

**Lab report**

|  |  |
| --- | --- |
| **Course**: | Class Libraries and Data Structures |
| **Semester**: | 1st semester of the academic year **2020-2021** |
| **Major**: | Software Engineering |
| **Class**: | 2019 |
| **Student Name**: | Fu Ruoxuan |
| **Student ID:** | 222019321062060 |
| **Teacher:** | ZHAO, Hengjun (赵恒军) |

**School of Computer and Information Science**

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Recursion and Backtrack | | |
| Date | Dec 10，2020 | Type | □Confirmatory  √ Design  □Comprehensive |
| 1. **Objective & Requirements**    1. Familiarize you with the basic idea of recursion    2. Understand the design principles of recursive functions and can design your own recursive functions    3. Understand the principle of backtracking and can design a backtracking algorithm to solve a practical problem | | | |
| 1. **Experimental environment (**platform and software**)**   Windows 7 (or higher versions) + Visual Studio 2010 (or higher versions) | | | |
| 1. **Experimental content and design** (Main Content, Procedure, Codes and Results)   Task 1:  Use recursion to output the n-th Fibonacci number.  Task 2:  A chessboard has eight rows and eight columns. In the game of chess, the queen is the most powerful piece: she can attack any piece in her row, any piece in her column, and any piece in either of her diagonals. Develop and validate a program to place eight queens on a chessboard in such a way that no queen is under attack from any other queen. (Hint: use recursion and backtrack)  Task 1 code:   1. #include <iostream> 3. **using** **namespace** std; 5. **int** Fibonacci(**int** n) 6. { 7. **return** n < 2 ? n : Fibonacci(n - 1) + Fibonacci(n - 2); 8. } 10. **int** main() 11. { 12. **int** n = 0; 13. cin >> n; 14. **int** result = Fibonacci(n); 15. cout << result; 16. }   Task 1 result: | | | |

|  |  |  |
| --- | --- | --- |
| Task 2 code:   1. #include <iostream> 2. #include <cmath> 3. **using** **namespace** std; 4. **static** **const** **int** len = 8; 5. **int** columns[len]; 6. **void** init() 7. { 8. **for** (**int** i = 0; i < len; i++) 9. { 10. columns[i] = -1; 11. } 12. } 13. **void** printMaze() 14. { 15. cout<<"------------------------"<<endl; 16. **for** (**int** row = 0; row < len; row++) 17. { 18. **for** (**int** col = 0; col < len; col++) 19. { 20. **if** (columns[row] == col) 21. { 22. cout<<("|Q|"); 23. } 24. **else** 25. { 26. cout<<("| |"); 27. } 28. } 29. cout << endl; 30. } 31. cout<<"------------------------"<<endl; 32. }  35. **bool** isLegal(**int** queen) 36. { 37. **for** (**int** i = 0; i < queen; i++) 38. **if** (columns[i] == columns[queen] || abs(i - queen) == abs(columns[i] - columns[queen])) 39. **return** **false**; 40. **return** **true**; 41. }   45. **void** extendPath(**int** row, **int** column) 46. { 47. columns[row] = column; 48. } 50. **void** goBack(**int** row) 51. { 52. columns[row] = -1; 53. } 55. **bool** success(**int** row, **int** column) 56. { 57. **return** (row == 7 && column == 7); 58. } 60. **bool** backTrack(**int** row) 61. { 62. **for** (**int** i = 0; i < len; i++) 63. { 64. extendPath(row, i); 66. **if** (success(row, i)) //reach goal 67. **return** **true**; 68. **else** //not goal 69. { 71. **if** (isLegal(row) && backTrack(row + 1)) 72. { 73. **return** **true**; 74. } 75. **else** //cannot reach goal from current (row, column) 76. { 77. goBack(row); //lead to end position 78. **if** (i == len-1) 79. { 80. **return** **false**; 81. } 82. } 83. } 84. } 86. } 88. **int** main() 89. { 90. init(); 91. **if** (backTrack(0)) 92. printMaze(); 93. **else** 94. cout << "No path exists!\n"; 95. **return** 0; 96. }   Task2 Result     1. **Result analysis and discussion**（Analysis of experimental results and summing up the harvest and the existing problems）   **Analysis of experimental results**  Task1 can correctly use recursion to generate fibonacci results  Task2 also uses a recursive approach to backtracking. The experiment separates out each module: print, determine if a piece can be placed, place the piece, return the step, determine success, and backtrack the body. The board is represented using a one-dimensional array. The index is used to represent the rows, and the elements pointed to by the index represent the columns. This representation eliminates the need to determine that rows are legal and reduces the complexity of the code Backtrack can effectively implement backtracking through functions whose return value is bool. When the return values of all functions called by backtrack are true, then the eight queens problem is solved; if one is false, then return to the upper function to continue trying the next position; if all are false, then there is no suitable solution  **Harvest**  The programming technique by which a program calls itself is called recursion. A procedure or function has a way to call itself directly or indirectly in its definition or description. It usually transforms a large and complex problem layer by layer into a smaller scale problem similar to the original problem to solve, and the recursive strategy requires only a small number of procedures to describe the many iterations of computation needed to solve the problem, greatly reducing the amount of code in the program. In general, recursion requires a boundary condition, a recursive forward segment, and a recursive return segment. When the boundary condition is not satisfied, recursion advances; when the boundary condition is satisfied, recursion returns.  The backtracking algorithm is actually an enumeration-like search attempt process, which focuses on finding the solution to the problem during the search attempt, and then "backtracking" back to try another path when the solution conditions are no longer met. The backtracking method is a selective search method, which searches forward according to the selected conditions to reach the goal. However, when the search reaches a certain step and the original choice is found to be not optimal or does not reach the goal, the search is retraced to a new choice.  max is a reserved word in C++  **Existing problems**  The code that determines the legality of a drop combines column legality with diagonal legality, without separating the two functions | | |
| Comments & Evaluation | Content & Design (A-E) |  |
| Procedure & Codes (A-E) |  |
| Results (A-E) |  |
| Analysis & Discussion (A-E) |  |
| Score (A-E):  Feedback comments: | |