Authors: Huy Nguyen, Poppy La
Course: CSS343

PUZZLE SOLVER DESIGN DOCUMENT

Definitions

Sudoku Puzzles

Sudoku is a logic-based, combinatorial number-placement puzzle. In classic sudoku, the objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 sub-grids that compose the grid (also called "boxes", "blocks", or "regions") contain all digits from 1 to 9. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a single solution.

Completed games are always an example of a Latin square, including an additional constraint on the contents of individual regions. For example, the same single integer may not appear twice in the same row, column, or any of the nine 3×3 subregions of the 9×9 playing board.

Genetic Programming

Genetic programming is a domain-independent method that genetically breeds a population of computer programs to solve a problem. Specifically, genetic programming iteratively transforms a population of computer programs into a new generation of programs by applying analogs of naturally occurring genetic operations. The genetic operations include crossover (sexual recombination), mutation, reproduction, gene duplication, and gene deletion.

Genetic programming typically starts with a population of randomly generated computer programs composed of the available programmatic ingredients. Genetic programming iteratively transforms a population of computer programs into a new generation of the population by applying analogs of naturally occurring genetic operations. These operations are applied to individual(s) selected from the population. The individuals are probabilistically selected to participate in the genetic operations based on their fitness (as measured by the fitness measure provided by the human user in the third preparatory step). The iterative transformation of the population is executed inside the main generational loop of the run of genetic programming. The flowchart below illustrates the execution of genetic programming.

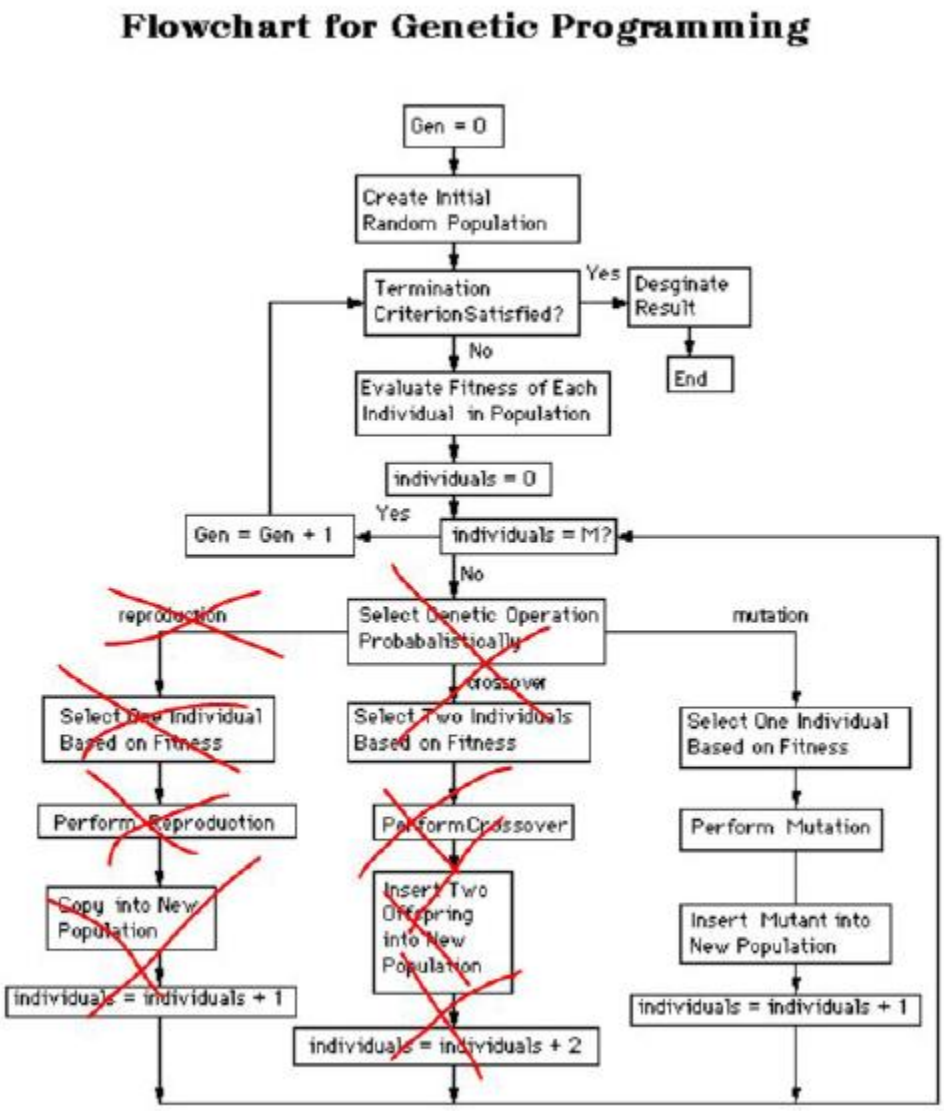
Specification

The purpose of this program is to design and implement a set of classes that define the abstractions necessary to solve puzzles of any kind using the genetic programming approach (GA).

For the purpose of this assignment, there are some core requirements that the program needs to satisfy:

1. The program must use Genetic Programing and Object-Oriented approaches (only mutation is used
for the purpose of this assignment).
2. The program must implement templates and interfaces.
3. The program must use interfaces and implement concrete classes as shown in the UML in the ***Design*** section.
4. The program must be done using pair programming methodology.
5. The program must be able to read input and produce the correct input as specified.

The executional steps of genetic programming (that is, the flowchart of genetic programming) are as follows:



- Create an initial population (generation 0) of the original Sudoku.
- Iteratively perform the following sub-steps (called a generation) on the population until the termination criterion is satisfied:
    - Execute each program in the population and ascertain its fitness (explicitly or implicitly) using the problem's fitness measure.
    - Select 10% of Sudoku from the old population with the best fitness scores (10% with the lowest fitness score) to participate in the genetic operations in the next step.
    - Create new Sudoku puzzles for the new population by applying *Mutation* : Create one new

offspring program for the new population by randomly mutating the random “0” value of the Sudoku puzzle with a specific probability. The probability should be smaller than 5% to get the accurate solution.

- After the termination criterion is satisfied, the single best Sudoku solution with the smallest fitness number in the population produced during the run (the best-so-far individual) is collected and designated as the result. If the fitness number equals 0, the result is a solution (or approximate solution) to the problem.

Input

The program should take two command line arguments: the size of the population and the maximum number of generations. It should then read the sudoku puzzle from **cin** and generate the first generation (i.e., randomly fill in the squares that are not fixed to produce a population of the specified size with that many different Puzzles). It should then begin the GA iterations.

Output

At the end, it should output the best puzzle found, with its fitness value. Along with the fitness value, the best puzzle found has to be “human friendly” output, that is, it should look like the following:

```
+-------+-------+-------+
| 4 2 3 | 7 5 1 | 9 6 8 |
| 7 5 9 | 6 8 3 | 1 2 4 |
| 1 6 8 | 2 4 9 | 3 5 7 |
+-------+-------+-------+
| 9 4 5 | 3 6 2 | 8 7 1 |
| 8 7 2 | 9 1 5 | 4 3 6 |
| 3 1 6 | 4 7 8 | 2 9 5 |
+-------+-------+-------+
| 5 3 7 | 1 9 4 | 6 8 2 |
| 6 9 1 | 8 2 7 | 5 4 3 |
| 2 8 4 | 5 3 6 | 0 1 9 |
+-------+-------+-------+
```

Error Handling/Special Cases:

1. The program shall take only 2 command line arguments.
2. The program shall be able to handle file input.
3. The program shall be able to handle non-numeric characters/invalid command line arguments/invalid file input.
4. The program shall execute without crashing.

Design

This program consists of 5 interfaces (Puzzle, PuzzleFactory, Population, Fitness, Reproduction) and 5 subclasses (Sudoku class, SudokuFactory class, SudokuPopulation class, SudokuFitness class, SudokuOffSpring class) that implement the interfaces.

Puzzle <<interface>

Public:

- abstract operator<< (ostream&, Puzzle &): ostream &
    - The output operator overload for puzzle abstract. The operator prints out the Puzzle object.
- abstract operator>> (istream&, Puzzle &): istream &
    - The input operator overload for puzzle abstract, it takes the Puzzle input using cin to create the Puzzle object.
- abstract setFit(int): void
    - Set the Puzzle's fitness score
- abstract getFit(): int:
    - Return the Puzzle's fitness score

Protected:

- fit_score:
    - Fitness score of Sudoku.
- fitness_: Fitness
    - The Puzzle's Fitness object that stores fitness score.

Sudoku class

Public:

- operator<< (ostream&, Sudoku &): ostream &
    - This operator prints out a Sudoku puzzle to the console using the "user friendly" format specified in the Specification section.
- operator>> (istream&, string&): istream &
    - This operator takes in a sequence of strings read from a text format that can be used to construct a Sudoku puzzle
- setFit(int): void
    - Set the Sudoku's fitness score
- getFit(): int:
    - Return the Sudoku's fitness score

Private:

- inputHelper(string): void
    - This helper method will help the input stream above read in a sequence of strings of a text format containing 81 digits (extract the non-digits character if present) to construct a Sudoku puzzle.
- outputHelper(): void
    - This helper method will help the output stream above print out the "user friendly" Sudoku format (the expected format).
- grid_: vector<vector<int>>
    - This vector is a 2-D vector that represents a 9x9 Sudoku grid.
- bestFit(): bool
    - This boolean value shows if the Sudoku puzzle object has the best fit (smallest fitness value).

PuzzleFactory <<interface>>

Public:

- abstract createPuzzle(Repuduction &): Puzzle
    - The method uses a Reproduction object to create a new Puzzle object.

SudokuFactory class

Public:

- createPuzzle(SudokuOffSpring&): Sudoku
    - The method uses a SudokuOffSpring object to create a new Sudoku puzzle object of the new generation.

Population <<interface>>

Public:

- abstract createOriginalPopulation(Puzzle &): void
    - Take a Puzzle object in parameter and produce an original population of Puzzles equal to the population size.
- abstract createNewGen(): void
    - Method will use the old generation that's culled (10% most fit remaining of the old generation) to create a new generation. Produce a new generation of Puzzle with n individuals for every single individual in the old generation (after culled) to keep the contents size of population. The new generation of Puzzles will be stored in a new_population_ variable.
- abstract bestFit(): int
    - Return the best (smallest) fitness value of the Puzzle in the current generation.
- abstract bestInvidual(): Puzzle
    - Return the Puzzle that has the best fitness number (smallest fitness number) in generation.
- abstract cull(): void
    - Eliminate 90% Puzzle with least fit number (largest fitness number) in population, keep the rest 10% Puzzle in old_population_ variable for reproduction.
- abstract getSize(): int
    - Return the Puzzle's population size.

Protected:

- new_population_: vector<Puzzle>
    - Stores the new generation contain old and new generations.
- old_population_: vector<Puzzle>
    - Stores the old generation of Puzzle population.

Private:

- size: int
    - Population's size.

SudokuPopulation

Public:

- createOriginalPopulation(Sudoku &): void
    - Take a Sudoku object in parameter and produce an original population of Puzzles equal to

the population size.

- createNewGen(): void
  - Method will use the old generation that's culled (10% most fit remaining of the old generation) to create a new generation. The new generation of Sudoku puzzles with n individuals for every single individual in the old generation (after culled) to keep the content size of the population.
- bestFit(): int
  - Return the best (smallest) fitness value of the Sudoku puzzle in the current generation.
- bestInvidual(): Sudoku
  - Return the Sudoku puzzle that has the best fitness number in generation.
- cull(): void
  - Eliminate 90% Sudoku puzzle with least fit number (largest numbers) in population, keep the rest 10% Sudoku puzzle, and store in old_population_ variable for reproduction.
- getSize(): int
  - Return the Sudoku population size.

Fitness <<interface>>

Public:

- abstract howFit(Puzzle &): int
  - Return the Puzzle's fitness number.

SudokuFitness class

Public:

- howFit(Sudoku &): int
  - Return the Sudoku's fitness number. The fit number equals the total duplicate entries in the Sudoku's puzzle.

Private:

- scanRow(Sudoku &): int
  - Return the number of duplicate entries there in each row.
- scanColumn(Sudoku &): int
  - Return the number of duplicate entries there in each column.
- scanBlock(Sudoku &) int
  - Return the number of duplicate entries there in each block.

Reproduction <<interface>>

Public:

- abstract makeOffSpring(Puzzle &): Reproduction
  - Create a new Reproduction object from a Puzzle object, the new Reproduction object has Puzzle's "0" values mutated.

Private:

- mutate(Puzzle &, int): bool
  - Change "0" values in the Puzzle with a random number in the range of (1, 9), using the probability passed in integer.

SudokuOffSpring class

Public:

- makeOffSpring(Sudoku &): SudokuOffSpring
    - Create a new Reproduction object from a Sudoku object, the new SudokuOffSpring object has Sudoku's "0" values mutated.

Private:

- mutate(Sudoku &, int): bool
    - Change "0" values in the Sudoku puzzle with random numbers in the range of (1, 9).

GeneticAlgorithm class

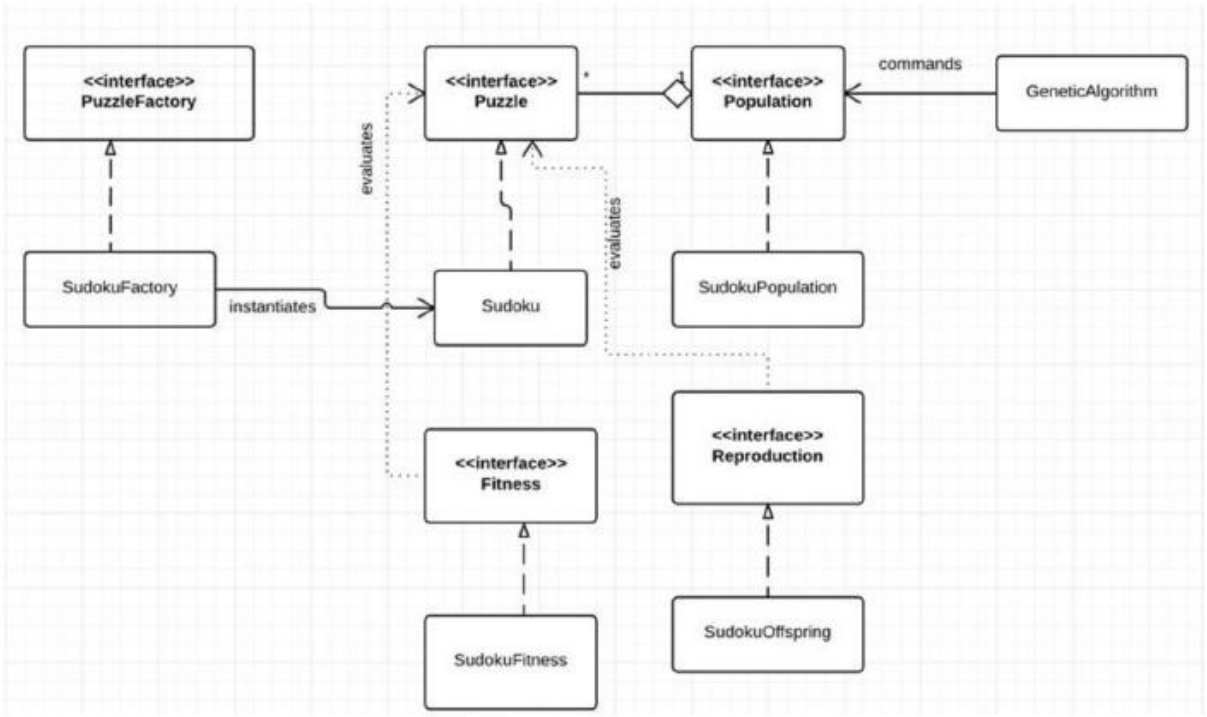
Public:

- processSudoku(int, int): void
    - Takes value from the command line argument and stores in population_size and generation_number variable. Give command for SudokuPopulation object to create a population of Sudoku and find the most fit version of Sudoku's solution.
- printSudoku(): void
    - Print out the most fit Sudoku in desired form.
- getResultFitness(Sudoku &): int
    - Return the fitness number of the most fit Sudoku in the population.

Private:

- original_sudoku: Sudoku
    - The input Sudoku created from cin.
- result_sudoku: Sudoku
    - The most fit Sudoku.
- population: SudokuPopulation
    - All the input Sudoku solutions.
- population_size: int
    - Inputted size of population.
- generation_number: int
    - Input maximum number of generations.

UML & Class Definitions



Implementation and Test Plan

The general strategy for implementation and testing is Incremental Development and Unit Testing, respectively. The order of implementation and testing is as followed:

1. Starting with the abstract class **Puzzle** which will be defined, and the concrete class **SudokuPuzzle** will implement the methods from **Puzzle** and those methods will be tested in the following order:

1.1 Sudoku::operator>> will be tested by reading a 81 character string (representing a sudoku puzzle) and printed out using the <<operator, which will be implemented and tested immediately after this. Note that **inputHelper(string)** will also be implemented at the same time with this operator since they work side by side.

Input: dummy string containing 81 numerical characters from (0 to 9 inclusive). Another test could be done by inputting 81 characters that include numeric and non- numeric characters and the operator should be able to handle both cases.

Expected output: the operator should be able to handle the two cases above, and able to correctly construct a sudoku puzzle given the string input.

1.2 Sudoku::operator<< will be implemented and tested after **operator>>** has been implemented. These operators have to be implemented and tested side by side to know which these operators are working correctly. Note operator<< also uses an **outputHelper()** method to help print out the sudoku puzzle to the console.

Input: the input will be created by making a Sudoku object, using the operator to print out the “user

friendly" format of a sudoku puzzle.

Expected output: the sudoku puzzle should have the format specified in the *Specification* section.

1.3 Sudoku::getFit() will return the fitness score of a Sudoku puzzle object by returning the fit_score_ integer.

Input: none.

Expected output: any non negative integer value ranging from 0 to n inclusive.

2. The next abstract and subclass to be implemented and tested are **Fitness** interface and **SudokuFitness** class:

2.1 SudokuFitness::howFit(Sudoku &) this method will return an integer value that shows how fit a Sudoku puzzle is. A value of 0 is the most fit, the larger the value, the less fit a Sudoku puzzle is. This function will call **scanRow(Sudoku &)** , **scanColumn(Sudoku &)** , **scanBlock(Sudoku &)** , to calculate the final fitness score for a Sudoku puzzle object.

Input: given a completed Sudoku puzzle object (this does not necessarily mean that the numbers are filled in correctly) determine the fitness_score_, by calculating how "correct" the Sudoku puzzle object is (if it follows the sudoku puzzle rules).

Expected output: a positive integer value between 0 (best fit) to n.

2.2 SudokuFitness::scanRow(Sudoku &) this method will scan all the rows in the 9x9 sudoku grid to find duplicates. Any pair of numbers that are duplicates will add 1 to the total count (increasing the overall fitness score). More duplicates equal a larger fitness score, and less to none duplicates will result in a low or 0 fitness score (with 0 being the fittest).

Input: given a completed Sudoku puzzle object (this does not necessarily mean that the numbers are filled in correctly) determine the total fitness score for all the rows, by calculating how "correct" the Sudoku puzzle object is (if it follows the sudoku puzzle rules) in the row major order.

Expected output: returns an integer, 0 if there are no duplicates, n score if there are duplicates.

2.3 SudokuFitness::howColumn(Sudoku &) this method will scan all the columns in the 9x9 sudoku grid to find duplicates. Any pair of numbers that are duplicates will add 1 to the total count (increasing the overall fitness score). More duplicates equals a larger fitness score, and less to none duplicates will result in a low or 0 fitness score (with 0 being the fittest).

Input: given a completed Sudoku puzzle object (this does not necessarily mean that the numbers are filled in correctly) determine the total fitness score for all the columns, by calculating how "correct" the Sudoku puzzle object is (if it follows the sudoku puzzle rules) in the column major order.

Expected output: returns an integer, 0 if there are no duplicates, n score if there are duplicates.

2.4 SudokuFitness::scanBlock(Sudoku &) this method will process each individual 3x3 block in the sudoku grid to find duplicates. The goal is that the block has to contain all unique values (no duplicates). More duplicates equals a larger fitness score, and less to none duplicates will result in a low or 0 fitness score (with 0 being the fittest).

Input: given a completed Sudoku puzzle object (this does not necessarily mean that the numbers are filled in correctly) determine the total fitness score for all the 3x3 blocks, by calculating how "correct" the Sudoku puzzle object is (if it follows the sudoku puzzle rules) in every block.

Expected output: returns an integer, 0 if there are no duplicates, n score if there are duplicates.

3. The next abstract and subclass to be implemented and tested are **PuzzleFactory** interface and **SudokuFatory** class:

3.1 SudokuFactory::createPuzzle (SudokuOffString&) this method will create a new Sudoku puzzle object using the passed in SudokuOffString object. Basically this method works as a converter that converts a SudokuOffString object into a Sudoku object.

Input: given a SudokuOffString object that has been created and initialized correctly, the goal is to convert SudokuOffSpring object to a Sudoku object.

Expected output: a Sudoku puzzle object is returned by the method.

4. The next abstract and subclass are **Reproduction** interface and **SudokuOffSpring** class:

4.1 SudokuOffSpring::makeOffString (Sudoku &) this will call the **mutate(Sudoku &)** method and return a SudokuOffString object (the newly created and mutated Sudoku object). Note that the **mutate(Sudoku &)** method has to be implemented at the same time as this method since these two methods work side by side together.

Input: given a Sudoku object that has been created and initialized correctly, the goal is to turn the mutated Sudoku object into SudokuOffString object as a result.

Expected output: a mutated SudokuOffString object.

4.2 SudokuFactory::createPuzzle (Sudoku &, int) This method will mutate the Sudoku object with the possibility rate by parameter. It randomly swaps the zeros in the original Sudoku object into random non-zero digits in the range of 1-9 inclusive using the probability rate.

Input: given a Sudoku object that has been created and initialized correctly, the goal is to mutate the Sudoku object that was passed in.

Expected output: a Sudoku puzzle object that has been mutated. Returning true if the mutation process is successful, false if it fails.

5. The next abstract and subclass are **Population** interface and **SudokuPopulation** class. This class methods can be called and performed in main.

5.1 SudokuPopulation::createOriginalPopulation(Sudoku &) This method takes a Sudoku object in parameter and creates the population of Sudoku, then stores it in new_generation_ variable. The population can be printed using cout.

Input: Given Sudoku object.

Expected output : a list of lines, each line contains all 81 digit characters of each Sudoku puzzle in

population.

5.2 SudokuPopulation::createNewGen(): After calling this method, the population will use the old Sudoku generation that's culled (10% most fit of old generation) to create a new generation of Sudoku. Print out the original population before calling the method, then call the method and print the new population using cout to check the differences.

Input: The method will use the old_generation_ variable to create a new generation.

Expected output : a list of lines, each line contains all 81 digit characters of the new generation.

5.3 SudokuPopulation::bestFit() . Call the method to return the smallest fitness score in the population/fitness score of the best Sudoku solution. The result can be printed using cout.

Input: None

Expected output : a positive integer number.

5.4 SudokuPopulation::bestInvidual() . Call the method to return the best Sudoku solution with the smallest fitness number in the population . The result Sudoku can be checked using cout.

Input: None.

Expected output : The Sudoku that has the best solution, with the Sudoku desired format.

5.5 SudokuPopulation::cull() . The method reduced the new_generation by 90% (eliminated the high fitness score) and stored it in old_generation_ variable. Print the unculled new_generation and the updated old_generation variables to compare, the variables can be printed using cout. The 2 variable sizes can also be used to compare using cout.

Input: None. The method uses the new_generation.

Expected output : a list of lines, each line contains all 81 digit characters of the new generation. The updated old_generation lines should equal 10% of new_generation.

5.6 SudokuPopulation::getSize() . Method return the size of population

Input: None

Expected output : a positive integer number.

6. GeneticAlgorithm is the final class that will be implemented and tested since it will need to call the classes above. This class will also act as a driver that will run the program.

6.1 GeneticAlgorithm::processSudoku(int, int) This method will take 2 values from the command line (the population size and max generation number) and find the solution (the most fit Sudoku). The result Sudoku can be printed using cout or printSudoku().

Input: given two integer values.

Expected output: Result Sudoku with the smallest fitness score.

6.2 GeneticAlgorithm::printSudoku() this method will print out the best fit Sudoku (the solution) to the screen in a “user friendly” format, see *Design* section for how the format will look like.

Input: none, method will print out the result_sudoku variable.

Expected output: a Sudoku object with the smallest fitness score.

6.2 GeneticAlgorithm::getResultFitness() this method will return the fitness score for the best fit Sudoku (solution)

Input: none

Expected output: a small positive integer value ranging from 0 to n.