CSC 445/545 - SUMMER 2022
OPERATIONS RESEARCH: LINEAR PROGRAMMING
PROGRAMMING PROJECT
UNIVERSITY OF VICTORIA

Due: Friday, July 15th, 2022 at 11:59pm Victoria time (PDT). **Late assignments will not be accepted.**

This assignment will be submitted electronically through BrightSpace (as described in 'Submission Instructions' below). All code submissions must be your own work. Sending or receiving code from other students is plagiarism.

1 Overview

This assignment requires implementing an LP solver which reads a text-based representation of a linear program from standard input and outputs the result of solving the LP (including whether the LP turned out to be infeasible or unbounded). Unlike the other assignments in this course, you are permitted to use floating-point computation to solve the LP (so it is not necessary to use exact fractional arithmetic).

Section 2 describes the input format and Section 3 describes the required output format. For this assignment, **correctness is critical**, and no marks will be given for any code that cannot be rigorously validated on actual inputs (so you should prioritize basic correctness over adding extra functionality).

The project will be marked based on a combination of empirical validation (i.e. running your solver on various test LPs), human inspection of your code (which, in general, will not be performed unless it is clear that your implementation is correct) and evaluation of the accompanying documentation.

You may implement the project in C, C++, Python or Java, and must design your solution to run correctly on the Computer Science login servers at `linux.csc.uvic.ca` (which requires working with the compiler or interpreter versions installed on that system). You may be allowed to use a different language at the instructor's discretion (but only if given explicit permission in advance). Note that some aspects of the project might become more difficult depending on the language choice; this is entirely your responsibility.

2 Input Format

The input format is a simple text-based encoding of a maximization LP in standard form, which will be provided to your solver via standard input. An LP of the form

$$\begin{array}{lrcl} \text{max.} & c_1x_1 + c_2x_2 + \ldots + c_nx_n & & \\ \text{s.t.} & a_{1,1}x_1 + a_{1,2}x_2 + \ldots + a_{1,n}x_n & \leq & b_1 \\ & a_{2,1}x_1 + a_{2,2}x_2 + \ldots + a_{2,n}x_n & \leq & b_2 \\ & \vdots & & \\ & a_{k,1}x_1 + a_{k,2}x_2 + \ldots + a_{k,n}x_n & \leq & b_k \\ & x_1, x_2, \ldots, x_n & \geq & 0 \end{array}$$

will be encoded as follows:

- The first line of the input file will contain the values

  $$c_1 \quad c_2 \quad \ldots \quad c_n$$

  separated by whitespace (tabs or spaces)
- Each remaining line will encode one constraint. The line for constraint i will contain

  $$a_{i,1} \quad a_{i,2} \quad \ldots \quad a_{i,n} \quad b_i$$

  separated by whitespace.
- Blank lines (lines which are empty or contain only whitespace characters) may be present and must be completely ignored.
- The non-negativity constraints ($x_1, x_2, \ldots, x_n \geq 0$) do **not** appear in the input encoding, since they are implied by the use of standard form.

For example, the LP

$$\begin{array}{lrcl} \text{max.} & 0.5x_1 + 3x_2 & & \\ \text{s.t.} & x_1 + 4x_2 & \leq & 4 \\ & 4x_1 + 2x_2 & \leq & 4 \\ & 3x_1 + 4x_2 & \leq & 5 \\ & 5x_1 + 1.8x_2 & \leq & 5 \\ & x_1, x_2 & \geq & 0 \end{array}$$

would be encoded as

```
0.5   3
1     4     4
4     2     4
3     4     5
5     1.8   5
```

Notice that the number of variables and constraints is implied by the number of values on each line (and the number of non-empty lines). You may assume that the input format is consistent: If the first line (containing the objective coefficients) contains n values (indicating n optimization variables), each subsequent line (a constraint) will contain $n + 1$ values. You are not required to write error-handling logic to address the case of a malformed input.

If your program does not use the required input format, or uses an input source other than standard input, it may be impossible to evaluate and will receive a mark of zero.

3 Output Format

This section describes the output format of the solver. The output must be printed to standard output, and the program must generate **no other output** to standard output besides what is described here. It is permissible for the program to generate other output to standard error (and such output will be ignored during marking).

If your program does not use the required output format, it may be impossible to evaluate and will receive a mark of zero.

For clarity, examples of output in the text below are shown in a box. The expected output is **only** the text in the box.

If the input LP is infeasible (that is, if the feasible region is the empty set), the output will consist of the single line below.

Output

```
infeasible
```

If the input LP is unbounded (that is, if it is possible to increase the objective function to arbitrarily large values on feasible points), the output will consist of the single line below.

Output

```
unbounded
```

Otherwise, if the input LP is bounded and feasible, the output will consist of three lines.

- The first line will be the single word "`optimal`".
- The second line will contain the optimal objective value.
- The third line will contain an optimal assignment of each optimization variable $x_1, x_2, \ldots, x_n$, separated by whitespace (with no other separators, like commas). The optimal assignment should **only** include optimization variables, not slack variables. In cases where multiple feasible points attain the optimal solution, any feasible point is considered acceptable for this line (you do not have to ensure that your program finds a specific optimal assignment, only that the assignment provided does attain the optimal objective value).

For example, if the optimal objective value is 1.87 at the point $(x_1, x_2, x_3) = (6, 10, 1.7)$, the output will be

Output

```
optimal
1.87
6 10 1.7
```

The exact formatting of numerical values is somewhat flexible, although a minimum of seven significant digits should be printed. It is not necessary to zero-pad values to provide sufficient significant digits (so the value '1.5' can be written instead of '0001.500' or '1.500000'). You are encouraged to use the equivalent of the C-style "`%g`" format specifier to output floating point values.

4 Submission Format

We will use automation to validate your submission before human inspection. To ease this process, we expect all submissions to be in a standard format, and we **will not mark** any submission that does not comply with the standard format.

Your submission must be a `.tar.bz2` archive (using BZip2 compression) called `lp.tar.bz2` containing a directory called `lp` (all-lowercase).

To create an archive to submit, run the following command (in the parent directory of your `lp` directory):

```
$ tar -jcvf lp.tar.bz2 lp/
```

5 Basic Features

All submissions must implement an LP solver with the following basic requirements.

- If the input LP has no feasible points, the solver will report a result of "`infeasible`".
- If the input LP has any feasible points, whether or not the point $\boldsymbol{x} = \mathbf{0}$ is feasible, the solver will report a result of "`unbounded`" or "`optimal`". This means that your solver should have some mechanism for finding a feasible point in the event that the LP is not initially feasible (e.g. by solving an auxiliary problem).
- If the input LP has any feasible points and is unbounded, the solver will report a result of "`unbounded`".
- If the input LP is both bounded and feasible, the solver will report a result of "`optimal`" and output the optimal objective value, along with a variable assignment that attains the optimal objective value.
- The solver must complete successfully (in a finite number of steps) on **any** LP. This will be assessed both by empirical evaluation and code inspection, so your code and/or documentation should contain a clear explanation of how the solver avoids infinite loops. It is your responsibility to make it clear why your solver will never cycle infinitely.

Additionally, it is expected that the solver will complete in a reasonable amount of time on each LP tested during marking. Normally, five seconds or less is considered reasonable, although for some inputs this time limit may be extended (for all submissions).

6 Extra Features

Correctly implementing and documenting the basic features will result in a mark of 16/20 (80%). To receive a higher mark, you must also implement some combination of the features listed below. It is your responsibility to clearly document which extra features you implement and to show that they are correct (if your documentation does not clearly describe a particular feature, it will not be marked). If you implement an extra feature that depends on a particular type of LP, you may want to construct some sample inputs that demonstrate the feature and submit them with your code.

Marks for extra features will only be awarded if the basic features are correct. Therefore, it is your responsibility to ensure that any extra features you implement do not break the basic functionality of the solver (e.g. if you implement support for high performance pivot rules, you must ensure that cycling is still impossible).

You may implement as many extra features as you want, but the cumulative grade for all extra features will be capped at 4 marks.

6.1 List of Extra Features

1. **Primal-Dual Methods** [2 marks]: Instead of solving the auxiliary LP introduced early in the course (with the proxy variable Ω), initially infeasible LPs are solved using a two-phase primal-dual method (similar to that described starting on Slide 94 of the Duality lecture or Slide 88 of the Revised Simplex lecture).
2. **Pivoting Strategies**. Marks will be awarded for at most one of the choices below.
   (a) **Largest-Increase Rule** [1 mark]: The solver uses the largest-increase rule by default, falling back on Bland's rule in specific cases where cycling is possible. The documentation should explain how the solver identifies such cases.
   (b) **Devex** [4 marks]: The solver uses the Steepest Edge or Devex pivoting rule, falling back on Bland's rule in specific cases where cycling is possible. The documentation should explain how the solver identifies such cases.
   (c) **Alternate Cycling Avoidance** [4 marks]: The solver uses a performance oriented pivoting rule (e.g. the largest coefficient or largest increase rule) in **all** cases, but still prevents cycling (using some mechanism not covered in this course, such as the perturbation or lexicographic method).
3. **Linear Algebraic Simplex Method** [2 marks]: The solver uses the linear algebraic Simplex Algorithm instead of the dictionary-based algorithm. It is permissible for your implementation to rely on external linear algebra libraries (for tasks like matrix multiplication, solving systems of equations, and inversion), but you must provide any external libraries necessary to compile/run your code (as well as a full citation in the documentation). This can be combined with the sub-item below.
   (a) **Revised Simplex** [2 marks]: The solver implements the Revised Simplex Method instead of the basic linear algebraic version (in particular, the solver does not compute any inverse matrices). The documentation should describe the computations that are used to replace matrix inversions.
4. **Propose Your Own**: If you have an idea for a different extra feature, talk to your instructor. Only those features approved **before July 11th** will be marked.

7 Documentation

You are required to include a `README.txt` or `README.md` document with your submission which contains the following information.

- A brief guide to running your code. If your code is written in a compiled language, you should include a Makefile or shell script to handle compilation (and then explain how to use it in the documentation). Note that regardless of these instructions, your program must function correctly with an LP provided via standard input and generate its output to standard output, with no command line parameters required for basic operation. If you implement extra features, it is okay to have these require command line parameters.
- A high level description of the solver architecture (e.g. the type of Simplex implementation used, the procedure for resolving initially-infeasible problems, etc.)
- Descriptions of any novel or extra features (see Section 6). If you implement extra features, you will only receive marks if they are described in the documentation.

The `README` file does not need to be written in a formal style (it is not a technical report, just development documentation), but it should be formatted and written clearly.

If your documentation is incomplete or unclear, it may not be possible to mark parts of your project.

8 Note on Numerical Computations

Your solver's input and output will contain decimal approximations of real-valued data (see Sections 2 and 3 for details), so you will need to use a floating point representation as input data is read and output data is written. It is your decision how to represent numerical data internally inside your program, but you may discover that the choice of representation has significant development implications. Consider the two options below before making your decision.

8.1 Floating Point Representations

The most natural choice is likely to use floating point data (e.g. the `float` or `double` types in most statically-typed languages) to represent numerical values during the computation steps of your solver. If you implement the dictionary-based Simplex Method, remember that numerical anomalies (like rounding errors) can accumulate over time and result in strange behavior. If you use the dictionary-based Simplex Method, where the dictionary at each iteration is used to create the dictionary at the next iteration (such that any numerical errors are compounded over time), using floating point arithmetic requires some extra caution, especially for tasks like feasibility testing.

Consider the set of numerical operations performed by the Python code below (which would normally default to 64-bit floating point arithmetic).

```
a = 1/9.0
b = 9.0
x = a - b + b - a
```

Clearly, the value of `x` should be exactly zero, but due to numerical errors, with 64-bit (double precision) floating point arithmetic, the actual value is `-3.88578e-16`. In the context of most numerical algorithms, a "slightly-negative" value like this would be harmless (since it is so close to zero), and it is possible to implement the Simplex Method with extra logic to account for this case (i.e. by treating any value close enough to zero as being zero). However, without extra logic, the presence of such a value can break the Simplex Method (and lead to a frustrating debugging experience). For example, a comparison like `x < 0` would evaluate to `true` (since `x` is technically negative), so if `x` happens to be the value of a basic variable in a Simplex dictionary, the algorithm might conclude that the dictionary was infeasible. Similarly, if `x` appears as a coefficient in a row of a dictionary, the algorithm might make an incorrect choice of entering or leaving variable (compared to the correct case where `x` is treated as zero).

If you use floating point computation in your solver, you are responsible for adding the necessary logic to defend against accumulated numerical errors.

8.2 Rational Representations

Another option is to represent all data internally using rational numbers (that is, as a pair of integers (a, b) representing the value $\frac{a}{b}$). The easiest way to achieve rational arithmetic is to use an existing library that provides a rational type (such as the `fractions` module in the Python standard library or the `rational` library in the Boost package for C++), and convert each value in the input data to a rational number when the data is read, then convert the data back to a floating point type after the computation is complete.

9 Evaluation

Your submission must be an archive named '`lp.tar.bz2`' containing a directory `lp` (which will contain documentation, compile scripts, and your code, as described above). Your code must compile and run correctly on `linux.csc.uvic.ca`. If your code does not compile as submitted, you will receive a mark of zero.

This assignment is worth 20% of your final grade. All submissions will be marked out of 20. **Correctness is critical**: All parts of the rubric assume correctness as a baseline, and it may be impossible to assign any marks to submissions with significant correctness problems (although a single isolated bug is generally excusable).

Marks	Component
8	Empirical validation of the solver on a variety of sample linear programs.
4	Code quality (via human inspection). This will only be assessed if the empirical evaluation receives at least 4/8.
4	Documentation
4	Extra features (see Section 6)

Submission Instructions

All submissions for this assignment will be accepted electronically. You are permitted to delete and resubmit your assignment as many times as you want before the due date, but no submissions will be accepted after the due date has passed.

The final due date is Friday, July 15th, 2022 at 11:59pm PDT.

Ensure that each file you submit contains a comment with your name and student number, and that the files for each question are named correctly (as described in the question). If you do not name your files correctly, or if you do not submit them electronically, it will not be possible to mark your submission and you will receive a mark of zero.

If you have problems with the submission process, send an email to the instructor **before** the due date.

To verify that you submitted the correct file, download your submission from BrightSpace after submitting and test that it works correctly. It will be assumed that all submissions have been tested this way, and therefore that there is no reasonable chance that an incorrect file was submitted accidentally, so no exceptions will be made to the due date as a result of apparently incorrect versions being submitted.