Report 1

Team information (B23-ISE-02):

- Team leader: Kirill Efimovich (i.efimovich@innopolis.university)
- Team member 1: Arsen Galiev (a.galiev@innopolis.university)
- Team member 2: Asqar Arslanov (a.arslanov@innopolis.university)
- Team member 3: Ilya-Linh Nguen (i.nguen@innopolis.university)

Link to the product:

- https://github.com/quintet-sdr/optimization-a1

Programming language:

- TypeScript (Bun)
- To launch the code: `$ bun test`

Linear programming problem:

- Maximization
- Approximation accuracy $\epsilon = 0.001$

Tests:

(1) **Input:**

- Objective function: $F(x_1, x_2) = 10x_1 + 20x_2$
- Constraints: $\begin{cases} -x_1 + 2x_2 \leq 15 \\ x_1 + x_2 \leq 12 \\ 5x_1 + 3x_2 \leq 45 \\ x_1, x_2 \geq 0 \end{cases}$

Output:

- Decision variables: $\{3, 9\}$
- Maximum value: 210

(2) **Input:**

- Objective function: ...
- Constraints: ...

Output:

- Unbounded solution

Code:

src/util.ts

```
export function arrayOf<T>(n: number, item: () => T): T[] {
  return new Array(n).fill(undefined).map(item);
}

export function prettyPrintWith(
  tableau: number[][],
  rowNames: string[],
  colNames: string[],
  precision: number,
): void {
  prettyPrint([
    ["Basic", ...colNames],
    ...tableau.map((row, j) => [
      rowNames[j],
      ...row.map((num) => num.toFixed(precision)),
    ]),
  ]);
}

export function printHeading(text: string): void {
  const lines = new Array(24).fill("-").join("");
  console.log("\n" + `${lines}[ ${text} ]${lines}` + "\n");
}

function prettyPrint(tableau: string[][]) {
  let colMaxes = [];
  for (let j = 0; j < tableau[0].length; j += 1) {
    colMaxes.push(
      Math.max.apply(
        null,
        tableau.map((row) => row[j]).map((n) => n.length),
      ),
    );
  }

  tableau.forEach((row) =>
    console.log.apply(
      null,
      row.map(
        (val, j) =>
          `${new Array(colMaxes[j] - val.length + 1).join(" ")}${val}  `,
      ),
    ),
  );
}
```

src/lib.ts

```
import { arrayOf, prettyPrintWith, printHeading } from "./util";

export type SimplexResult =
  | {
      solverState: "solved";
      /** The optimal vector of the decision variables. */
      x: number[];
      /** The maximum value of the objective function. */
      z: number;
    }
  | {
      solverState: "unbounded";
    };

/**
 * @param c - A vector of coefficients of the objective function.
 * @param a - A matrix of coefficients of the constraint functions.
 * @param b - A vector of right-hand side values.
 * @param eps - Approximation accuracy (digits after the decimal point).
 */
export function maximize(
  c: number[],
  a: number[][],
  b: number[],
  eps: number = 3,
): SimplexResult | never {
  assertLengths(c, a, b);

  const xStrings = c.map((_, i) => i + 1).map((i) => `x[${i}]`);
  const sStrings = a.map((_, i) => i + 1).map((i) => `s[${i}]`);

  printHeading("* Simplex *");

  console.log(
    `Function: F(${xStrings.join(", ")}) = ${c.map((coeff, i) => `${coeff}${xStrings[i]}`).join(" + ")}`,
  );

  const rowNames = ["z", ...sStrings];
  const colNames = [...xStrings, ...sStrings, "Solution"];

  let tableau = buildTableau(c, a, b);
  const tableauCols = tableau[0].length;

  console.log("\n" + "Initial table:");
  prettyPrintWith(tableau, rowNames, colNames, eps);

  let iteration = 0;
  while (true) {
    if (checkUnbounded(tableau)) {
      printHeading("~Unbounded~");
      return { solverState: "unbounded" };
    }

    iteration += 1;
    printHeading(`Iteration ${iteration}`);

    const pivotCol = findPivotCol(tableau[0])!;

    const ratios = tableau.map((row) => row[tableauCols - 1] / row[pivotCol]);
    const pivotRow = findPivotRow(ratios)!;
```

```
  const pivotElement = tableau[pivotRow][pivotCol];

  console.log("Initially:");
  prettyPrintWith(
    tableau.map((row, i) => [...row, ratios[i]]),
    rowNames,
    [...colNames, "Ratio"],
    eps,
  );
  console.log();
  console.log(`Pivot row: ${rowNames[pivotRow]}`);
  console.log(`Pivot column: ${colNames[pivotCol]}`);
  console.log(`Pivot element: ${pivotElement.toFixed(eps)}`);

  tableau[pivotRow] = tableau[pivotRow].map((it) => it / pivotElement);

  console.log("\n" + "After dividing the pivot row:");
  prettyPrintWith(tableau, rowNames, colNames, eps);

  tableau = crissCrossed(tableau, pivotRow, pivotCol);

  if (rowNames[pivotRow] !== colNames[pivotCol]) {
    console.log(
      "\n" + `${rowNames[pivotRow]} leaves, ${colNames[pivotCol]} enters`,
    );
    rowNames[pivotRow] = colNames[pivotCol];
  }

  console.log("\n" + "After the iteration:");
  prettyPrintWith(tableau, rowNames, colNames, eps);

  if (tableau[0].every((it) => it >= 0)) {
    break;
  }
}

printHeading("Final Table");
prettyPrintWith(tableau, rowNames, colNames, eps);

const answer = tableau[0][tableauCols - 1];
const xIndexes = arrayOf(c.length, () => 0);

rowNames
  .map((rowName, i) => ({ rowName, i }))
  .slice(1)
  .forEach(({ rowName, i }) => {
    if (rowName.startsWith("x")) {
      const numPart = Number.parseInt(rowName.match(/\[(\d+)\]/)![1]);
      xIndexes[numPart - 1] = tableau[i][tableauCols - 1];
    }
  });

const result: SimplexResult = {
  solverState: "solved",
  x: xIndexes,
  z: answer,
};

console.log();
console.log(`Decision variables: ${result.x}`);
console.log(`Maximum value: ${result.z}`);
```

```
  return result;
}

function buildTableau(c: number[], a: number[][], b: number[]): number[][] {
  const tableau = arrayOf(1 + a.length, () =>
    arrayOf(c.length + a.length + 1, () => 0),
  );

  for (let i = 0; 1 + i < tableau.length; i += 1) {
    for (let j = 0; j < c.length; j += 1) {
      // Z-row
      tableau[0][j] = -1 * c[j];

      // X-es
      tableau[1 + i][j] = a[i][j];
    }

    // S-es
    tableau[1 + i][c.length + i] = 1;

    // Solution row
    tableau[1 + i][tableau[0].length - 1] = b[i];
  }

  return tableau;
}

function crissCrossed(
  tableau: number[][],
  pivotRow: number,
  pivotCol: number,
): number[][] {
  const newTableau = tableau.map((row) => row.slice());

  for (let i = 0; i < tableau.length; i += 1) {
    for (let j = 0; j < tableau[0].length; j += 1) {
      if (i != pivotRow) {
        newTableau[i][j] =
          tableau[i][j] - tableau[i][pivotCol] * tableau[pivotRow][j];
      }
    }
  }

  return newTableau;
}

function findPivotCol(zRow: number[]): number | undefined {
  return zRow
    .map((value, i) => ({ value, i }))
    .filter(({ value }) => value < 0)
    .reduce((acc: { value: number; i: number } | undefined, cur) => {
      if (acc === undefined || cur.value < acc.value) {
        return cur;
      } else {
        return acc;
      }
    }, undefined)?.i;
}

function findPivotRow(ratios: number[]): number | undefined {
  return ratios
```

```
    .map((ratio, i) => ({ ratio, i }))
    .filter(({ ratio }) => ratio > 0)
    .reduce((acc: { ratio: number; i: number } | undefined, cur) => {
      if (acc === undefined || cur.ratio < acc.ratio) {
        return cur;
      } else {
        return acc;
      }
    }, undefined)?.i;
}

function checkUnbounded(tableau: number[][]): boolean {
  return tableau[0]
    .slice(0, -1)
    .map((_, j) => j)
    .some((j) => tableau.every((row) => row[j] <= 0));
}

function assertLengths(c: number[], a: number[][], b: number[]): void | never {
  if (a.length !== b.length) {
    throw new Error(
      `numbers of constraints and right-hand sides don't match':\n` +
        ` constraints: ${a} (${a.length})` +
        ` right-hand sides: ${b} (${b.length})\n`,
    );
  }

  a.forEach((row, i) => {
    if (row.length !== c.length) {
      throw new Error(
        `numbers of coefficients don't match:\n` +
          ` function: ${c} (${c.length})\n` +
          ` constraint ${i + 1}: ${row} (${row.length})`,
      );
    }
  });
}
```

src/lib.test.ts

```
import { expect, test } from "bun:test";

import { maximize, type SimplexResult } from "./lib";

test("tut-3", () => {
  const left = maximize(
    [10, 20],
    [
      [-1, 2],
      [1, 1],
      [5, 3],
    ],
    [15, 12, 45],
  );
  const right: SimplexResult = { solverState: "solved", x: [3, 9], z: 210 };

  assertEq(left, right);
});

test("lab-3-problem-1", () => {
  const left = maximize(
    [9, 10, 16],
    [
      [18, 15, 12],
      [6, 4, 8],
      [5, 3, 3],
    ],
    [360, 192, 180],
  );
  const right: SimplexResult = { solverState: "solved", x: [0, 8, 20], z: 400 };

  assertEq(left, right);
});

test("lab-3-problem-3", () => {
  const left = maximize(
    [2, -2, 6],
    [
      [2, 1, -2],
      [1, 2, 4],
      [1, -1, 2],
    ],
    [24, 23, 10],
  );
  const right: SimplexResult = {
    solverState: "solved",
    x: [0, 3 / 4, 43 / 8],
    z: 123 / 4,
  };

  assertEq(left, right);
});

test("unbounded-1", () => {
  const left = maximize(
    [2, 1],
    [
      [1, -1],
      [2, 0],
    ],
```

```
    [10, 40],
  );
  const right: SimplexResult = { solverState: "unbounded" };

  assertEq(left, right);
});

test("unbounded-2", () => {
  const left = maximize(
    [3, 2],
    [
      [1, -1],
      [-2, 1],
    ],
    [2, -1],
  );
  const right: SimplexResult = { solverState: "unbounded" };

  assertEq(left, right);
});

function assertEq(
  left: SimplexResult,
  right: SimplexResult,
  eps: number = 3,
): void | never {
  expect(left.solverState).toBe(right.solverState);

  // HACK: this allows TypeScript to validate types.
  if (left.solverState === "unbounded" || right.solverState === "unbounded") {
    return;
  }

  expect(left.z).toBeCloseTo(right.z, eps);
  left.x.forEach((_, i) => expect(left.x[i]).toBeCloseTo(right.x[i], eps));
}
```