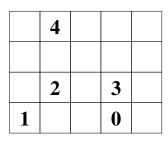
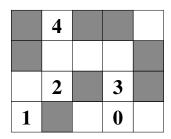
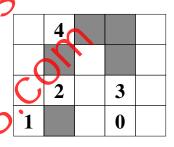
CSCI-1200 Data Structures — Fall 2024 Homework 6 — Recursive Kurotto Solver

In this homework we will develop a solver for a paper-and-pencil puzzle game named *Kurotto*. The input to Kurotto is a 2D rectangular grid where all the cells are initially *water*. Some of the cells in the grid are labeled with a non-negative integer. To solve the puzzle we will shade some of the un-numbered cells in the grid. Each cell we shade becomes *land*. Land cells that share a horizontal or vertical edge are joined to form islands. Cells that have numbers cannot be shaded, they will remain water. A Kurotto puzzle is *solved* if each numbered cell is touching (along horizontal or vertical edges) islands whose area sums exactly to the number in that cell. For our homework, we will work with inputs that may have no solutions, exactly one solution, or multiple solutions. The small example below has 3 different solutions! One of the solutions has 5 islands and the other two solutions have 3 islands.









You are encouraged to play through example puzzles so you understand the rules. First focus on verifying that a proposed solution of shaded cells is indeed a complete and valid solution. Then think about how you create a solution by filling in shaded cells yourself. Even though the rules are very simple, this puzzle is surprisingly difficult! As you play, think about how you as a human solve these puzzles. As the puzzles grow larger how and why does it become more challenging? Can you generalize your method of finding a solution so you can teach a friend how to play and ultimately "teach" the computer to play? Here is a link with more information about the game and some medium and larger puzzles:

https://www.gmpurales.com/blog/kurotto-rules-and-info/

IMPORTANT NOTE: You may not search for, study, or use any code or techniques related to existing solvers for this puzzle game. Please carefully read the entire assignment and study the examples before beginning your implementation.

Command Line Arguments & Input/Output

Your program will accept one or more command line arguments. The required first argument is the name of a puzzle board file. The input file for the puzzle in the diagrams above is shown on the right. The board is contained in a simple ASCII art frame. The space character represents water cells. The '#' character represents land. Digits '0' through C9' represent the numbered cells 0-9, and then we switch to letters: 'a'=10, 'b'=11, etc.



The butput of your program will print to std::cout. We can use file redirection (> output4.txt) on the command line to write scoop up the output and save it to a file. See the provided output files for sample output from each of these command lines. If no other command line arguments are specified, your program should search for and print a single valid solution to the puzzle (if any exists). If there are multiple solutions, your code can print any solution. If the optional --find_all_solutions command line argument is specified, your program should output all solutions, they can be printed in any order.

- ./kurotto_solver.out input4.txt > output4.txt
- ./kurotto_solver.out input4.txt --find_all_solutions > output4_all.txt

```
However, before you tackle finding one or all solutions, it is important that
                              you can identify all of the islands in a proposed/possible solution. Since the
|#4## |
                              input and output file formats match, we can actually feed a sample output
      #|
                              file as input to the program and then compute and print information about
2#3#|
                              the solution board. The optional command line argument --no_solver will
11# 0 |
                              skip the search to find solutions and instead just analyze the shaded and
                              cells already on the board.
ISLAND COUNT = 5
ISLAND:
           (0,0) (0,1)
                              When the optional command line argument --print_islands is specified,
ISLAND:
           (2,0) (3,0)
                              your program should collect the land cells within the diagram, group them
ISLAND:
           (4,1) (4,2)
                              into islands, and then print this information. An example of this output is
                              shown on the left. Note that you can print the islands in any order, and
ISLAND:
           (2,2)
                              likewise the col, row grid positions within each island can be printed in any
ISLAND:
           (1,3)
                              order.
  ./kurotto_solver.out output4.txt --no_solver --print_islands > output4_print_islands.txt
  ./kurotto_solver.out output4.txt --no_solver --print_check_solution
                                                         > output4_print_check_solution.txt
  ./kurotto_solver.out input4_buggy.txt --no_solver --print_check_solution
                                                   > output4 buggy_print
                                                                         check_solution.txt
+----+
                                                correctly
                                   you have
1#4##
                            identified the islands on ke
      #|
                            board, you can proceed to
2#3#1
                            verify that each numbered cell
|1# 0 |
                            is satisfied. When the optional
                                              argument
+----+
                            command Nine
                            --print_theck_solution (is
Location (1,0) = 4
                                                            Location (1,0) = 4
                            provided the program should
Location (1,2) = 2
                                                            Location (1,2) should be 2 is 1
                            output information about
Location (3,2) = 3
                                                            Location (3,2) should be 3 is 2
                            each numbered location on
Location (0,3) = 1
                                                            Location (0,3) = 1
```

And finally, here is an example command line using the final optional command line argument --print_solution_count Be sure to study the sample input and output files on the website.

the board and the total area

of adjacent islands.

```
./kurotto.solver.out input4.1xt --find_all_solutions --print_islands

--print_check_solution --print_solution_count > output4_all_print.txt
```

We provide basic code to handle the required and optional command line arguments and a function to read the puzzle board. You may modify the provided code as necessary to complete the assignment.

Location (3,3) = 0

Puzzle is NOT SOLVED

Additional Requirements: Recursion & Big O Notation

Location (3.3) = 0

Puzzle is solved

You must use recursion in a non-trivial way in your solution – e.g., for detecting islands, for shading land blocks, and for enumerating all possible solutions to the puzzles. As always, we recommend you work on this program in logical steps. *IMPORTANT NOTE: This problem is computationally expensive, even for medium-sized puzzles! Be sure to create your own simple test cases as you debug your program.*

As always, use good coding style when you design and implement your program. You are encouraged but not required to write one or more new classes for this assignment. Make thoughtful decisions about how to efficiently use the STL classes we have discussed in lecture and lab.

Once you have finished your implementation, analyze the performance of your program using Big O Notation. What important variables control the complexity of a particular problem? The dimensions of the board (w and h)? How many cells are labeled with numbers (n)? The average or maximum value of cell number labels (v)? The area of the largest island in the solution (a)? In your README.txt file write a concise paragraph (< 200 words) justifying your answer. Include a table summarizing the running time and number of solutions found by your program on each of the provided examples and your own new test cases for the different command line arguments. Note: It's ok if your program can't solve the medium or big ouzgles in a reasonable amount of time.

You must do this assignment on your own, as described in the "Collaboration Policy & Academic Integrity" handout. If you did discuss this assignment, problem solving techniques, or error messages, etc. with anyone, please list their names in your README.txt file.

Homework 6 Kurotto Solver Contest Rules

- All students are required to submit their program to the contest. Extra credit will be awarded for programs that have a strong performance in the contest.
- Members of the teaching staff will also be submitting to the cortest... who will win???
- Contest submissions are a separate Submitty gradeable. Contest submissions are due Tuesday, October 29th at 11:59pm. You may not use late days for the contest. (The regular homework deadline is Thursday, Oct 24th at 11:59pm and late days are allowed for the regular homework submissions.)
- You may submit the same code for both the regular homework submission and the contest. Or you may make a small or significant change for the contest.
- Contest submissions do not need to use recursion.
- We will compile your code with optimizations enabled (g++ -03 *.cpp) and run all submitted entries on the homework server.
- Programs must be single-threaded and single-process.
- We will run your program by redirecting std:.cout to a file and measure performance with the UNIX time command. For example:

```
time ./kurotty solver.out input4.txt --find_all_solutions > output4_all.txt
```

- You may want to use a $C\mathcal{I}+$ code profiler to measure the efficiency of your program and identify the portions of your code that consume most of the running time. A profiler can confirm your suspicions about what is slow, incover the efficient protions of the code.
- We will be testing with and without the optional command line arguments --no_solver, --print_islands --print_check_solution, --print_solution_count, and --find_all_solutions and will highlight the most correct and the fastest programs.
- You may submit one or two new Kurotto puzzle board file with your regular homework submission for possible inclusion in the contest. Name these test files: puzzle_smithj_1.txt and puzzle_smithj_2.txt (where smithj is your RCS username). Extra credit will be awarded for interesting test cases that are used in the contest. Caution: Don't make the test cases so difficult that your own program cannot solve them in a reasonable amount of time!
- In your README_contest.txt file, describe the optimizations you implemented for the contest and summarize the performance of your program on all test cases.