창의적 소프트웨어 설계 경진대회

“Bejeweled”

제출 기한

12월 16일 금요일 23:59 까지 제출 - zip 파일로 Makefile, Source code 묶어서 개별 제출 (**조교 이메일**)
제출 시 반드시 이메일 제목 통일: 201XXXXXXX 홍길동
평가 기준 - 개인전, 취득 점수 순.
12/16 23:59 이전까지 반복해서 제출 가능.

문제

- 매 턴마다 n개의 보석으로 이루어진 w x h 입력이 주어지고 하나의 액션을 선택할 수 있다.
- 액션은 주어진 좌표의 보석을 상,하,좌,우의 것과 바꾸는 것이다.
- 액션으로 세개 이상의 같은 종류의 보석이 가로, 세로로 연결되면 해당 보석들이 없어지면서 점수를 얻는다.
- 없어진 부분은 위쪽의 보석들이 떨어지면서 채우고 보이지 않던 부분이 내려온다.
- 콤보: 채워지면서 세개 이상의 보석이 연결되면 같은 방식으로 없어지며 더이상 연결된 보석이 없으면 그 다음 입력을 받는다. 각 단계의 점수는 다음과 같은 식으로 계산한다. (콤보수는 0부터 시작)
  static_cast<int>(<없어진_개수> * 10 * pow(1.5, <콤보수>))
  예) 3개 : 30점, 3개+3개 : 75점, 3개+3개+3개 : 143점, 3개+3개+3개+3개 : 244점
- 올바르지 않은 입력(판 바깥 위치와 바꾸는 경우 등)은 점수를 얻지 못한다.
- 프로그램이 비정상 종료하는 경우 0점 처리한다.
- N 턴이 지난 후 점수를 가장 많이 얻은 사람이 우승한다.

입출력

보석의 종류는 ‘a’ 부터 n 종류의 알파벳을 사용하여 표시한다. (예: n=3 인경우 a, b, c)
w, h, n 은 처음 실행시 주어진다.
좌표는 좌상단을 (0,0), 우하단을 (w-1, h-1)로 본다. 액션은 상:U, 하:D, 좌:L, 우:R 로 지정한다.
출력은 0.1초 이내에 하도록 한다.

예) 파란색 - 입력 / 빨간색 - 출력 / 회색 - 설명

```
7 5 5                 // 입력으로부터 w=7, h=5, n=5 (a~e)
c a b c d c d          // 게임 시작
a b b c c d c
b a a d c d d
a d c d b a b
c b d e e a c
1 2 L                 // (1,2)의 'a'를 왼쪽으로 이동
30                    // 현재 점수
a a b c d c d          // 이전 액션으로 없어지고 난 후의 상태
e b b c c d c
b b a d c d d
c d c d b a b
c b d e e a c
0 2 U                 // (0,2)의 'b'를 위쪽으로 이동
…
```

심판 프로그램 (contest3.py)
실행: python contest3.py ./refree ./player

contest3.py

```
import time
import os
import sys
from subprocess import Popen, PIPE, STDOUT

if len(sys.argv) < 3:
    sys.exit('usage: contest3 refree player')

max_loop = 100
width = 5
height = 5
num_char = 5

refree_pipe = Popen([sys.argv[1], str(width), str(height), str(num_char)],
                    stdin=PIPE, stdout=PIPE, bufsize=1)
player_pipe = Popen([sys.argv[2]], stdin=PIPE, stdout=PIPE, bufsize=1)

format_str = str(width) + ' ' + str(height) + ' ' + str(num_char)
player_pipe.stdin.write(format_str + '\n')

loop = 1
while loop <= max_loop:
    for i in range(0, height):
        board_line = refree_pipe.stdout.readline()
        player_pipe.stdin.write(board_line + '\n')

    move = player_pipe.stdout.readline()
    print("[" + str(loop) + "] move: " + move)
    refree_pipe.stdin.write(move + '\n')

    score = refree_pipe.stdout.readline()
    print("[" + str(loop) + "] score: " + score)
    player_pipe.stdin.write(score + '\n')

    time.sleep(0.1)
    loop = loop + 1

player_pipe.kill()
player_pipe.wait()
refree_pipe.kill()
refree_pipe.wait()

sys.exit()
```

refree.cpp

```
#include <stdlib.h>
#include <algorithm>
#include <iostream>
#include <string>
#include <map>
#include <set>
#include <vector>

using namespace std;

namespace {
```

```
typedef pair<int, int> Coord;

struct Move {
  Coord coord;
  int dir;
};

inline int Random(int n) {
  return static_cast<int>(static_cast<double>(rand()) / RAND_MAX * n);
}

inline Coord GetCoordInDir(const Coord& coord, int dir) {
  Coord ret = coord;
  switch (dir) {
    case 0: ret.second -= 1; break;  // Up.
    case 1: ret.second += 1; break;  // Down.
    case 2: ret.first -= 1; break;  // Left.
    case 3: ret.first += 1; break;  // Right.
  }
  return ret;
}

class Board {
 public:
  Board(int w, int h, int n) : board_(), width_(w), height_(h), num_char_(n) {}
  Board(const Board& b) : board_(b.board_), width_(b.width_),
                          height_(b.height_), num_char_(b.num_char_) {}

  int width() const { return width_; }
  int height() const { return height_; }
  int num_char() const { return num_char_; }

  char get(int x, int y) const { return get(Coord(x, y)); }

  char get(const Coord& c) const {
    map<Coord, char>::const_iterator it = board_.find(c);
    return it == board_.end() ? '\0' : it->second;
  }

  void Initialize();
  int MoveCell(int x, int y, char dir);

  ostream& DumpPossibleMove(ostream& os) const {
    Move move;
    if (FindPossibleMove(&move) == false) {
      os << "no possible move." << endl;
    } else {
      const char dir[] = "udlr";
      os << "try " << move.coord.first << " " << move.coord.second
         << " " << dir[move.dir] << endl;
    }
    return os;
  }

 private:
  char RandomChar() const { return 'a' + Random(num_char_); }

  bool SwapCells(const Coord& c0, const Coord& c1) {
    map<Coord, char>::iterator it0 = board_.find(c0), it1 = board_.find(c1);
    if (c0 == c1 || it0 == board_.end() || it1 == board_.end()) return false;
    std::swap(it0->second, it1->second);
    return true;
  }

  bool FindMatchesAt(const Coord& c, set<Coord>* matches) const {
    return FindMatchesAt(c.first, c.second, matches);
```

```
  }
  bool FindMatchesAt(int x, int y, set<Coord>* matches) const;
  bool FindAllMatches(set<Coord>* matches) const;
  void ClearAllMatches(const set<Coord>& matches, vector<int>* new_cells);

  bool FindPossibleMove(Move* move) const;
  bool CheckPossibleMove() const { return FindPossibleMove(NULL); }

  ostream& Dump(ostream& os, const set<Coord>& matches) const {
    for (int y = 0; y < height_; ++y) {
      for (int x = 0; x < width_; ++x) {
        os << get(x, y) << (matches.count(Coord(x, y)) ? "<" : " ");
      }
      os << endl;
    }
    return os;
  }

  map<Coord, char> board_;
  int width_, height_, num_char_;
};

ostream& operator<<(ostream& os, const Board& b) {
  for (int y = 0; y < b.height(); ++y) {
    if (b.width() > 0) os << b.get(0, y);
    for (int x = 1; x < b.width(); ++x) os << " " << b.get(x, y);
    os << endl;
  }
  return os;
}

bool Board::FindMatchesAt(int x, int y, set<Coord>* matches) const {
  const char ch = get(x, y);
  if (ch == '\0') return false;
  // Match horizontally and vertically.
  set<Coord> horz_match, vert_match;
  for (int u = x; u < width_; ++u) {
    if (get(u, y) == ch) horz_match.insert(Coord(u, y));
    else break;
  }
  for (int u = x - 1; u >= 0; --u) {
    if (get(u, y) == ch) horz_match.insert(Coord(u, y));
    else break;
  }
  for (int v = y; v < height_; ++v) {
    if (get(x, v) == ch) vert_match.insert(Coord(x, v));
    else break;
  }
  for (int v = y - 1; v >= 0; --v) {
    if (get(x, v) == ch) vert_match.insert(Coord(x, v));
    else break;
  }
  if (horz_match.size() < 3 && vert_match.size() < 3) return false;
  if (horz_match.size() >= 3 && matches != NULL) {
    matches->insert(horz_match.begin(), horz_match.end());
  }
  if (vert_match.size() >= 3 && matches != NULL) {
    matches->insert(vert_match.begin(), vert_match.end());
  }
  return true;
}

bool Board::FindAllMatches(set<Coord>* matches) const {
  set<Coord> all_matches;
  for (int y = 0; y < height_; ++y) {
    for (int x = 0; x < width_; ++x) {
```

```
      FindMatchesAt(x, y, &all_matches);
    }
  }
  if (matches != NULL) *matches = all_matches;
  return !all_matches.empty();
}

bool Board::FindPossibleMove(Move* move) const {
  Board test_board(*this);
  for (int y = height_ - 1; y >= 0; --y) {
    for (int x = 0; x < width_; ++x) {
      const Coord c0(x, y);
      for (int d = 0; d < 4; ++d) {
        const Coord c1 = GetCoordInDir(c0, d);
        if (test_board.SwapCells(c0, c1)) {
          if (test_board.FindMatchesAt(c0, NULL) ||
              test_board.FindMatchesAt(c1, NULL)) {
            if (move != NULL) move->coord = c0, move->dir = d;
            return true;
          }
          test_board.SwapCells(c0, c1);  // Restore to the original state.
        }
      }
    }
  }
  return false;
}

void Board::ClearAllMatches(const set<Coord>& matches, vector<int>* new_cells) {
  if (matches.empty()) return;
  for (int x = 0; x < width_; ++x) {
    int cur_y = height_ - 1;
    for (int y = height_ - 1; y >= 0; --y) {
      if (matches.count(Coord(x, y)) == 0) {
        board_[Coord(x, cur_y)] = get(x, y);
        --cur_y;
      }
    }
    while (cur_y >= 0) {
      Coord c(x, cur_y--);
      board_[c] = RandomChar();
      if (new_cells != NULL) (*new_cells)[x] += 1;
    }
  }
}

void Board::Initialize() {
  // Initialize the empty cells in the board with random characters.
  board_.clear();
  for (int y = 0; y < height_; ++y) {
    for (int x = 0; x < width_; ++x) {
      board_.insert(make_pair(Coord(x, y), RandomChar()));
    }
  }
  // Make sure that there exist no match and at least one possible move.
  set<Coord> matches;
  while (FindAllMatches(&matches) == true || CheckPossibleMove() == false) {
    ClearAllMatches(matches, NULL);
    int x = Random(width_), y = Random(height_);
    board_[Coord(x, y)] = RandomChar();
  }
}

int Board::MoveCell(int x, int y, char dir) {
  int d = dir == 'U' ? 0 : dir == 'D' ? 1 : dir == 'L' ? 2 : dir == 'R' ? 3 :
          -1;
```

```
  if (d < 0) return 0;
  Coord c0(x, y);
  Coord c1 = GetCoordInDir(c0, d);
  if (SwapCells(c0, c1) == false) return 0;
  cerr << "MoveCell" << endl;
  int score = 0;
  double ratio = 1.0;
  set<Coord> matches;
  vector<int> new_cells(width_, 0);
  while (FindAllMatches(&matches)) {
    Dump(cerr, matches) << "  " << matches.size() << " matches found" << endl;
    score += static_cast<int>(ratio * matches.size() * 10);
    ratio *= 1.5;
    ClearAllMatches(matches, &new_cells);
  }
  if (score == 0) {
    SwapCells(c0, c1);  // Restore to the original board.
  } else {
    while (CheckPossibleMove() == false) {
      Coord c;
      do {
        c.first = Random(width_);
      } while (new_cells[c.first] <= 0);
      c.second = Random(new_cells[c.first]);
      const char org_ch = board_[c];
      board_[c] = RandomChar();
      if (FindAllMatches(NULL) == true) board_[c] = org_ch;
    }
  }
  return score;
}

}  // anonymous namespace

int main(int argc, char** argv) {
  if (argc != 4) {
    cout << "usage: " << *argv << " <width> <height> <num_char>" << endl;
    return -1;
  }
  const int w = atoi(argv[1]);
  const int h = atoi(argv[2]);
  const int n = atoi(argv[3]);
  cerr << w << " " << h << " " << n << endl;

  // Initialize the board;
  Board board(w, h, n);
  board.Initialize();
  cout << board;
  int score = 0;
  do {
    int x, y;
    string dir;
    cin >> x >> y >> dir;
    if (dir == "?") {
      board.DumpPossibleMove(cerr);
      continue;
    }
    score += board.MoveCell(x, y, dir[0]);
    cerr << board;
    cout << score << endl;
    cout << board;
  } while (cin.good());
  return 0;
}
```