node = new BinarySearchTreeNode(10);

    node.insert(20);
    node.insert(30);
    node.insert(5);
    node.insert(40);
    node.insert(1);
```

```
    expect(node.findMin()).not.toBeNull();
    expect(node.findMin().value).toBe(1);
  });

  it('should be possible to attach meta information to binary search tree nodes', () => {
    const node = new BinarySearchTreeNode(10);

    node.insert(20);
    const node1 = node.insert(30);
    node.insert(5);
    node.insert(40);
    const node2 = node.insert(1);

    node.meta.set('color', 'red');
    node1.meta.set('color', 'black');
    node2.meta.set('color', 'white');

    expect(node.meta.get('color')).toBe('red');

    expect(node.findMin()).not.toBeNull();
    expect(node.findMin().value).toBe(1);
    expect(node.findMin().meta.get('color')).toBe('white');
    expect(node.find(30).meta.get('color')).toBe('black');
  });

  it('should find node', () => {
    const node = new BinarySearchTreeNode(10);

    node.insert(20);
    node.insert(30);
    node.insert(5);
    node.insert(40);
    node.insert(1);

    expect(node.find(6)).toBeNull();
    expect(node.find(5)).not.toBeNull();
    expect(node.find(5).value).toBe(5);
  });

  it('should remove leaf nodes', () => {
    const bstRootNode = new BinarySearchTreeNode();

    bstRootNode.insert(10);
    bstRootNode.insert(20);
    bstRootNode.insert(5);

    expect(bstRootNode.toString()).toBe('5,10,20');

    const removed1 = bstRootNode.remove(5);
    expect(bstRootNode.toString()).toBe('10,20');
    expect(removed1).toBe(true);

    const removed2 = bstRootNode.remove(20);
    expect(bstRootNode.toString()).toBe('10');
    expect(removed2).toBe(true);
  });

  it('should remove nodes with one child', () => {
    const bstRootNode = new BinarySearchTreeNode();

    bstRootNode.insert(10);
    bstRootNode.insert(20);
    bstRootNode.insert(5);
    bstRootNode.insert(30);

    expect(bstRootNode.toString()).toBe('5,10,20,30');

    bstRootNode.remove(20);
    expect(bstRootNode.toString()).toBe('5,10,30');

    bstRootNode.insert(1);
    expect(bstRootNode.toString()).toBe('1,5,10,30');

    bstRootNode.remove(5);
    expect(bstRootNode.toString()).toBe('1,10,30');
  });

  it('should remove nodes with two children', () => {
    const bstRootNode = new BinarySearchTreeNode();

    bstRootNode.insert(10);
    bstRootNode.insert(20);
```

```
    bstRootNode.insert(5);
    bstRootNode.insert(30);
    bstRootNode.insert(15);
    bstRootNode.insert(25);

    expect(bstRootNode.toString()).toBe('5,10,15,20,25,30');
    expect(bstRootNode.find(20).left.value).toBe(15);
    expect(bstRootNode.find(20).right.value).toBe(30);

    bstRootNode.remove(20);
    expect(bstRootNode.toString()).toBe('5,10,15,25,30');

    bstRootNode.remove(15);
    expect(bstRootNode.toString()).toBe('5,10,25,30');

    bstRootNode.remove(10);
    expect(bstRootNode.toString()).toBe('5,25,30');
    expect(bstRootNode.value).toBe(25);

    bstRootNode.remove(25);
    expect(bstRootNode.toString()).toBe('5,30');

    bstRootNode.remove(5);
    expect(bstRootNode.toString()).toBe('30');
  });

  it('should remove node with no parent', () => {
    const bstRootNode = new BinarySearchTreeNode();
    expect(bstRootNode.toString()).toBe('');

    bstRootNode.insert(1);
    bstRootNode.insert(2);
    expect(bstRootNode.toString()).toBe('1,2');

    bstRootNode.remove(1);
    expect(bstRootNode.toString()).toBe('2');

    bstRootNode.remove(2);
    expect(bstRootNode.toString()).toBe('');
  });

  it('should throw error when trying to remove not existing node', () => {
    const bstRootNode = new BinarySearchTreeNode();

    bstRootNode.insert(10);
    bstRootNode.insert(20);

    function removeNotExistingElementFromTree() {
      bstRootNode.remove(30);
    }

    expect(removeNotExistingElementFromTree).toThrow();
  });

  it('should be possible to use objects as node values', () => {
    const nodeValueComparatorCallback = (a, b) => {
      const normalizedA = a || { value: null };
      const normalizedB = b || { value: null };

      if (normalizedA.value === normalizedB.value) {
        return 0;
      }

      return normalizedA.value < normalizedB.value ? -1 : 1;
    };

    const obj1 = { key: 'obj1', value: 1, toString: () => 'obj1' };
    const obj2 = { key: 'obj2', value: 2, toString: () => 'obj2' };
    const obj3 = { key: 'obj3', value: 3, toString: () => 'obj3' };

    const bstNode = new BinarySearchTreeNode(obj2, nodeValueComparatorCallback);
    bstNode.insert(obj1);

    expect(bstNode.toString()).toBe('obj1,obj2');
    expect(bstNode.contains(obj1)).toBe(true);
    expect(bstNode.contains(obj3)).toBe(false);

    bstNode.insert(obj3);

    expect(bstNode.toString()).toBe('obj1,obj2,obj3');
    expect(bstNode.contains(obj3)).toBe(true);
```

```
    expect(bstNode.findMin().value).toEqual(obj1);
  });

  it('should abandon removed node', () => {
    const rootNode = new BinarySearchTreeNode('foo');
    rootNode.insert('bar');
    const childNode = rootNode.find('bar');
    rootNode.remove('bar');

    expect(childNode.parent).toBeNull();
  });
});
```

Listing 283: BinarySearchTree.js

```
import BinarySearchTreeNode from './BinarySearchTreeNode';

export default class BinarySearchTree {
  /**
   * @param {function} [nodeValueCompareFunction]
   */
  constructor(nodeValueCompareFunction) {
    this.root = new BinarySearchTreeNode(null, nodeValueCompareFunction);

    // Steal node comparator from the root.
    this.nodeComparator = this.root.nodeComparator;
  }

  /**
   * @param {*} value
   * @return {BinarySearchTreeNode}
   */
  insert(value) {
    return this.root.insert(value);
  }

  /**
   * @param {*} value
   * @return {boolean}
   */
  contains(value) {
    return this.root.contains(value);
  }

  /**
   * @param {*} value
   * @return {boolean}
   */
  remove(value) {
    return this.root.remove(value);
  }

  /**
   * @return {string}
   */
  toString() {
    return this.root.toString();
  }
}
```

Listing 284: BinarySearchTreeNode.js

```
import BinaryTreeNode from '../BinaryTreeNode';
import Comparator from '../../../utils/comparator/Comparator';

export default class BinarySearchTreeNode extends BinaryTreeNode {
  /**
   * @param {*} [value] - node value.
   * @param {function} [compareFunction] - comparator function for node values.
   */
  constructor(value = null, compareFunction = undefined) {
    super(value);

    // This comparator is used to compare node values with each other.
    this.compareFunction = compareFunction;
    this.nodeValueComparator = new Comparator(compareFunction);
  }

  /**
   * @param {*} value
   * @return {BinarySearchTreeNode}
   */
  insert(value) {
    if (this.nodeValueComparator.equal(this.value, null)) {
      this.value = value;

      return this;
    }

    if (this.nodeValueComparator.lessThan(value, this.value)) {
      // Insert to the left.
      if (this.left) {
        return this.left.insert(value);
      }

      const newNode = new BinarySearchTreeNode(value, this.compareFunction);
      this.setLeft(newNode);

      return newNode;
    }

    if (this.nodeValueComparator.greaterThan(value, this.value)) {
      // Insert to the right.
      if (this.right) {
        return this.right.insert(value);
      }

      const newNode = new BinarySearchTreeNode(value, this.compareFunction);
      this.setRight(newNode);

      return newNode;
    }

    return this;
  }

  /**
   * @param {*} value
   * @return {BinarySearchTreeNode}
   */
  find(value) {
    // Check the root.
    if (this.nodeValueComparator.equal(this.value, value)) {
      return this;
    }

    if (this.nodeValueComparator.lessThan(value, this.value) && this.left) {
      // Check left nodes.
      return this.left.find(value);
    }

    if (this.nodeValueComparator.greaterThan(value, this.value) && this.right) {
      // Check right nodes.
      return this.right.find(value);
    }

    return null;
  }

  /**
   * @param {*} value
```

```
   * @return {boolean}
   */
  contains(value) {
    return !!this.find(value);
  }

  /**
   * @param {*} value
   * @return {boolean}
   */
  remove(value) {
    const nodeToRemove = this.find(value);

    if (!nodeToRemove) {
      throw new Error('Item not found in the tree');
    }

    const { parent } = nodeToRemove;

    if (!nodeToRemove.left && !nodeToRemove.right) {
      // Node is a leaf and thus has no children.
      if (parent) {
        // Node has a parent. Just remove the pointer to this node from the parent.
        parent.removeChild(nodeToRemove);
      } else {
        // Node has no parent. Just erase current node value.
        nodeToRemove.setValue(undefined);
      }
    } else if (nodeToRemove.left && nodeToRemove.right) {
      // Node has two children.
      // Find the next biggest value (minimum value in the right branch)
      // and replace current value node with that next biggest value.
      const nextBiggerNode = nodeToRemove.right.findMin();
      if (!this.nodeComparator.equal(nextBiggerNode, nodeToRemove.right)) {
        this.remove(nextBiggerNode.value);
        nodeToRemove.setValue(nextBiggerNode.value);
      } else {
        // In case if next right value is the next bigger one and it doesn't have left child
        // then just replace node that is going to be deleted with the right node.
        nodeToRemove.setValue(nodeToRemove.right.value);
        nodeToRemove.setRight(nodeToRemove.right.right);
      }
    } else {
      // Node has only one child.
      // Make this child to be a direct child of current node's parent.
      /** @var BinarySearchTreeNode */
      const childNode = nodeToRemove.left || nodeToRemove.right;

      if (parent) {
        parent.replaceChild(nodeToRemove, childNode);
      } else {
        BinaryTreeNode.copyNode(childNode, nodeToRemove);
      }
    }

    // Clear the parent of removed node.
    nodeToRemove.parent = null;

    return true;
  }

  /**
   * @return {BinarySearchTreeNode}
   */
  findMin() {
    if (!this.left) {
      return this;
    }

    return this.left.findMin();
  }
}
```

Listing 285: BinaryTreeNode.js

```
 import Comparator from '../../utils/comparator/Comparator';
import HashTable from '../hash-table/HashTable';

export default class BinaryTreeNode {
  /**
   * @param {*} [value] - node value.
   */
  constructor(value = null) {
    this.left = null;
    this.right = null;
    this.parent = null;
    this.value = value;

    // Any node related meta information may be stored here.
    this.meta = new HashTable();

    // This comparator is used to compare binary tree nodes with each other.
    this.nodeComparator = new Comparator();
  }

  /**
   * @return {number}
   */
  get leftHeight() {
    if (!this.left) {
      return 0;
    }

    return this.left.height + 1;
  }

  /**
   * @return {number}
   */
  get rightHeight() {
    if (!this.right) {
      return 0;
    }

    return this.right.height + 1;
  }

  /**
   * @return {number}
   */
  get height() {
    return Math.max(this.leftHeight, this.rightHeight);
  }

  /**
   * @return {number}
   */
  get balanceFactor() {
    return this.leftHeight - this.rightHeight;
  }

  /**
   * Get parent's sibling if it exists.
   * @return {BinaryTreeNode}
   */
  get uncle() {
    // Check if current node has parent.
    if (!this.parent) {
      return undefined;
    }

    // Check if current node has grand-parent.
    if (!this.parent.parent) {
      return undefined;
    }

    // Check if grand-parent has two children.
    if (!this.parent.parent.left || !this.parent.parent.right) {
      return undefined;
    }

    // So for now we know that current node has grand-parent and this
    // grand-parent has two children. Let's find out who is the uncle.
    if (this.nodeComparator.equal(this.parent, this.parent.parent.left)) {
```

```
    // Right one is an uncle.
    return this.parent.parent.right;
  }

  // Left one is an uncle.
  return this.parent.parent.left;
}

/**
 * @param {*} value
 * @return {BinaryTreeNode}
 */
setValue(value) {
  this.value = value;

  return this;
}

/**
 * @param {BinaryTreeNode} node
 * @return {BinaryTreeNode}
 */
setLeft(node) {
  // Reset parent for left node since it is going to be detached.
  if (this.left) {
    this.left.parent = null;
  }

  // Attach new node to the left.
  this.left = node;

  // Make current node to be a parent for new left one.
  if (this.left) {
    this.left.parent = this;
  }

  return this;
}

/**
 * @param {BinaryTreeNode} node
 * @return {BinaryTreeNode}
 */
setRight(node) {
  // Reset parent for right node since it is going to be detached.
  if (this.right) {
    this.right.parent = null;
  }

  // Attach new node to the right.
  this.right = node;

  // Make current node to be a parent for new right one.
  if (node) {
    this.right.parent = this;
  }

  return this;
}

/**
 * @param {BinaryTreeNode} nodeToRemove
 * @return {boolean}
 */
removeChild(nodeToRemove) {
  if (this.left && this.nodeComparator.equal(this.left, nodeToRemove)) {
    this.left = null;
    return true;
  }

  if (this.right && this.nodeComparator.equal(this.right, nodeToRemove)) {
    this.right = null;
    return true;
  }

  return false;
}

/**
 * @param {BinaryTreeNode} nodeToReplace
 * @param {BinaryTreeNode} replacementNode
 * @return {boolean}
```

```
   */
  replaceChild(nodeToReplace, replacementNode) {
    if (!nodeToReplace || !replacementNode) {
      return false;
    }

    if (this.left && this.nodeComparator.equal(this.left, nodeToReplace)) {
      this.left = replacementNode;
      return true;
    }

    if (this.right && this.nodeComparator.equal(this.right, nodeToReplace)) {
      this.right = replacementNode;
      return true;
    }

    return false;
  }

  /**
   * @param {BinaryTreeNode} sourceNode
   * @param {BinaryTreeNode} targetNode
   */
  static copyNode(sourceNode, targetNode) {
    targetNode.setValue(sourceNode.value);
    targetNode.setLeft(sourceNode.left);
    targetNode.setRight(sourceNode.right);
  }

  /**
   * @return {*[]}
   */
  traverseInOrder() {
    let traverse = [];

    // Add left node.
    if (this.left) {
      traverse = traverse.concat(this.left.traverseInOrder());
    }

    // Add root.
    traverse.push(this.value);

    // Add right node.
    if (this.right) {
      traverse = traverse.concat(this.right.traverseInOrder());
    }

    return traverse;
  }

  /**
   * @return {string}
   */
  toString() {
    return this.traverseInOrder().toString();
  }
}
```

Listing 286: FenwickTree.test.js

```
import FenwickTree from '../FenwickTree';

describe('FenwickTree', () => {
  it('should create empty fenwick tree of correct size', () => {
    const tree1 = new FenwickTree(5);
    expect(tree1.treeArray.length).toBe(5 + 1);

    for (let i = 0; i < 5; i += 1) {
      expect(tree1.treeArray[i]).toBe(0);
    }

    const tree2 = new FenwickTree(50);
    expect(tree2.treeArray.length).toBe(50 + 1);
  });

  it('should create correct fenwick tree', () => {
    const inputArray = [3, 2, -1, 6, 5, 4, -3, 3, 7, 2, 3];

    const tree = new FenwickTree(inputArray.length);
    expect(tree.treeArray.length).toBe(inputArray.length + 1);

    inputArray.forEach((value, index) => {
      tree.increase(index + 1, value);
    });

    expect(tree.treeArray).toEqual([0, 3, 5, -1, 10, 5, 9, -3, 19, 7, 9, 3]);

    expect(tree.query(1)).toBe(3);
    expect(tree.query(2)).toBe(5);
    expect(tree.query(3)).toBe(4);
    expect(tree.query(4)).toBe(10);
    expect(tree.query(5)).toBe(15);
    expect(tree.query(6)).toBe(19);
    expect(tree.query(7)).toBe(16);
    expect(tree.query(8)).toBe(19);
    expect(tree.query(9)).toBe(26);
    expect(tree.query(10)).toBe(28);
    expect(tree.query(11)).toBe(31);

    expect(tree.queryRange(1, 1)).toBe(3);
    expect(tree.queryRange(1, 2)).toBe(5);
    expect(tree.queryRange(2, 4)).toBe(7);
    expect(tree.queryRange(6, 9)).toBe(11);

    tree.increase(3, 1);

    expect(tree.query(1)).toBe(3);
    expect(tree.query(2)).toBe(5);
    expect(tree.query(3)).toBe(5);
    expect(tree.query(4)).toBe(11);
    expect(tree.query(5)).toBe(16);
    expect(tree.query(6)).toBe(20);
    expect(tree.query(7)).toBe(17);
    expect(tree.query(8)).toBe(20);
    expect(tree.query(9)).toBe(27);
    expect(tree.query(10)).toBe(29);
    expect(tree.query(11)).toBe(32);

    expect(tree.queryRange(1, 1)).toBe(3);
    expect(tree.queryRange(1, 2)).toBe(5);
    expect(tree.queryRange(2, 4)).toBe(8);
    expect(tree.queryRange(6, 9)).toBe(11);
  });

  it('should correctly execute queries', () => {
    const tree = new FenwickTree(5);

    tree.increase(1, 4);
    tree.increase(3, 7);

    expect(tree.query(1)).toBe(4);
    expect(tree.query(3)).toBe(11);
    expect(tree.query(5)).toBe(11);
    expect(tree.queryRange(2, 3)).toBe(7);

    tree.increase(2, 5);
    expect(tree.query(5)).toBe(16);

    tree.increase(1, 3);
```

```
    expect(tree.queryRange(1, 1)).toBe(7);
    expect(tree.query(5)).toBe(19);
    expect(tree.queryRange(1, 5)).toBe(19);
  });

  it('should throw exceptions', () => {
    const tree = new FenwickTree(5);

    const increaseAtInvalidLowIndex = () => {
      tree.increase(0, 1);
    };

    const increaseAtInvalidHighIndex = () => {
      tree.increase(10, 1);
    };

    const queryInvalidLowIndex = () => {
      tree.query(0);
    };

    const queryInvalidHighIndex = () => {
      tree.query(10);
    };

    const rangeQueryInvalidIndex = () => {
      tree.queryRange(3, 2);
    };

    expect(increaseAtInvalidLowIndex).toThrowError();
    expect(increaseAtInvalidHighIndex).toThrowError();
    expect(queryInvalidLowIndex).toThrowError();
    expect(queryInvalidHighIndex).toThrowError();
    expect(rangeQueryInvalidIndex).toThrowError();
  });
});
```

Listing 287: FenwickTree.js

```
export default class FenwickTree {
  /**
   * Constructor creates empty fenwick tree of size 'arraySize',
   * however, array size is size+1, because index is 1-based.
   *
   * @param  {number} arraySize
   */
  constructor(arraySize) {
    this.arraySize = arraySize;

    // Fill tree array with zeros.
    this.treeArray = Array(this.arraySize + 1).fill(0);
  }

  /**
   * Adds value to existing value at position.
   *
   * @param  {number} position
   * @param  {number} value
   * @return {FenwickTree}
   */
  increase(position, value) {
    if (position < 1 || position > this.arraySize) {
      throw new Error('Position is out of allowed range');
    }

    for (let i = position; i <= this.arraySize; i += (i & -i)) {
      this.treeArray[i] += value;
    }

    return this;
  }

  /**
   * Query sum from index 1 to position.
   *
   * @param  {number} position
   * @return {number}
   */
  query(position) {
    if (position < 1 || position > this.arraySize) {
      throw new Error('Position is out of allowed range');
    }

    let sum = 0;

    for (let i = position; i > 0; i -= (i & -i)) {
      sum += this.treeArray[i];
    }

    return sum;
  }

  /**
   * Query sum from index leftIndex to rightIndex.
   *
   * @param  {number} leftIndex
   * @param  {number} rightIndex
   * @return {number}
   */
  queryRange(leftIndex, rightIndex) {
    if (leftIndex > rightIndex) {
      throw new Error('Left index can not be greater than right one');
    }

    if (leftIndex === 1) {
      return this.query(rightIndex);
    }

    return this.query(rightIndex) - this.query(leftIndex - 1);
  }
}
```

Listing 288: RedBlackTree.test.js

```
import RedBlackTree from '../RedBlackTree';

describe('RedBlackTree', () => {
  it('should always color first inserted node as black', () => {
    const tree = new RedBlackTree();

    const firstInsertedNode = tree.insert(10);

    expect(tree.isNodeColored(firstInsertedNode)).toBe(true);
    expect(tree.isNodeBlack(firstInsertedNode)).toBe(true);
    expect(tree.isNodeRed(firstInsertedNode)).toBe(false);

    expect(tree.toString()).toBe('10');
    expect(tree.root.height).toBe(0);
  });

  it('should always color new leaf node as red', () => {
    const tree = new RedBlackTree();

    const firstInsertedNode = tree.insert(10);
    const secondInsertedNode = tree.insert(15);
    const thirdInsertedNode = tree.insert(5);

    expect(tree.isNodeBlack(firstInsertedNode)).toBe(true);
    expect(tree.isNodeRed(secondInsertedNode)).toBe(true);
    expect(tree.isNodeRed(thirdInsertedNode)).toBe(true);

    expect(tree.toString()).toBe('5,10,15');
    expect(tree.root.height).toBe(1);
  });

  it('should balance itself', () => {
    const tree = new RedBlackTree();

    tree.insert(5);
    tree.insert(10);
    tree.insert(15);
    tree.insert(20);
    tree.insert(25);
    tree.insert(30);

    expect(tree.toString()).toBe('5,10,15,20,25,30');
    expect(tree.root.height).toBe(3);
  });

  it('should balance itself when parent is black', () => {
    const tree = new RedBlackTree();

    const node1 = tree.insert(10);

    expect(tree.isNodeBlack(node1)).toBe(true);

    const node2 = tree.insert(-10);

    expect(tree.isNodeBlack(node1)).toBe(true);
    expect(tree.isNodeRed(node2)).toBe(true);

    const node3 = tree.insert(20);

    expect(tree.isNodeBlack(node1)).toBe(true);
    expect(tree.isNodeRed(node2)).toBe(true);
    expect(tree.isNodeRed(node3)).toBe(true);

    const node4 = tree.insert(-20);

    expect(tree.isNodeBlack(node1)).toBe(true);
    expect(tree.isNodeBlack(node2)).toBe(true);
    expect(tree.isNodeBlack(node3)).toBe(true);
    expect(tree.isNodeRed(node4)).toBe(true);

    const node5 = tree.insert(25);

    expect(tree.isNodeBlack(node1)).toBe(true);
    expect(tree.isNodeBlack(node2)).toBe(true);
    expect(tree.isNodeBlack(node3)).toBe(true);
    expect(tree.isNodeRed(node4)).toBe(true);
    expect(tree.isNodeRed(node5)).toBe(true);

    const node6 = tree.insert(6);
```

```
    expect(tree.isNodeBlack(node1)).toBe(true);
    expect(tree.isNodeBlack(node2)).toBe(true);
    expect(tree.isNodeBlack(node3)).toBe(true);
    expect(tree.isNodeRed(node4)).toBe(true);
    expect(tree.isNodeRed(node5)).toBe(true);
    expect(tree.isNodeRed(node6)).toBe(true);

    expect(tree.toString()).toBe('-20,-10,6,10,20,25');
    expect(tree.root.height).toBe(2);

    const node7 = tree.insert(4);

    expect(tree.root.left.value).toEqual(node2.value);

    expect(tree.toString()).toBe('-20,-10,4,6,10,20,25');
    expect(tree.root.height).toBe(3);

    expect(tree.isNodeBlack(node1)).toBe(true);
    expect(tree.isNodeRed(node2)).toBe(true);
    expect(tree.isNodeBlack(node3)).toBe(true);
    expect(tree.isNodeBlack(node4)).toBe(true);
    expect(tree.isNodeBlack(node4)).toBe(true);
    expect(tree.isNodeRed(node5)).toBe(true);
    expect(tree.isNodeBlack(node6)).toBe(true);
    expect(tree.isNodeRed(node7)).toBe(true);
  });

  it('should balance itself when uncle is red', () => {
    const tree = new RedBlackTree();

    const node1 = tree.insert(10);
    const node2 = tree.insert(-10);
    const node3 = tree.insert(20);
    const node4 = tree.insert(-20);
    const node5 = tree.insert(6);
    const node6 = tree.insert(15);
    const node7 = tree.insert(25);
    const node8 = tree.insert(2);
    const node9 = tree.insert(8);

    expect(tree.toString()).toBe('-20,-10,2,6,8,10,15,20,25');
    expect(tree.root.height).toBe(3);

    expect(tree.isNodeBlack(node1)).toBe(true);
    expect(tree.isNodeRed(node2)).toBe(true);
    expect(tree.isNodeBlack(node3)).toBe(true);
    expect(tree.isNodeBlack(node4)).toBe(true);
    expect(tree.isNodeBlack(node5)).toBe(true);
    expect(tree.isNodeRed(node6)).toBe(true);
    expect(tree.isNodeRed(node7)).toBe(true);
    expect(tree.isNodeRed(node8)).toBe(true);
    expect(tree.isNodeRed(node9)).toBe(true);

    const node10 = tree.insert(4);

    expect(tree.toString()).toBe('-20,-10,2,4,6,8,10,15,20,25');
    expect(tree.root.height).toBe(3);

    expect(tree.root.value).toBe(node5.value);

    expect(tree.isNodeBlack(node5)).toBe(true);
    expect(tree.isNodeRed(node1)).toBe(true);
    expect(tree.isNodeRed(node2)).toBe(true);
    expect(tree.isNodeRed(node10)).toBe(true);
    expect(tree.isNodeRed(node6)).toBe(true);
    expect(tree.isNodeRed(node7)).toBe(true);
    expect(tree.isNodeBlack(node4)).toBe(true);
    expect(tree.isNodeBlack(node8)).toBe(true);
    expect(tree.isNodeBlack(node9)).toBe(true);
    expect(tree.isNodeBlack(node3)).toBe(true);
  });

  it('should do left-left rotation', () => {
    const tree = new RedBlackTree();

    const node1 = tree.insert(10);
    const node2 = tree.insert(-10);
    const node3 = tree.insert(20);
    const node4 = tree.insert(7);
    const node5 = tree.insert(15);
```

```
    expect(tree.toString()).toBe('-10,7,10,15,20');
    expect(tree.root.height).toBe(2);

    expect(tree.isNodeBlack(node1)).toBe(true);
    expect(tree.isNodeBlack(node2)).toBe(true);
    expect(tree.isNodeBlack(node3)).toBe(true);
    expect(tree.isNodeRed(node4)).toBe(true);
    expect(tree.isNodeRed(node5)).toBe(true);

    const node6 = tree.insert(13);

    expect(tree.toString()).toBe('-10,7,10,13,15,20');
    expect(tree.root.height).toBe(2);

    expect(tree.isNodeBlack(node1)).toBe(true);
    expect(tree.isNodeBlack(node2)).toBe(true);
    expect(tree.isNodeBlack(node5)).toBe(true);
    expect(tree.isNodeRed(node4)).toBe(true);
    expect(tree.isNodeRed(node6)).toBe(true);
    expect(tree.isNodeRed(node3)).toBe(true);
  });

  it('should do left-right rotation', () => {
    const tree = new RedBlackTree();

    const node1 = tree.insert(10);
    const node2 = tree.insert(-10);
    const node3 = tree.insert(20);
    const node4 = tree.insert(7);
    const node5 = tree.insert(15);

    expect(tree.toString()).toBe('-10,7,10,15,20');
    expect(tree.root.height).toBe(2);

    expect(tree.isNodeBlack(node1)).toBe(true);
    expect(tree.isNodeBlack(node2)).toBe(true);
    expect(tree.isNodeBlack(node3)).toBe(true);
    expect(tree.isNodeRed(node4)).toBe(true);
    expect(tree.isNodeRed(node5)).toBe(true);

    const node6 = tree.insert(17);

    expect(tree.toString()).toBe('-10,7,10,15,17,20');
    expect(tree.root.height).toBe(2);

    expect(tree.isNodeBlack(node1)).toBe(true);
    expect(tree.isNodeBlack(node2)).toBe(true);
    expect(tree.isNodeBlack(node6)).toBe(true);
    expect(tree.isNodeRed(node4)).toBe(true);
    expect(tree.isNodeRed(node5)).toBe(true);
    expect(tree.isNodeRed(node3)).toBe(true);
  });

  it('should do recoloring, left-left and left-right rotation', () => {
    const tree = new RedBlackTree();

    const node1 = tree.insert(10);
    const node2 = tree.insert(-10);
    const node3 = tree.insert(20);
    const node4 = tree.insert(-20);
    const node5 = tree.insert(6);
    const node6 = tree.insert(15);
    const node7 = tree.insert(30);
    const node8 = tree.insert(1);
    const node9 = tree.insert(9);

    expect(tree.toString()).toBe('-20,-10,1,6,9,10,15,20,30');
    expect(tree.root.height).toBe(3);

    expect(tree.isNodeBlack(node1)).toBe(true);
    expect(tree.isNodeRed(node2)).toBe(true);
    expect(tree.isNodeBlack(node3)).toBe(true);
    expect(tree.isNodeBlack(node4)).toBe(true);
    expect(tree.isNodeBlack(node5)).toBe(true);
    expect(tree.isNodeRed(node6)).toBe(true);
    expect(tree.isNodeRed(node7)).toBe(true);
    expect(tree.isNodeRed(node8)).toBe(true);
    expect(tree.isNodeRed(node9)).toBe(true);

    tree.insert(4);

    expect(tree.toString()).toBe('-20,-10,1,4,6,9,10,15,20,30');
```

```
    expect(tree.root.height).toBe(3);
  });

  it('should do right-left rotation', () => {
    const tree = new RedBlackTree();

    const node1 = tree.insert(10);
    const node2 = tree.insert(-10);
    const node3 = tree.insert(20);
    const node4 = tree.insert(-20);
    const node5 = tree.insert(6);
    const node6 = tree.insert(30);

    expect(tree.toString()).toBe('-20,-10,6,10,20,30');
    expect(tree.root.height).toBe(2);

    expect(tree.isNodeBlack(node1)).toBe(true);
    expect(tree.isNodeBlack(node2)).toBe(true);
    expect(tree.isNodeBlack(node3)).toBe(true);
    expect(tree.isNodeRed(node4)).toBe(true);
    expect(tree.isNodeRed(node5)).toBe(true);
    expect(tree.isNodeRed(node6)).toBe(true);

    const node7 = tree.insert(25);

    const rightNode = tree.root.right;
    const rightLeftNode = rightNode.left;
    const rightRightNode = rightNode.right;

    expect(rightNode.value).toBe(node7.value);
    expect(rightLeftNode.value).toBe(node3.value);
    expect(rightRightNode.value).toBe(node6.value);

    expect(tree.toString()).toBe('-20,-10,6,10,20,25,30');
    expect(tree.root.height).toBe(2);

    expect(tree.isNodeBlack(node1)).toBe(true);
    expect(tree.isNodeBlack(node2)).toBe(true);
    expect(tree.isNodeBlack(node7)).toBe(true);
    expect(tree.isNodeRed(node4)).toBe(true);
    expect(tree.isNodeRed(node5)).toBe(true);
    expect(tree.isNodeRed(node3)).toBe(true);
    expect(tree.isNodeRed(node6)).toBe(true);
  });

  it('should do left-left rotation with left grand-parent', () => {
    const tree = new RedBlackTree();

    tree.insert(20);
    tree.insert(15);
    tree.insert(25);
    tree.insert(10);
    tree.insert(5);

    expect(tree.toString()).toBe('5,10,15,20,25');
    expect(tree.root.height).toBe(2);
  });

  it('should do right-right rotation with left grand-parent', () => {
    const tree = new RedBlackTree();

    tree.insert(20);
    tree.insert(15);
    tree.insert(25);
    tree.insert(17);
    tree.insert(19);

    expect(tree.toString()).toBe('15,17,19,20,25');
    expect(tree.root.height).toBe(2);
  });

  it('should throw an error when trying to remove node', () => {
    const removeNodeFromRedBlackTree = () => {
      const tree = new RedBlackTree();

      tree.remove(1);
    };

    expect(removeNodeFromRedBlackTree).toThrowError();
  });
});
```

Listing 289: RedBlackTree.js

```
import BinarySearchTree from '../binary-search-tree/BinarySearchTree';

// Possible colors of red-black tree nodes.
const RED_BLACK_TREE_COLORS = {
  red: 'red',
  black: 'black',
};

// Color property name in meta information of the nodes.
const COLOR_PROP_NAME = 'color';

export default class RedBlackTree extends BinarySearchTree {
  /**
   * @param {*} value
   * @return {BinarySearchTreeNode}
   */
  insert(value) {
    const insertedNode = super.insert(value);

    // if (!this.root.left && !this.root.right) {
    if (this.nodeComparator.equal(insertedNode, this.root)) {
      // Make root to always be black.
      this.makeNodeBlack(insertedNode);
    } else {
      // Make all newly inserted nodes to be red.
      this.makeNodeRed(insertedNode);
    }

    // Check all conditions and balance the node.
    this.balance(insertedNode);

    return insertedNode;
  }

  /**
   * @param {*} value
   * @return {boolean}
   */
  remove(value) {
    throw new Error(`Can't remove ${value}. Remove method is not implemented yet`);
  }

  /**
   * @param {BinarySearchTreeNode} node
   */
  balance(node) {
    // If it is a root node then nothing to balance here.
    if (this.nodeComparator.equal(node, this.root)) {
      return;
    }

    // If the parent is black then done. Nothing to balance here.
    if (this.isNodeBlack(node.parent)) {
      return;
    }

    const grandParent = node.parent.parent;

    if (node.uncle && this.isNodeRed(node.uncle)) {
      // If node has red uncle then we need to do RECOLORING.

      // Recolor parent and uncle to black.
      this.makeNodeBlack(node.uncle);
      this.makeNodeBlack(node.parent);

      if (!this.nodeComparator.equal(grandParent, this.root)) {
        // Recolor grand-parent to red if it is not root.
        this.makeNodeRed(grandParent);
      } else {
        // If grand-parent is black root don't do anything.
        // Since root already has two black sibling that we've just recolored.
        return;
      }

      // Now do further checking for recolored grand-parent.
      this.balance(grandParent);
    } else if (!node.uncle || this.isNodeBlack(node.uncle)) {
      // If node uncle is black or absent then we need to do ROTATIONS.
```

```
    if (grandParent) {
      // Grand parent that we will receive after rotations.
      let newGrandParent;

      if (this.nodeComparator.equal(grandParent.left, node.parent)) {
        // Left case.
        if (this.nodeComparator.equal(node.parent.left, node)) {
          // Left-left case.
          newGrandParent = this.leftLeftRotation(grandParent);
        } else {
          // Left-right case.
          newGrandParent = this.leftRightRotation(grandParent);
        }
      } else {
        // Right case.
        if (this.nodeComparator.equal(node.parent.right, node)) {
          // Right-right case.
          newGrandParent = this.rightRightRotation(grandParent);
        } else {
          // Right-left case.
          newGrandParent = this.rightLeftRotation(grandParent);
        }
      }

      // Set newGrandParent as a root if it doesn't have parent.
      if (newGrandParent && newGrandParent.parent === null) {
        this.root = newGrandParent;

        // Recolor root into black.
        this.makeNodeBlack(this.root);
      }

      // Check if new grand parent don't violate red-black-tree rules.
      this.balance(newGrandParent);
    }
  }
}

/**
 * Left Left Case (p is left child of g and x is left child of p)
 * @param {BinarySearchTreeNode|BinaryTreeNode} grandParentNode
 * @return {BinarySearchTreeNode}
 */
leftLeftRotation(grandParentNode) {
  // Memorize the parent of grand-parent node.
  const grandGrandParent = grandParentNode.parent;

  // Check what type of sibling is our grandParentNode is (left or right).
  let grandParentNodeIsLeft;
  if (grandGrandParent) {
    grandParentNodeIsLeft = this.nodeComparator.equal(grandGrandParent.left, grandParentNode);
  }

  // Memorize grandParentNode's left node.
  const parentNode = grandParentNode.left;

  // Memorize parent's right node since we're going to transfer it to
  // grand parent's left subtree.
  const parentRightNode = parentNode.right;

  // Make grandParentNode to be right child of parentNode.
  parentNode.setRight(grandParentNode);

  // Move child's right subtree to grandParentNode's left subtree.
  grandParentNode.setLeft(parentRightNode);

  // Put parentNode node in place of grandParentNode.
  if (grandGrandParent) {
    if (grandParentNodeIsLeft) {
      grandGrandParent.setLeft(parentNode);
    } else {
      grandGrandParent.setRight(parentNode);
    }
  } else {
    // Make parent node a root
    parentNode.parent = null;
  }

  // Swap colors of granParent and parent nodes.
  this.swapNodeColors(parentNode, grandParentNode);

  // Return new root node.
```

```
    return parentNode;
  }

  /**
   * Left Right Case (p is left child of g and x is right child of p)
   * @param {BinarySearchTreeNode|BinaryTreeNode} grandParentNode
   * @return {BinarySearchTreeNode}
   */
  leftRightRotation(grandParentNode) {
    // Memorize left and left-right nodes.
    const parentNode = grandParentNode.left;
    const childNode = parentNode.right;

    // We need to memorize child left node to prevent losing
    // left child subtree. Later it will be re-assigned to
    // parent's right sub-tree.
    const childLeftNode = childNode.left;

    // Make parentNode to be a left child of childNode node.
    childNode.setLeft(parentNode);

    // Move child's left subtree to parent's right subtree.
    parentNode.setRight(childLeftNode);

    // Put left-right node in place of left node.
    grandParentNode.setLeft(childNode);

    // Now we're ready to do left-left rotation.
    return this.leftLeftRotation(grandParentNode);
  }

  /**
   * Right Right Case (p is right child of g and x is right child of p)
   * @param {BinarySearchTreeNode|BinaryTreeNode} grandParentNode
   * @return {BinarySearchTreeNode}
   */
  rightRightRotation(grandParentNode) {
    // Memorize the parent of grand-parent node.
    const grandGrandParent = grandParentNode.parent;

    // Check what type of sibling is our grandParentNode is (left or right).
    let grandParentNodeIsLeft;
    if (grandGrandParent) {
      grandParentNodeIsLeft = this.nodeComparator.equal(grandGrandParent.left, grandParentNode);
    }

    // Memorize grandParentNode's right node.
    const parentNode = grandParentNode.right;

    // Memorize parent's left node since we're going to transfer it to
    // grand parent's right subtree.
    const parentLeftNode = parentNode.left;

    // Make grandParentNode to be left child of parentNode.
    parentNode.setLeft(grandParentNode);

    // Transfer all left nodes from parent to right sub-tree of grandparent.
    grandParentNode.setRight(parentLeftNode);

    // Put parentNode node in place of grandParentNode.
    if (grandGrandParent) {
      if (grandParentNodeIsLeft) {
        grandGrandParent.setLeft(parentNode);
      } else {
        grandGrandParent.setRight(parentNode);
      }
    } else {
      // Make parent node a root.
      parentNode.parent = null;
    }

    // Swap colors of granParent and parent nodes.
    this.swapNodeColors(parentNode, grandParentNode);

    // Return new root node.
    return parentNode;
  }

  /**
   * Right Left Case (p is right child of g and x is left child of p)
   * @param {BinarySearchTreeNode|BinaryTreeNode} grandParentNode
   * @return {BinarySearchTreeNode}
```

```
   */
  rightLeftRotation(grandParentNode) {
    // Memorize right and right-left nodes.
    const parentNode = grandParentNode.right;
    const childNode = parentNode.left;

    // We need to memorize child right node to prevent losing
    // right child subtree. Later it will be re-assigned to
    // parent's left sub-tree.
    const childRightNode = childNode.right;

    // Make parentNode to be a right child of childNode.
    childNode.setRight(parentNode);

    // Move child's right subtree to parent's left subtree.
    parentNode.setLeft(childRightNode);

    // Put childNode node in place of parentNode.
    grandParentNode.setRight(childNode);

    // Now we're ready to do right-right rotation.
    return this.rightRightRotation(grandParentNode);
  }

  /**
   * @param {BinarySearchTreeNode|BinaryTreeNode} node
   * @return {BinarySearchTreeNode}
   */
  makeNodeRed(node) {
    node.meta.set(COLOR_PROP_NAME, RED_BLACK_TREE_COLORS.red);

    return node;
  }

  /**
   * @param {BinarySearchTreeNode|BinaryTreeNode} node
   * @return {BinarySearchTreeNode}
   */
  makeNodeBlack(node) {
    node.meta.set(COLOR_PROP_NAME, RED_BLACK_TREE_COLORS.black);

    return node;
  }

  /**
   * @param {BinarySearchTreeNode|BinaryTreeNode} node
   * @return {boolean}
   */
  isNodeRed(node) {
    return node.meta.get(COLOR_PROP_NAME) === RED_BLACK_TREE_COLORS.red;
  }

  /**
   * @param {BinarySearchTreeNode|BinaryTreeNode} node
   * @return {boolean}
   */
  isNodeBlack(node) {
    return node.meta.get(COLOR_PROP_NAME) === RED_BLACK_TREE_COLORS.black;
  }

  /**
   * @param {BinarySearchTreeNode|BinaryTreeNode} node
   * @return {boolean}
   */
  isNodeColored(node) {
    return this.isNodeRed(node) || this.isNodeBlack(node);
  }

  /**
   * @param {BinarySearchTreeNode|BinaryTreeNode} firstNode
   * @param {BinarySearchTreeNode|BinaryTreeNode} secondNode
   */
  swapNodeColors(firstNode, secondNode) {
    const firstColor = firstNode.meta.get(COLOR_PROP_NAME);
    const secondColor = secondNode.meta.get(COLOR_PROP_NAME);

    firstNode.meta.set(COLOR_PROP_NAME, secondColor);
    secondNode.meta.set(COLOR_PROP_NAME, firstColor);
  }
}
```

Listing 290: SegmentTree.test.js

```
import SegmentTree from '../SegmentTree';

describe('SegmentTree', () => {
  it('should build tree for input array #0 with length of power of two', () => {
    const array = [-1, 2];
    const segmentTree = new SegmentTree(array, Math.min, Infinity);

    expect(segmentTree.segmentTree).toEqual([-1, -1, 2]);
    expect(segmentTree.segmentTree.length).toBe((2 * array.length) - 1);
  });

  it('should build tree for input array #1 with length of power of two', () => {
    const array = [-1, 2, 4, 0];
    const segmentTree = new SegmentTree(array, Math.min, Infinity);

    expect(segmentTree.segmentTree).toEqual([-1, -1, 0, -1, 2, 4, 0]);
    expect(segmentTree.segmentTree.length).toBe((2 * array.length) - 1);
  });

  it('should build tree for input array #0 with length not of power of two', () => {
    const array = [0, 1, 2];
    const segmentTree = new SegmentTree(array, Math.min, Infinity);

    expect(segmentTree.segmentTree).toEqual([0, 0, 2, 0, 1, null, null]);
    expect(segmentTree.segmentTree.length).toBe((2 * 4) - 1);
  });

  it('should build tree for input array #1 with length not of power of two', () => {
    const array = [-1, 3, 4, 0, 2, 1];
    const segmentTree = new SegmentTree(array, Math.min, Infinity);

    expect(segmentTree.segmentTree).toEqual([
      -1, -1, 0, -1, 4, 0, 1, -1, 3, null, null, 0, 2, null, null,
    ]);
    expect(segmentTree.segmentTree.length).toBe((2 * 8) - 1);
  });

  it('should build max array', () => {
    const array = [-1, 2, 4, 0];
    const segmentTree = new SegmentTree(array, Math.max, -Infinity);

    expect(segmentTree.segmentTree).toEqual([4, 2, 4, -1, 2, 4, 0]);
    expect(segmentTree.segmentTree.length).toBe((2 * array.length) - 1);
  });

  it('should build sum array', () => {
    const array = [-1, 2, 4, 0];
    const segmentTree = new SegmentTree(array, (a, b) => (a + b), 0);

    expect(segmentTree.segmentTree).toEqual([5, 1, 4, -1, 2, 4, 0]);
    expect(segmentTree.segmentTree.length).toBe((2 * array.length) - 1);
  });

  it('should do min range query on power of two length array', () => {
    const array = [-1, 3, 4, 0, 2, 1];
    const segmentTree = new SegmentTree(array, Math.min, Infinity);

    expect(segmentTree.rangeQuery(0, 5)).toBe(-1);
    expect(segmentTree.rangeQuery(0, 2)).toBe(-1);
    expect(segmentTree.rangeQuery(1, 3)).toBe(0);
    expect(segmentTree.rangeQuery(2, 4)).toBe(0);
    expect(segmentTree.rangeQuery(4, 5)).toBe(1);
    expect(segmentTree.rangeQuery(2, 2)).toBe(4);
  });

  it('should do min range query on not power of two length array', () => {
    const array = [-1, 2, 4, 0];
    const segmentTree = new SegmentTree(array, Math.min, Infinity);

    expect(segmentTree.rangeQuery(0, 4)).toBe(-1);
    expect(segmentTree.rangeQuery(0, 1)).toBe(-1);
    expect(segmentTree.rangeQuery(1, 3)).toBe(0);
    expect(segmentTree.rangeQuery(1, 2)).toBe(2);
    expect(segmentTree.rangeQuery(2, 3)).toBe(0);
    expect(segmentTree.rangeQuery(2, 2)).toBe(4);
  });

  it('should do max range query', () => {
    const array = [-1, 3, 4, 0, 2, 1];
```

```
    const segmentTree = new SegmentTree(array, Math.max, -Infinity);

    expect(segmentTree.rangeQuery(0, 5)).toBe(4);
    expect(segmentTree.rangeQuery(0, 1)).toBe(3);
    expect(segmentTree.rangeQuery(1, 3)).toBe(4);
    expect(segmentTree.rangeQuery(2, 4)).toBe(4);
    expect(segmentTree.rangeQuery(4, 5)).toBe(2);
    expect(segmentTree.rangeQuery(3, 3)).toBe(0);
  });

  it('should do sum range query', () => {
    const array = [-1, 3, 4, 0, 2, 1];
    const segmentTree = new SegmentTree(array, (a, b) => (a + b), 0);

    expect(segmentTree.rangeQuery(0, 5)).toBe(9);
    expect(segmentTree.rangeQuery(0, 1)).toBe(2);
    expect(segmentTree.rangeQuery(1, 3)).toBe(7);
    expect(segmentTree.rangeQuery(2, 4)).toBe(6);
    expect(segmentTree.rangeQuery(4, 5)).toBe(3);
    expect(segmentTree.rangeQuery(3, 3)).toBe(0);
  });
});
```

Listing 291: SegmentTree.js

```
import isPowerOfTwo from '../../../algorithms/math/is-power-of-two/isPowerOfTwo';

export default class SegmentTree {
  /**
   * @param {number[]} inputArray
   * @param {function} operation - binary function (i.e. sum, min)
   * @param {number} operationFallback - operation fallback value (i.e. 0 for sum, Infinity for min)
   */
  constructor(inputArray, operation, operationFallback) {
    this.inputArray = inputArray;
    this.operation = operation;
    this.operationFallback = operationFallback;

    // Init array representation of segment tree.
    this.segmentTree = this.initSegmentTree(this.inputArray);

    this.buildSegmentTree();
  }

  /**
   * @param {number[]} inputArray
   * @return {number[]}
   */
  initSegmentTree(inputArray) {
    let segmentTreeArrayLength;
    const inputArrayLength = inputArray.length;

    if (isPowerOfTwo(inputArrayLength)) {
      // If original array length is a power of two.
      segmentTreeArrayLength = (2 * inputArrayLength) - 1;
    } else {
      // If original array length is not a power of two then we need to find
      // next number that is a power of two and use it to calculate
      // tree array size. This is happens because we need to fill empty children
      // in perfect binary tree with nulls.And those nulls need extra space.
      const currentPower = Math.floor(Math.log2(inputArrayLength));
      const nextPower = currentPower + 1;
      const nextPowerOfTwoNumber = 2 ** nextPower;
      segmentTreeArrayLength = (2 * nextPowerOfTwoNumber) - 1;
    }

    return new Array(segmentTreeArrayLength).fill(null);
  }

  /**
   * Build segment tree.
   */
  buildSegmentTree() {
    const leftIndex = 0;
    const rightIndex = this.inputArray.length - 1;
    const position = 0;
    this.buildTreeRecursively(leftIndex, rightIndex, position);
  }

  /**
   * Build segment tree recursively.
   *
   * @param {number} leftInputIndex
   * @param {number} rightInputIndex
   * @param {number} position
   */
  buildTreeRecursively(leftInputIndex, rightInputIndex, position) {
    // If low input index and high input index are equal that would mean
    // the we have finished splitting and we are already came to the leaf
    // of the segment tree. We need to copy this leaf value from input
    // array to segment tree.
    if (leftInputIndex === rightInputIndex) {
      this.segmentTree[position] = this.inputArray[leftInputIndex];
      return;
    }

    // Split input array on two halves and process them recursively.
    const middleIndex = Math.floor((leftInputIndex + rightInputIndex) / 2);
    // Process left half of the input array.
    this.buildTreeRecursively(leftInputIndex, middleIndex, this.getLeftChildIndex(position));
    // Process right half of the input array.
    this.buildTreeRecursively(middleIndex + 1, rightInputIndex, this.getRightChildIndex(position));

    // Once every tree leaf is not empty we're able to build tree bottom up using
```

```
  // provided operation function.
  this.segmentTree[position] = this.operation(
    this.segmentTree[this.getLeftChildIndex(position)],
    this.segmentTree[this.getRightChildIndex(position)],
  );
}

/**
 * Do range query on segment tree in context of this.operation function.
 *
 * @param {number} queryLeftIndex
 * @param {number} queryRightIndex
 * @return {number}
 */
rangeQuery(queryLeftIndex, queryRightIndex) {
  const leftIndex = 0;
  const rightIndex = this.inputArray.length - 1;
  const position = 0;

  return this.rangeQueryRecursive(
    queryLeftIndex,
    queryRightIndex,
    leftIndex,
    rightIndex,
    position,
  );
}

/**
 * Do range query on segment tree recursively in context of this.operation function.
 *
 * @param {number} queryLeftIndex - left index of the query
 * @param {number} queryRightIndex - right index of the query
 * @param {number} leftIndex - left index of input array segment
 * @param {number} rightIndex - right index of input array segment
 * @param {number} position - root position in binary tree
 * @return {number}
 */
rangeQueryRecursive(queryLeftIndex, queryRightIndex, leftIndex, rightIndex, position) {
  if (queryLeftIndex <= leftIndex && queryRightIndex >= rightIndex) {
    // Total overlap.
    return this.segmentTree[position];
  }

  if (queryLeftIndex > rightIndex || queryRightIndex < leftIndex) {
    // No overlap.
    return this.operationFallback;
  }

  // Partial overlap.
  const middleIndex = Math.floor((leftIndex + rightIndex) / 2);

  const leftOperationResult = this.rangeQueryRecursive(
    queryLeftIndex,
    queryRightIndex,
    leftIndex,
    middleIndex,
    this.getLeftChildIndex(position),
  );

  const rightOperationResult = this.rangeQueryRecursive(
    queryLeftIndex,
    queryRightIndex,
    middleIndex + 1,
    rightIndex,
    this.getRightChildIndex(position),
  );

  return this.operation(leftOperationResult, rightOperationResult);
}

/**
 * Left child index.
 * @param {number} parentIndex
 * @return {number}
 */
getLeftChildIndex(parentIndex) {
  return (2 * parentIndex) + 1;
}

/**
 * Right child index.
```

```
   * @param {number} parentIndex
   * @return {number}
   */
  getRightChildIndex(parentIndex) {
    return (2 * parentIndex) + 2;
  }
}
```

Listing 292: Trie.test.js

```
import Trie from '../Trie';

describe('Trie', () => {
  it('should create trie', () => {
    const trie = new Trie();

    expect(trie).toBeDefined();
    expect(trie.head.toString()).toBe('*');
  });

  it('should add words to trie', () => {
    const trie = new Trie();

    trie.addWord('cat');

    expect(trie.head.toString()).toBe('*:c');
    expect(trie.head.getChild('c').toString()).toBe('c:a');

    trie.addWord('car');
    expect(trie.head.toString()).toBe('*:c');
    expect(trie.head.getChild('c').toString()).toBe('c:a');
    expect(trie.head.getChild('c').getChild('a').toString()).toBe('a:t,r');
    expect(trie.head.getChild('c').getChild('a').getChild('t').toString()).toBe('t*');
  });

  it('should delete words from trie', () => {
    const trie = new Trie();

    trie.addWord('carpet');
    trie.addWord('car');
    trie.addWord('cat');
    trie.addWord('cart');
    expect(trie.doesWordExist('carpet')).toBe(true);
    expect(trie.doesWordExist('car')).toBe(true);
    expect(trie.doesWordExist('cart')).toBe(true);
    expect(trie.doesWordExist('cat')).toBe(true);

    // Try to delete not-existing word first.
    trie.deleteWord('carpool');
    expect(trie.doesWordExist('carpet')).toBe(true);
    expect(trie.doesWordExist('car')).toBe(true);
    expect(trie.doesWordExist('cart')).toBe(true);
    expect(trie.doesWordExist('cat')).toBe(true);

    trie.deleteWord('carpet');
    expect(trie.doesWordExist('carpet')).toEqual(false);
    expect(trie.doesWordExist('car')).toEqual(true);
    expect(trie.doesWordExist('cart')).toBe(true);
    expect(trie.doesWordExist('cat')).toBe(true);

    trie.deleteWord('cat');
    expect(trie.doesWordExist('car')).toEqual(true);
    expect(trie.doesWordExist('cart')).toBe(true);
    expect(trie.doesWordExist('cat')).toBe(false);

    trie.deleteWord('car');
    expect(trie.doesWordExist('car')).toEqual(false);
    expect(trie.doesWordExist('cart')).toBe(true);

    trie.deleteWord('cart');
    expect(trie.doesWordExist('car')).toEqual(false);
    expect(trie.doesWordExist('cart')).toBe(false);
  });

  it('should suggests next characters', () => {
    const trie = new Trie();

    trie.addWord('cat');
    trie.addWord('cats');
    trie.addWord('car');
    trie.addWord('caption');

    expect(trie.suggestNextCharacters('ca')).toEqual(['t', 'r', 'p']);
    expect(trie.suggestNextCharacters('cat')).toEqual(['s']);
    expect(trie.suggestNextCharacters('cab')).toBeNull();
  });

  it('should check if word exists', () => {
    const trie = new Trie();
```

```
    trie.addWord('cat');
    trie.addWord('cats');
    trie.addWord('carpet');
    trie.addWord('car');
    trie.addWord('caption');

    expect(trie.doesWordExist('cat')).toBe(true);
    expect(trie.doesWordExist('cats')).toBe(true);
    expect(trie.doesWordExist('carpet')).toBe(true);
    expect(trie.doesWordExist('car')).toBe(true);
    expect(trie.doesWordExist('cap')).toBe(false);
    expect(trie.doesWordExist('call')).toBe(false);
  });
});
```

Listing 293: TrieNode.test.js

```
import TrieNode from '../TrieNode';

describe('TrieNode', () => {
  it('should create trie node', () => {
    const trieNode = new TrieNode('c', true);

    expect(trieNode.character).toBe('c');
    expect(trieNode.isCompleteWord).toBe(true);
    expect(trieNode.toString()).toBe('c*');
  });

  it('should add child nodes', () => {
    const trieNode = new TrieNode('c');

    trieNode.addChild('a', true);
    trieNode.addChild('o');

    expect(trieNode.toString()).toBe('c:a,o');
  });

  it('should get child nodes', () => {
    const trieNode = new TrieNode('c');

    trieNode.addChild('a');
    trieNode.addChild('o');

    expect(trieNode.getChild('a').toString()).toBe('a');
    expect(trieNode.getChild('a').character).toBe('a');
    expect(trieNode.getChild('o').toString()).toBe('o');
    expect(trieNode.getChild('b')).toBeUndefined();
  });

  it('should check if node has children', () => {
    const trieNode = new TrieNode('c');

    expect(trieNode.hasChildren()).toBe(false);

    trieNode.addChild('a');

    expect(trieNode.hasChildren()).toBe(true);
  });

  it('should check if node has specific child', () => {
    const trieNode = new TrieNode('c');

    trieNode.addChild('a');
    trieNode.addChild('o');

    expect(trieNode.hasChild('a')).toBe(true);
    expect(trieNode.hasChild('o')).toBe(true);
    expect(trieNode.hasChild('b')).toBe(false);
  });

  it('should suggest next children', () => {
    const trieNode = new TrieNode('c');

    trieNode.addChild('a');
    trieNode.addChild('o');

    expect(trieNode.suggestChildren()).toEqual(['a', 'o']);
  });

  it('should delete child node if the child node has NO children', () => {
    const trieNode = new TrieNode('c');
    trieNode.addChild('a');
    expect(trieNode.hasChild('a')).toBe(true);

    trieNode.removeChild('a');
    expect(trieNode.hasChild('a')).toBe(false);
  });

  it('should NOT delete child node if the child node has children', () => {
    const trieNode = new TrieNode('c');
    trieNode.addChild('a');
    const childNode = trieNode.getChild('a');
    childNode.addChild('r');

    trieNode.removeChild('a');
    expect(trieNode.hasChild('a')).toEqual(true);
```

```
  });

  it('should NOT delete child node if the child node completes a word', () => {
    const trieNode = new TrieNode('c');
    const IS_COMPLETE_WORD = true;
    trieNode.addChild('a', IS_COMPLETE_WORD);

    trieNode.removeChild('a');
    expect(trieNode.hasChild('a')).toEqual(true);
  });
});
```

Listing 294: Trie.js

```
 import TrieNode from './TrieNode';

// Character that we will use for trie tree root.
const HEAD_CHARACTER = '*';

export default class Trie {
  constructor() {
    this.head = new TrieNode(HEAD_CHARACTER);
  }

  /**
   * @param {string} word
   * @return {Trie}
   */
  addWord(word) {
    const characters = Array.from(word);
    let currentNode = this.head;

    for (let charIndex = 0; charIndex < characters.length; charIndex += 1) {
      const isComplete = charIndex === characters.length - 1;
      currentNode = currentNode.addChild(characters[charIndex], isComplete);
    }

    return this;
  }

  /**
   * @param {string} word
   * @return {Trie}
   */
  deleteWord(word) {
    const depthFirstDelete = (currentNode, charIndex = 0) => {
      if (charIndex >= word.length) {
        // Return if we're trying to delete the character that is out of word's scope.
        return;
      }

      const character = word[charIndex];
      const nextNode = currentNode.getChild(character);

      if (nextNode == null) {
        // Return if we're trying to delete a word that has not been added to the Trie.
        return;
      }

      // Go deeper.
      depthFirstDelete(nextNode, charIndex + 1);

      // Since we're going to delete a word let's un-mark its last character isCompleteWord flag.
      if (charIndex === (word.length - 1)) {
        nextNode.isCompleteWord = false;
      }

      // childNode is deleted only if:
      // - childNode has NO children
      // - childNode.isCompleteWord === false
      currentNode.removeChild(character);
    };

    // Start depth-first deletion from the head node.
    depthFirstDelete(this.head);

    return this;
  }

  /**
   * @param {string} word
   * @return {string[]}
   */
  suggestNextCharacters(word) {
    const lastCharacter = this.getLastCharacterNode(word);

    if (!lastCharacter) {
      return null;
    }

    return lastCharacter.suggestChildren();
  }
```

```
  /**
   * Check if complete word exists in Trie.
   *
   * @param {string} word
   * @return {boolean}
   */
  doesWordExist(word) {
    const lastCharacter = this.getLastCharacterNode(word);

    return !!lastCharacter && lastCharacter.isCompleteWord;
  }

  /**
   * @param {string} word
   * @return {TrieNode}
   */
  getLastCharacterNode(word) {
    const characters = Array.from(word);
    let currentNode = this.head;

    for (let charIndex = 0; charIndex < characters.length; charIndex += 1) {
      if (!currentNode.hasChild(characters[charIndex])) {
        return null;
      }

      currentNode = currentNode.getChild(characters[charIndex]);
    }

    return currentNode;
  }
}
```

Listing 295: TrieNode.js

```
 import HashTable from '../hash-table/HashTable';

export default class TrieNode {
  /**
   * @param {string} character
   * @param {boolean} isCompleteWord
   */
  constructor(character, isCompleteWord = false) {
    this.character = character;
    this.isCompleteWord = isCompleteWord;
    this.children = new HashTable();
  }

  /**
   * @param {string} character
   * @return {TrieNode}
   */
  getChild(character) {
    return this.children.get(character);
  }

  /**
   * @param {string} character
   * @param {boolean} isCompleteWord
   * @return {TrieNode}
   */
  addChild(character, isCompleteWord = false) {
    if (!this.children.has(character)) {
      this.children.set(character, new TrieNode(character, isCompleteWord));
    }

    const childNode = this.children.get(character);

    // In cases similar to adding "car" after "carpet" we need to mark "r" character as complete.
    childNode.isCompleteWord = childNode.isCompleteWord || isCompleteWord;

    return childNode;
  }

  /**
   * @param {string} character
   * @return {TrieNode}
   */
  removeChild(character) {
    const childNode = this.getChild(character);

    // Delete childNode only if:
    // - childNode has NO children,
    // - childNode.isCompleteWord === false.
    if (
      childNode
      && !childNode.isCompleteWord
      && !childNode.hasChildren()
    ) {
      this.children.delete(character);
    }

    return this;
  }

  /**
   * @param {string} character
   * @return {boolean}
   */
  hasChild(character) {
    return this.children.has(character);
  }

  /**
   * Check whether current TrieNode has children or not.
   * @return {boolean}
   */
  hasChildren() {
    return this.children.getKeys().length !== 0;
  }

  /**
   * @return {string[]}
   */
```

```
  suggestChildren() {
    return [...this.children.getKeys()];
  }

  /**
   * @return {string}
   */
  toString() {
    let childrenAsString = this.suggestChildren().toString();
    childrenAsString = childrenAsString ? `:${childrenAsString}` : '';
    const isCompleteString = this.isCompleteWord ? '*' : '';

    return `${this.character}${isCompleteString}${childrenAsString}`;
  }
}
```

Listing 296: playground.test.js

```
 describe('playground', () => {
  it('should perform playground tasks', () => {
    // Place your playground tests here.
  });
});
```

Listing 297: playground.js

```
// Place your playground code here.
```

Listing 298: Comparator.test.js

```
import Comparator from '../Comparator';

describe('Comparator', () => {
  it('should compare with default comparator function', () => {
    const comparator = new Comparator();

    expect(comparator.equal(0, 0)).toBe(true);
    expect(comparator.equal(0, 1)).toBe(false);
    expect(comparator.equal('a', 'a')).toBe(true);
    expect(comparator.lessThan(1, 2)).toBe(true);
    expect(comparator.lessThan(-1, 2)).toBe(true);
    expect(comparator.lessThan('a', 'b')).toBe(true);
    expect(comparator.lessThan('a', 'ab')).toBe(true);
    expect(comparator.lessThan(10, 2)).toBe(false);
    expect(comparator.lessThanOrEqual(10, 2)).toBe(false);
    expect(comparator.lessThanOrEqual(1, 1)).toBe(true);
    expect(comparator.lessThanOrEqual(0, 0)).toBe(true);
    expect(comparator.greaterThan(0, 0)).toBe(false);
    expect(comparator.greaterThan(10, 0)).toBe(true);
    expect(comparator.greaterThanOrEqual(10, 0)).toBe(true);
    expect(comparator.greaterThanOrEqual(10, 10)).toBe(true);
    expect(comparator.greaterThanOrEqual(0, 10)).toBe(false);
  });

  it('should compare with custom comparator function', () => {
    const comparator = new Comparator((a, b) => {
      if (a.length === b.length) {
        return 0;
      }

      return a.length < b.length ? -1 : 1;
    });

    expect(comparator.equal('a', 'b')).toBe(true);
    expect(comparator.equal('a', '')).toBe(false);
    expect(comparator.lessThan('b', 'aa')).toBe(true);
    expect(comparator.greaterThanOrEqual('a', 'aa')).toBe(false);
    expect(comparator.greaterThanOrEqual('aa', 'a')).toBe(true);
    expect(comparator.greaterThanOrEqual('a', 'a')).toBe(true);

    comparator.reverse();

    expect(comparator.equal('a', 'b')).toBe(true);
    expect(comparator.equal('a', '')).toBe(false);
    expect(comparator.lessThan('b', 'aa')).toBe(false);
    expect(comparator.greaterThanOrEqual('a', 'aa')).toBe(true);
    expect(comparator.greaterThanOrEqual('aa', 'a')).toBe(false);
    expect(comparator.greaterThanOrEqual('a', 'a')).toBe(true);
  });
});
```

Listing 299: Comparator.js

```
export default class Comparator {
  /**
   * @param {function(a: *, b: *)} [compareFunction] - It may be custom compare function that, let's
   * say may compare custom objects together.
   */
  constructor(compareFunction) {
    this.compare = compareFunction || Comparator.defaultCompareFunction;
  }

  /**
   * Default comparison function. It just assumes that "a" and "b" are strings or numbers.
   * @param {(string|number)} a
   * @param {(string|number)} b
   * @returns {number}
   */
  static defaultCompareFunction(a, b) {
    if (a === b) {
      return 0;
    }

    return a < b ? -1 : 1;
  }

  /**
   * Checks if two variables are equal.
   * @param {*} a
   * @param {*} b
   * @return {boolean}
   */
  equal(a, b) {
    return this.compare(a, b) === 0;
  }

  /**
   * Checks if variable "a" is less than "b".
   * @param {*} a
   * @param {*} b
   * @return {boolean}
   */
  lessThan(a, b) {
    return this.compare(a, b) < 0;
  }

  /**
   * Checks if variable "a" is greater than "b".
   * @param {*} a
   * @param {*} b
   * @return {boolean}
   */
  greaterThan(a, b) {
    return this.compare(a, b) > 0;
  }

  /**
   * Checks if variable "a" is less than or equal to "b".
   * @param {*} a
   * @param {*} b
   * @return {boolean}
   */
  lessThanOrEqual(a, b) {
    return this.lessThan(a, b) || this.equal(a, b);
  }

  /**
   * Checks if variable "a" is greater than or equal to "b".
   * @param {*} a
   * @param {*} b
   * @return {boolean}
   */
  greaterThanOrEqual(a, b) {
    return this.greaterThan(a, b) || this.equal(a, b);
  }

  /**
   * Reverses the comparison order.
   */
  reverse() {
    const compareOriginal = this.compare;
    this.compare = (a, b) => compareOriginal(b, a);
```

```
  }
}
```