1. Say we have a triangle ABC and want to know if point P is inside of this triangle. What we can do is to express AP as the linear combination of AB and AC

Solve for a and b

If a,b>0 and a+b<1 then the point P is inside of the triangle ABC.

|  |
| --- |
| 1. **struct** Double3 2. { 3. **double** x = 0; 4. **double** y = 0; 5. **double** z = 0; 6. **public**: 7. Double3(); 8. Double3(**double** a, **double** b, **double** c); 9. }; 10. Double3 add(Double3 A, Double3 B); 11. Double3 subtra(Double3 A, Double3 B); 12. Double3 cross(Double3 A, Double3 B); 13. **double** dot(Double3 A, Double3 B); 14. Double3::Double3() 15. {} 16. Double3::Double3(**double** a, **double** b, **double** c) 17. { 18. x = a; 19. y = b; 20. z = c; 21. } 22. Double3 add(Double3 A, Double3 B) 23. { 24. Double3 temp; 25. temp.x = A.x + B.x; 26. temp.y = A.y + B.y; 27. temp.z = A.z + B.z; 28. **return** temp; 29. } 30. Double3 subtra(Double3 A, Double3 B) 31. { 32. Double3 temp; 33. temp.x = A.x - B.x; 34. temp.y = A.y - B.y; 35. temp.z = A.z - B.z; 36. **return** temp; 37. } 38. Double3 cross(Double3 A, Double3 B) 39. { 40. Double3 temp; 41. temp.x = A.y\*B.z - A.z\*B.y; 42. temp.y = A.z\*B.x - A.x\*B.z; 43. temp.z = A.x\*B.y - A.y\*B.x; 44. **return** temp; 45. } 46. **double** dot(Double3 A, Double3 B) 47. { 48. **return** A.x\*B.x + A.y\*B.y + A.z\*B.z; 49. } 50. **bool** inTri(Double3 A, Double3 B, Double3 C, Double3 P) 51. { 52. Double3 AB = subtra(B, A); 53. Double3 AC = subtra(C, A); 54. Double3 AP = subtra(P, A); 56. **double** norm = dot(AB, AB)\*dot(AC, AC) - dot(AB, AC)\*dot(AB, AC); 57. **double** a = (dot(AP, AB)\*dot(AC, AC) - dot(AP, AC)\*dot(AC, AB)) / norm; 58. **double** b = (dot(AP, AC)\*dot(AB, AB) - dot(AP, AB)\*dot(AC, AB)) / norm; 60. **return** (a > 0 && b > 0 && a + b < 1); 61. } 62. **int** main() 63. { 64. Double3 A = { 0, 0, 0 }; 65. Double3 B = { 1, 0, 0 }; 66. Double3 C = { 0, 1, 0 }; 67. Double3 P = { 0.25, 0.25, 0 }; 68. std::cout << inTri(A, B, C, P)<<std::endl; 69. **char** wait; 70. std::cin >> wait; 71. **return** 1; 72. } |

1. This can be done by checking the pair of chars that located at opposite position.

|  |
| --- |
| 1. #include <iostream> 2. #include <string> 4. **using** **namespace** std; 6. **bool** checkPal(string \*s) 7. { 8. **int** i=0; 9. **while**(i<s->length()/2) 10. { 11. **if** ((\*s)[i]==(\*s)[s->length()-1-i]) 12. { 13. i++; 14. } 15. **else** 16. { 17. **return** **false**; 18. } 19. } 20. **return** **true**; 21. } 23. **void** main() 24. { 25. string s("112211"); 26. cout<<checkPal(&s)<<endl; 27. } |

1. DP solution O(N^2) time complexity, O(N) space complexity.

Basic idea is to use an array of same size as the data to store the length of the longest decreasing array end with arr[i]. And that can be done by finding the max(table[i],table[j]+1) for all arr[j]>arr[i],j<i.

|  |
| --- |
| 1. #include <iostream> 2. #include <math.h> 3. #include <vector> 5. **using** **namespace** std; 7. **void** findLDS(vector<**double**> \*arr, vector<**int**>\* table, vector<**double**>\* result) 8. { 9. **int** i, j; 10. **int** max = 0; 12. **for** (i = 0; i < (\*arr).size(); i++) 13. { 14. **for** (j = 0; j < i; j++) 15. { 16. **if** ((\*arr)[j] > (\*arr)[i]) 17. { 18. (\*table)[i] = (\*table)[i]>(\*table)[j] + 1 ? (\*table)[i] : (\*table)[j] + 1; 19. } 20. } 21. max = max > (\*table)[i] ? max : (\*table)[i]; 22. } 24. **for** (i = (\*table).size() - 1; i >=0; i--) 25. { 26. **if** ((\*table)[i] == max) 27. { 28. (\*result).push\_back((\*arr)[(\*arr).size() - i - 1]); 29. max--; 30. } 31. } 32. } 33. **int** main() 34. { 35. vector<**double**> arr={5,4,4,3,3,2,2,1,1}; 36. vector<**int**> table(arr.size(), 1 ); 37. vector<**double**> result; 38. **int** i; 39. findLDS(&arr, &table, &result); 40. **for** (i = 0; i < table.size(); i++) 41. { 42. cout << table[i] << ','; 43. } 44. cout << endl; 45. **for** (i = 0; i < result.size(); i++) 46. { 47. cout << result[i] << ','; 48. } 49. **return** 1; 50. } |

1. Backtracking method

Reference: <http://www.geeksforgeeks.org/backtracking-set-7-suduku/>

|  |
| --- |
| 1. // A Backtracking program  in C++ to solve Sudoku problem 2. #include <stdio.h> 4. // UNASSIGNED is used for empty cells in sudoku grid 5. #define UNASSIGNED 0 7. // N is used for size of Sudoku grid. Size will be NxN 8. #define N 9 10. // This function finds an entry in grid that is still unassigned 11. **bool** FindUnassignedLocation(**int** grid[N][N], **int** &row, **int** &col); 13. // Checks whether it will be legal to assign num to the given row,col 14. **bool** isSafe(**int** grid[N][N], **int** row, **int** col, **int** num); 16. /\* Takes a partially filled-in grid and attempts to assign values to 17. all unassigned locations in such a way to meet the requirements 18. for Sudoku solution (non-duplication across rows, columns, and boxes) \*/ 19. **bool** SolveSudoku(**int** grid[N][N]) 20. { 21. **int** row, col; 23. // If there is no unassigned location, we are done 24. **if** (!FindUnassignedLocation(grid, row, col)) 25. **return** **true**; // success! 27. // consider digits 1 to 9 28. **for** (**int** num = 1; num <= 9; num++) 29. { 30. // if looks promising 31. **if** (isSafe(grid, row, col, num)) 32. { 33. // make tentative assignment 34. grid[row][col] = num; 36. // return, if success, yay! 37. **if** (SolveSudoku(grid)) 38. **return** **true**; 40. // failure, unmake & try again 41. grid[row][col] = UNASSIGNED; 42. } 43. } 44. **return** **false**; // this triggers backtracking 45. } 47. /\* Searches the grid to find an entry that is still unassigned. If 48. found, the reference parameters row, col will be set the location 49. that is unassigned, and true is returned. If no unassigned entries 50. remain, false is returned. \*/ 51. **bool** FindUnassignedLocation(**int** grid[N][N], **int** &row, **int** &col) 52. { 53. **for** (row = 0; row < N; row++) 54. **for** (col = 0; col < N; col++) 55. **if** (grid[row][col] == UNASSIGNED) 56. **return** **true**; 57. **return** **false**; 58. } 60. /\* Returns a boolean which indicates whether any assigned entry 61. in the specified row matches the given number. \*/ 62. **bool** UsedInRow(**int** grid[N][N], **int** row, **int** num) 63. { 64. **for** (**int** col = 0; col < N; col++) 65. **if** (grid[row][col] == num) 66. **return** **true**; 67. **return** **false**; 68. } 70. /\* Returns a boolean which indicates whether any assigned entry 71. in the specified column matches the given number. \*/ 72. **bool** UsedInCol(**int** grid[N][N], **int** col, **int** num) 73. { 74. **for** (**int** row = 0; row < N; row++) 75. **if** (grid[row][col] == num) 76. **return** **true**; 77. **return** **false**; 78. } 80. /\* Returns a boolean which indicates whether any assigned entry 81. within the specified 3x3 box matches the given number. \*/ 82. **bool** UsedInBox(**int** grid[N][N], **int** boxStartRow, **int** boxStartCol, **int** num) 83. { 84. **for** (**int** row = 0; row < 3; row++) 85. **for** (**int** col = 0; col < 3; col++) 86. **if** (grid[row + boxStartRow][col + boxStartCol] == num) 87. **return** **true**; 88. **return** **false**; 89. } 91. /\* Returns a boolean which indicates whether it will be legal to assign 92. num to the given row,col location. \*/ 93. **bool** isSafe(**int** grid[N][N], **int** row, **int** col, **int** num) 94. { 95. /\* Check if 'num' is not already placed in current row, 96. current column and current 3x3 box \*/ 97. **return** !UsedInRow(grid, row, num) && 98. !UsedInCol(grid, col, num) && 99. !UsedInBox(grid, row - row % 3, col - col % 3, num); 100. } 102. /\* A utility function to print grid  \*/ 103. **void** printGrid(**int** grid[N][N]) 104. { 105. **for** (**int** row = 0; row < N; row++) 106. { 107. **for** (**int** col = 0; col < N; col++) 108. printf("%2d", grid[row][col]); 109. printf("\n"); 110. } 111. } 113. /\* Driver Program to test above functions \*/ 114. **int** main() 115. { 116. // 0 means unassigned cells 117. **int** grid[N][N] = { { 3, 0, 6, 5, 0, 8, 4, 0, 0 }, 118. { 5, 0, 0, 0, 0, 0, 0, 0, 0 }, 119. { 0, 8, 7, 0, 0, 0, 0, 3, 1 }, 120. { 0, 0, 3, 0, 1, 0, 0, 8, 0 }, 121. { 9, 0, 0, 8, 6, 3, 0, 0, 5 }, 122. { 0, 5, 0, 0, 9, 0, 6, 0, 0 }, 123. { 1, 3, 0, 0, 0, 0, 2, 5, 0 }, 124. { 0, 0, 0, 0, 0, 0, 0, 7, 4 }, 125. { 0, 0, 5, 2, 0, 6, 3, 0, 0 } }; 126. **if** (SolveSudoku(grid) == **true**) 127. printGrid(grid); 128. **else** 129. printf("No solution exists"); 131. **return** 0; 132. } |

1. DP, to find the max sum when reaching [i][j], we only need to know the max sum to reach [i-1][j] and [i][j-1].

|  |
| --- |
| 1. #include <iostream> 3. **using** **namespace** std; 4. **int** max(**int** a, **int** b) 5. { 6. **return** a > b ? a : b; 7. } 9. **int** main() 10. { 11. **const** **int** N = 3; 12. **const** **int** M = 3; 13. **int** board[N][M]; 14. **int** sum[N][M]; 15. **int** i, j; 16. **for** (i = 0; i < N; i++) 17. { 18. **for** (j = 0; j < M; j++) 19. { 20. board[i][j] = rand() % 10; 21. sum[N][M] = 0; 22. } 23. } 24. **for** (i = 0; i < N; i++) 25. { 26. **for** (j = 0; j < M; j++) 27. { 28. cout<<board[i][j]<<' '; 29. } 30. cout << endl; 31. } 32. //assume we need to find the max sum for reaching [i][j], then the only possible pathes to get there are from top or left, if we found 33. //optimum sum for reaching [i-1][j] and [i][j-1] we can then decide which one is the better way to reach [i][j] 34. sum[0][0] = board[0][0]; 35. **for** (i = 0; i < N; i++) 36. { 37. **for** (j = 0; j < M; j++) 38. { 39. **if** (i == 0 && j>0) 40. { 41. sum[i][j] = board[i][j] + sum[i][j - 1]; 42. } 43. **else** **if** (j == 0 && i>0) 44. { 45. sum[i][j] = board[i][j] + sum[i-1][j]; 46. } 47. **else** **if** (i > 0 && j>0) 48. { 49. sum[i][j] = board[i][j] + max(sum[i - 1][j], sum[i][j - 1]); 50. } 51. } 52. } 53. cout << sum[i - 1][j - 1]; 54. } |