# CS302 -- Lab 5 -- Superball!

- **CS302 -- Fundamental Algorithms**
- **Spring, 2022**
- **James S. Plank**
- **This file: http://web.eecs.utk.edu/~jplank/plank/classes/cs302/Labs/Lab5/**

*Tue Nov 27 16:37:58 EST 2007. Last revision: Fri Jan 21 10:08:31 EST 2022*

## What you hand in

This is split into two labs on Canvas:

- Lab 5A is for you to submit **sb-analyze.cpp**.
- Lab 5B is for you to submit **sb-play.cpp**.

They are both due on the same day, and will be graded independently. If you are late with one, but not the other, the late points will only be applied to the late one.

Please do not submit anything other than those programs. The TA's will compile them with the disjoint set implementation described below.

## Getting Started

You should start by going to whatever directory you'll be working in (on our lab machines), and doing:

```
UNIX> sh /home/jplank/cs302/Labs/Lab5/scripts/start_lab.sh
UNIX> git clone https://jimplank@bitbucket.org/jimplank/plank-disjoint-sets.git            # You'll need a bitbucket account for this one.
UNIX> cd plank-disjoint-sets
UNIX> make clean ; make
UNIX> cd ..
UNIX> make
```

If all is successful, then you'll have three programs in your **bin** directory:

```
UNIX> ls bin
sb-analyze      sb-play         sb-read
UNIX> bin/sb-read
usage: sb-read rows cols min-score-size colors
UNIX> bin/sb-analyze
This program doesn't do anything yet.
UNIX> bin/sb-play
This program doesn't do anything yet.
UNIX>
```

## How you are graded

The gradescript tests **sb-analyze**. It will be graded as a 50-point lab. For **sb-play**, we grade it by running **sh scripts/run_multiple.sh** on it for at least 10 runs. It too is graded as a 50-point lab.

Remember, (and I know I'm repeating myself here), that you are only submitting two programs -- the TA's will compile with their own copy of the disjoint set code.

## Also

Every year, someone asks me for the source to **sb-player**. Sorry, but I can't give it out, because it's too easy to modify it to solve the lab. I can try to make an **sb-player** binary for your machine, and if you want modifications, I'll listen. Let me know.

There is an **sb-player** binary for macs in **mac-binary/sb-player-mac**.

Plus, in 2015, Alex Teepe wrote a multiplatform Superball player to share. I have not tried it, but please do. Thanks, Alex!

https://drive.google.com/file/d/0B4rzPrfwFCsKbUpwd21pMlgtc1E/view.

There is a README here.

## Disjoint Sets

If you've done the command above correctly, in your directory **plank-disjoint-sets**, there is an implementation of disjoint sets that comes from https://bitbucket.org/jimplank/plank-disjoint-sets. You should have compiled the code in that directory, and then in the makefile, it will:

- Make sure that when you say `#include "disjoint_set.hpp"`, it will find it from **plank-disjoint-sets/include**.

- Link in **plank-disjoint-sets/obj/disjoint_set.o**, so that you can use this implementation
- When you type `make` in the current directory, it should make dummy versions of **bin/sb-analyze** and **bin/sb-play**. If not, you have something wrong with your compilation.

You are not allowed to modify the disjoint set code -- you should use it as is. The TA's will compile your code with their versions of the disjoint set code, so you can't customize it. Moreover, you shouldn't copy the code into your source files, or copy it from the lecture notes, and then modify it for your lab.

**If you don't understand how to compile your program correctly, please ask the TA's or ask on Piazza. DO NOT COPY THE DISJOINT SET CODE FROM LECTURE AND INCLUDE IT WITH YOUR PROGRAM. Instead, use the code from bitbucket, as specified above**.

---

## Superball

Superball is a simplistic game that was part of a games CD for my old Windows 95 box. It works as follows. You have a 8x10 grid which is the game board. Each cell of the game board may be empty or hold a color:

- P - Purple: worth 2 points.
- B - Blue: worth 3 points.
- Y - Yellow: worth 4 points.
- R - Red: worth 5 points.
- G - Green: worth 6 points.

The board starts with five random colors set. On your turn, you may do one of two things:

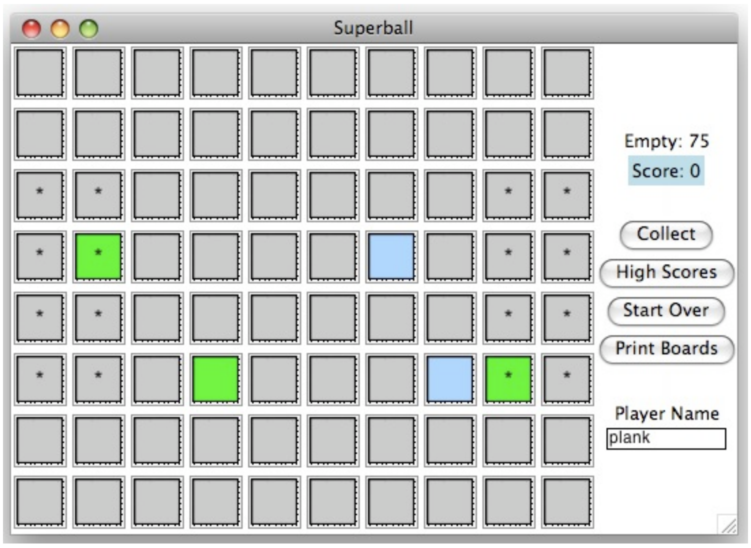
- You may swap two cells. After the swap, five new random cells will be filled with a random colors.
- You may "score" a cell. To score a cell, the cell must be one of the "goal" cells, and there are sixteen of these, in rows 2-5, columns 0, 1, 8 and 9. (Everything is zero indexed). Moreover, there must be at least five touching cells of the same color, one of which must be the goal cell that you want to score. When you score, you get the sum of the cells connected to the cell that you are scoring, and then all of those cells leave the board, and three new random ones are added.

I have a tcl/tk/shell-scripted Superball player at **/home/jplank/Superball**. Simply copy that directory to your home directory:

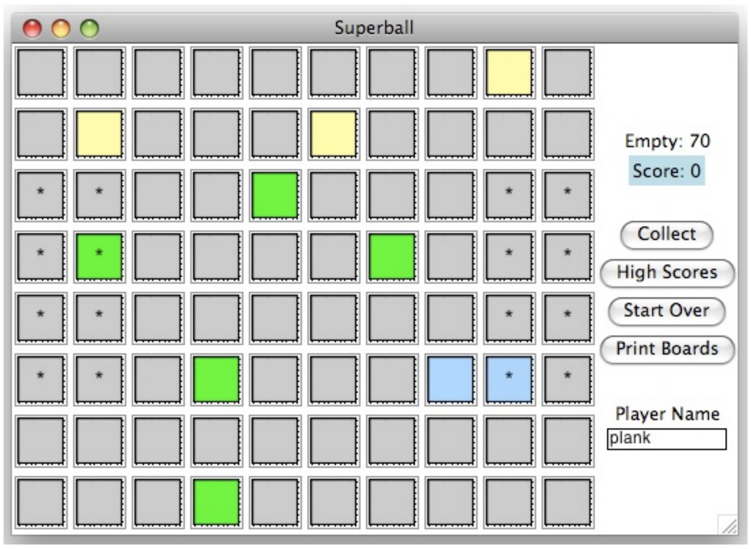
```
UNIX> cp -r /home/jplank/Superball $HOME
```

Then you can play it with **~/Superball/Superball**. The high score probably won't work -- you'll have to change the **open** command in the file **hscore** to the name of your web browser.

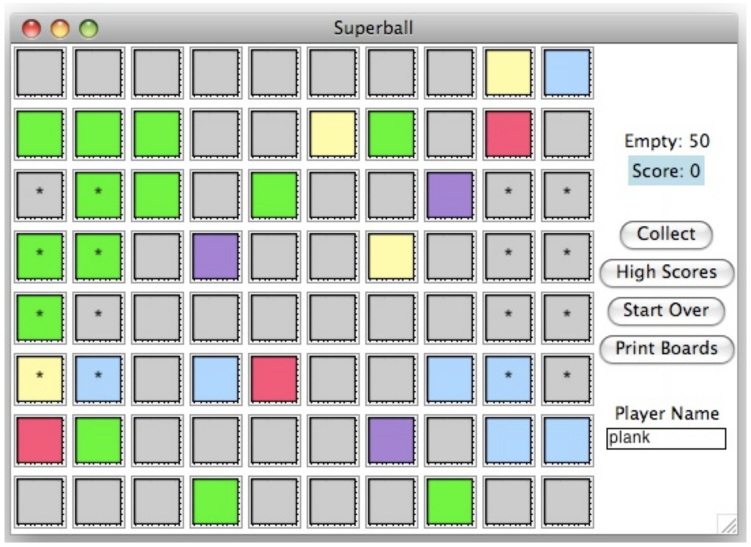
Let's look at some screen shots. Suppose we fire up **Superball**:



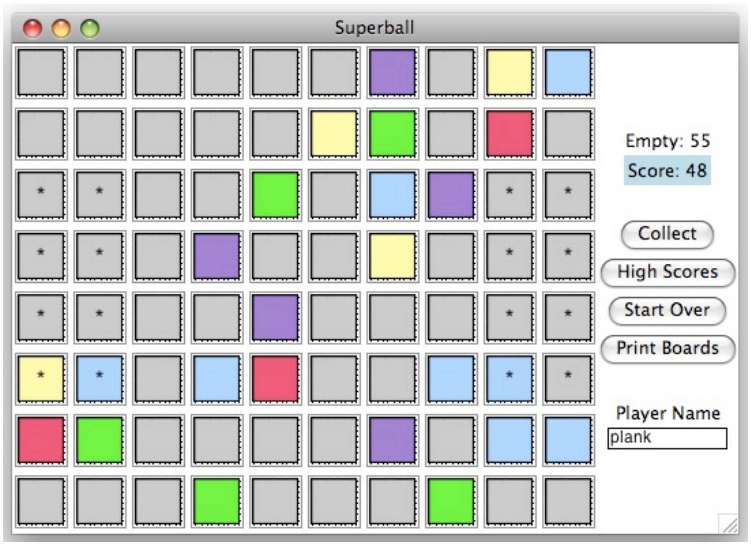
The "goal" cells are marked with asterisks, and there are five non-empty cells. Our only legal action is to swap two cells -- I'm going to swap cells [3,6] and [5,8]. This will make those two blue cells contiguous. In the game, I do that by clicking on the two cells that I want to swap. Afterwards, five new cells are put on the screen. Here's the screen shot:



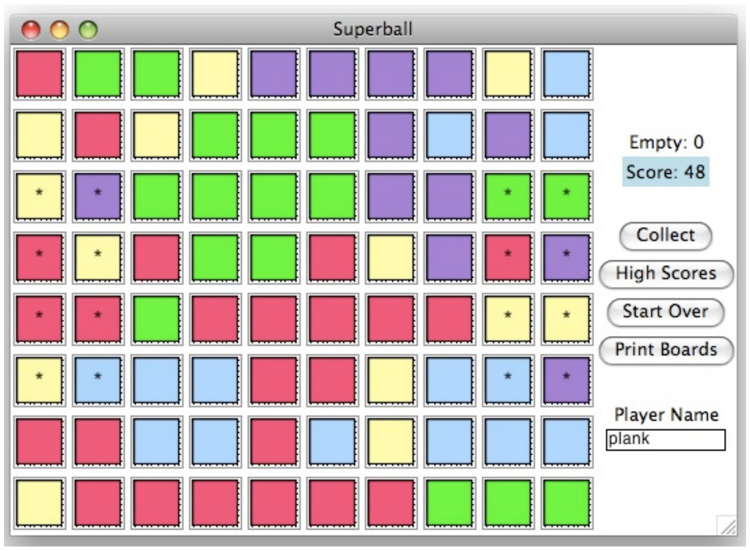
I do a bunch more swaps and end up with the following board:



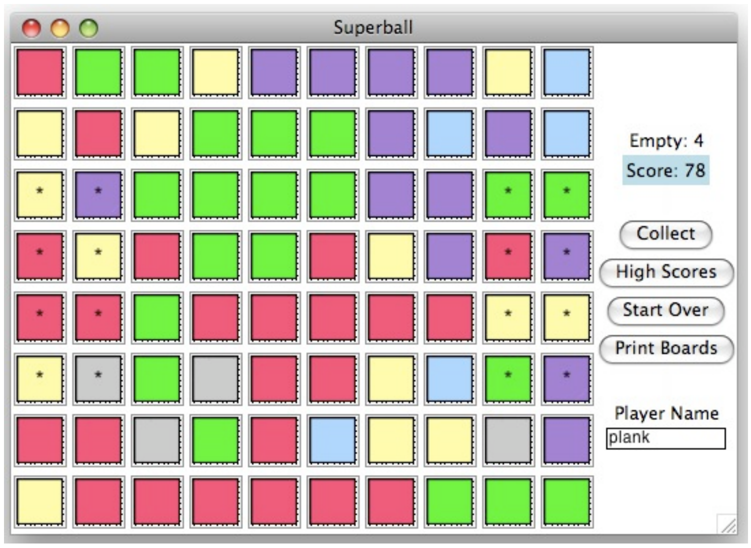
I can score the green cells by clicking on cell [2,1], [3,0], [3,1] or [4,0] and then clicking "Collect". This will score that group of eight green squares, which gets me 48 points (8*6), and three new cells will be added:



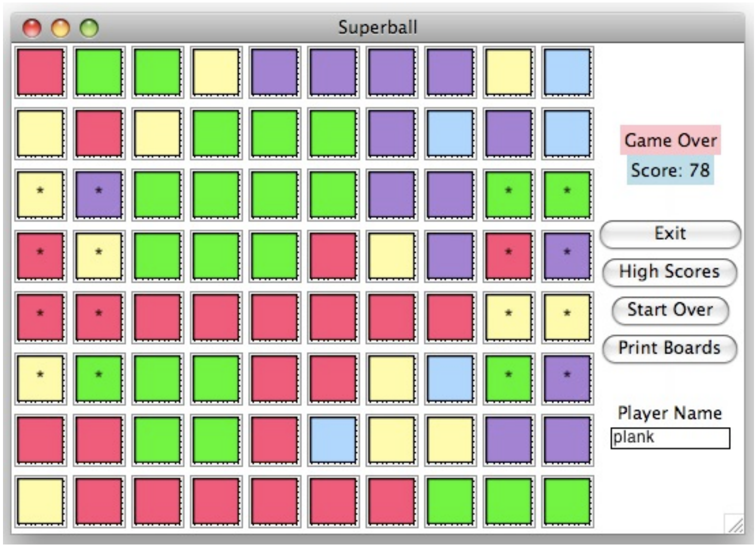
There are no cells to score here (the blues ones in the lower right-hand part of the board only compose a group of four). So I revert to swapping. Suppose I keep doing so until I reach:



I'm in trouble. I've got these beautiful groups of red, green and purple cells, but I can't score any of them because they are not connected to a goal call. Dang. I can only score those two groups of blue cells. When I do that, I'm only left with four open squares, and I can't score anything:



Perhaps I should have been a little more thoughtful while playing the game. Regardless, I'm stuck. I simply swap two random squares and end the game:



Oh well -- should have done that swap a little sooner....

---

For this lab, we are going to deal with a text-based version of the game. Our programs will have the following parameters:

- **rows** - the number of rows on the game board. Although the tcl/tk version has that set to eight, our programs will handle any number.
- **cols** - the number of columns on the game board.
- **min-score-size** - the number of contiguous cells that have to be together in order to score them. This is 5 in the tcl/tk version
- **colors** - this must be a string of distinct lower-case letters. They represent that the colors that a cell can have. The point value of the first of these is 2, and each succeeding character is worth one more point. To have the same values as the tcl/tk game, this parameter should be "pbyrg".

I have written an interactive game player. I'll discuss all the parameters later. Call it as done below:

```
UNIX> cd /home/jplank/cs302/Labs/Lab5/
UNIX> bin/sb-player
usage: sb-player rows cols min-score-size colors player interactive(y|n) output(y|n) seed
UNIX> bin/sb-player 8 10 5 pbyrg - y y -
Empty Cells: 75     Score: 0

..........
..........
**b....b**
**....b.**
**.g....**
**......**
..........
..g.......

Your Move:
```

The format of the board is as follows: When a letter is capitalized, it is on a goal cell. Dots and asterisks stand for empty cells -- asterisks are on the goal cells. If you click on the **Print Boards** button in the tcl/tk game, it will print out each board on standard output in that format. That's nice for testing.

You can type two commands:

```
SWAP r1 c1 r2 c2
SCORE r c
```

In the board above, you can't score anything, so you'll have to swap. We'll swap the blue cell in [2,2] with the green one in [7,2]:

```
Your Move: SWAP 2 2 7 2

Empty Cells: 70     Score: 0

.r........
..........
**g....b**
**....b.**
**.g....*Y
**......*P
.....rr...
..b.......

Your Move:
```

It's incredibly tedious -- play along with me:

```
Empty Cells: 70     Score: 0

.r........
..........
**g....b**
**....b.**
**.g....*Y
**......*P
.....rr...
..b.......

Your Move: SWAP 0 1 7 2
```

```
Empty Cells: 65     Score: 0

.b........
..........
**g....bB*
**....b.**
P*.g....RY
**......*P
.....rr...
.gry......

Your Move: SWAP 7 3 4 8
```

```
Empty Cells: 60     Score: 0

.b.......p
....g.....
**g.p..bB*
**r...b.*R
P*.g....YY
**......*P
.....rr...
.grr......

Your Move: SWAP 3 2 7 1
```

```
Empty Cells: 55     Score: 0

.b..r...pp
....g...b.
**g.p..bB*
**g...b.*R
P*.g....YY
**.g....*P
.....rr...
rrrr......

Your Move: SWAP 3 9 0 1
```

```
Empty Cells: 50     Score: 0

.r..rgy.pp
....g...b.
**g.p..bB*
**g...b.*B
P*.g....YY
**.g....*P
p...rrr...
rrrr.p....

Your Move: SWAP 6 0 0 1
```

```
Empty Cells: 45     Score: 0

.p..rgy.pp
.g..g...b.
**g.p..bB*
**g...b.*B
P*.g..y.YY
**.g..yp*P
r...rrr...
rrrr.py...

Your Move: SWAP 5 9 7 6
```

```
Empty Cells: 40     Score: 0

.p..rgy.pp
.g..g...b.
**g.p.pbB*
R*g...by*B
P*.g..y.YY
P*.g..yp*Y
r...rrrb..
rrrr.pp...

Your Move: SWAP 5 0 0 4
```

```
Empty Cells: 35     Score: 0

.p..pgy.pp
.g..g.r.b.
G*g.p.pbB*
R*g...by*B
P*.g..y.YY
R*.g..yp*Y
r..grrrb..
rrrrbppy..

Your Move: SWAP 7 4 1 6
```

```
Empty Cells: 30     Score: 0

.p..pgy.pp
.g.pg.b.b.
G*g.p.pbB*
R*g.r.by*B
P*pg..y.YY
R*.g.bypBY
r..grrrb..
rrrrrppy..

Your Move: SCORE 5 0
```

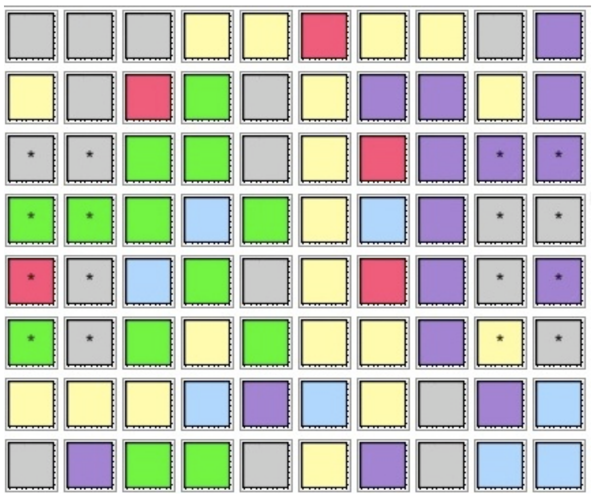
```
Empty Cells: 37     Score: 50

.p..pgy.pp
.g.pg.b.by
G*g.p.pbB*
R*g.r.byGB
P*pg..y.YY
**.g.bypBY
...g...b..
.p...ppy..

Your Move:
```

You'll note, I could have scored cell [5,0] when there were 35 empty cells, but I really wanted to make that patch of red cells bigger.

---

## Program #1: Sb-read

I have written src/sb-read.cpp for you. This program takes the four parameters detailed above, reads in a game board with those parameters and prints out some very basic information. For example, the following board:



May be represented by the following text (in txt/input-1.txt):

```
...yyryy.p
y.rg.yppyp
**gg.yrpPP
GGgbgybp**
R*bg.yrp*P
G*gygyypY*
yyybpby.pb
.pgg.yp.bb
```

When we run **bin/sb-read** on it, we get the following:

```
UNIX> bin/sb-read 8 10 5 pbyrg < txt/input-1.txt
Empty cells:                    20
Non-Empty cells:                60
Number of pieces in goal cells:  8
Sum of their values:            33
UNIX>
```

There are three purple pieces in goal cells, one yellow, three green and one red. That makes a total of 3*2 + 4 + 5 + 3*6 = 33.

You should take a look at src/sb-read.cpp. In particular, look at the **Superball** class:

```
class Superball {
  public:
    Superball(int argc, char **argv);
    int r;
    int c;
    int mss;
    int empty;
    vector <int> board;
    vector <int> goals;
    vector <int> colors;
};
```

**Mss** is min-score-size. **Empty** is the number of empty cells in the board. **Board** is a vector of **r * c** integers. The element in [i,j] is in entry **board[i*c+j]**, and is either '.', '*' or a lower case letter. **goals** is another array of **r * c** integers. It is equal to 0 if the cell is not a goal cell, and 1 if it is a goal cell. **Colors** is an array of 256 elements, which should be indexed by a letter. Its value is the value of the letter (e.g. in the above example, **colors['p'] = 2**).

**sb-read** does all manner of error checking for you. It is a nice program from which to build your other programs.

---

## Program #2: Sb-analyze

You are to write this one.

With **bin/sb-analyze**, you are to start with **sb-read.cpp** as a base, and augment it so that it prints out all possible scoring sets. For example, in the above game board (represented by txt/input-1.txt), there are two scoring sets -- the set of 10 purple cells in the upper right-hand corner, and the set of 6 green cells on the left side of the screen. Here is the output to **sb_analyze**:

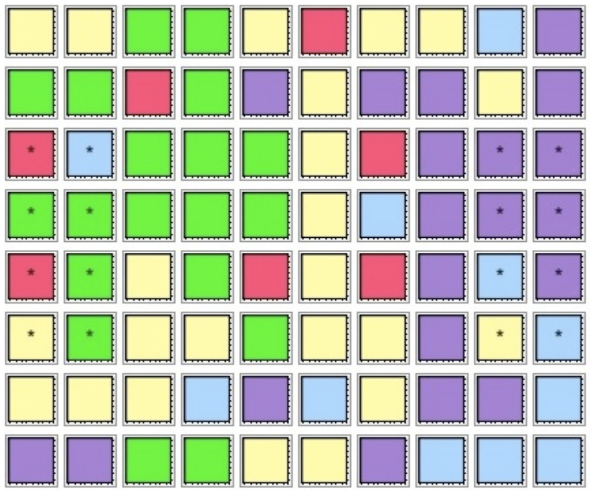
```
UNIX> sb-analyze
usage: sb-analyze rows cols min-score-size colors
UNIX> bin/sb-analyze 8 10 5 pbyrg < txt/input-1.txt
Scoring sets:
  Size: 10  Char: p  Scoring Cell: 2,8
  Size:  6  Char: g  Scoring Cell: 3,0
UNIX>
```

Each set must be printed exactly once, but in any order, and with any legal goal cell. Thus, the following output would also be ok:

```
UNIX> bin/sb-analyze 8 10 5 pbyrg < txt/input-1.txt
Scoring sets:
  Size:  6  Char: g  Scoring Cell: 3,1
  Size: 10  Char: p  Scoring Cell: 2,9
UNIX>
```

Think about how you would use the disjoint sets data structure to implement this -- it is a straightforward connected components application. I would recommend augmenting your **Superball** class with a **Disjoint_Set**, and then having a method called **analyze_superball()**, which performs the analysis.

Here's another example:



This is in the file txt/input-2.txt:

```
yyggyryybp
ggrgpyppyp
RBgggyrpPP
GGgggybpPP
RGygryrpBP
YGyygyypYB
yyybpbyppb
ppggyypbbb
```

```
UNIX> bin/sb-analyze 8 10 5 pbyrg < txt/input-2.txt
Scoring sets:
  Size: 14  Char: g  Scoring Cell: 5,1
  Size: 15  Char: p  Scoring Cell: 4,9
  Size:  7  Char: y  Scoring Cell: 5,0
  Size:  5  Char: b  Scoring Cell: 5,9
UNIX>
```

## Program #3: Sb-play

Your next program takes the same arguments and input as **sb-analyze**. However, now its job is to print a single move as would be accepted as input for the **sb-player** program. In other words, it needs to output a SWAP or SCORE line with legal values.

If you have fewer than five pieces and cannot score any, you will lose the game -- you should do that by swapping two legal pieces so that the game can end.

The **sb-player** program takes as its 5th argument the name of a program that it will use for input. I also have three programs - **sb-play**, **sb-play2** and **sb-play3** in that directory. **sb-play** simply swaps two random cells until there are fewer than five empty, then it scores a set if it can. The other two are smarter, but are by no means the best one can do.

Here's **sb-player** running on **bin/sb-play2** (note, **sb-player** creates a temporary file, so you must run it from your own directory):

```
UNIX> /home/jplank/cs302/Labs/Lab5/bin/sb-player 8 10 5 pbyrg /home/jplank/cs302/Labs/Lab5/bin/sb-play2 y y -
Empty Cells: 75     Score: 0

g.........
..........
**......**
*Pr.....**
**......**
**..p...**
........b.
..........
```

```
Type Return for the next play
```

It waits for you to press the return key. When you do so, it will send the game board to **/home/jplank/cs302/Labs/Lab5/sp-play2** and perform the output. Here's what happens:

```
Move is: SWAP 5 4 3 2

Empty Cells: 70     Score: 0

g........g
.......y..
**......**
*Pp.....**
**......G*
**..r...**
..g.....b.
........g.

Type Return for the next play
```

You can bet that the next move will swap that **b** with one of the **g**'s:

```
Move is: SWAP 6 8 0 0

Empty Cells: 65     Score: 0

b........g
.......y..
**..b...**
*Pp.g...**
**.....gG*
**..r...**
..g.....g.
.p...p..g.

Type Return for the next play
```

And so on. If you run it with **n** for the 6th argument, it will simply run the program without your input:

```
UNIX> /home/jplank/cs302/Labs/Lab5/bin/sb-player 8 10 5 pbyrg /home/jplank/cs302/Labs/Lab5/bin/sb-play2 n y -
Empty Cells: 75     Score: 0

..........
..........
**......**
**y..y..**
**......**
*P......**
..........
......p.g.

Move is: SWAP 3 5 3 2
```

*... a bunch of output skipped...*

```
Empty Cells:  1     Score: 505

yyrrgggpyy
grrbppg.yg
GYbgygggPB
GBggpgbpPB
PPgggggrYB
YBbybgpbYR
pprrrggggr
byyrppppgg

Move is: SWAP 0 1 7 5

Game over.  Final score = 505
UNIX>
```

Even though there were no good moves at the end, the program did a final SWAP so that the game could finish.

If you run with the 7th argument as **n**, it will only print out the end result, and the last argument can specify a seed (it uses the current time if that argument is "-"), so that you can compare multiple players on the same game:

```
UNIX> /home/jplank/cs302/Labs/Lab5/bin/sb-player 8 10 5 pbyrg /home/jplank/cs302/Labs/Lab5/bin/sb-play n n 1
Game over.  Final score = 0
UNIX> /home/jplank/cs302/Labs/Lab5/bin/sb-player 8 10 5 pbyrg /home/jplank/cs302/Labs/Lab5/bin/sb-play2 n n 1
Game over.  Final score = 855
UNIX> /home/jplank/cs302/Labs/Lab5/bin/sb-player 8 10 5 pbyrg /home/jplank/cs302/Labs/Lab5/bin/sb-play3 n n 1
Game over.  Final score = 2572
UNIX>
```

It can take a while for these to run -- if it appears to be hanging, send the process a **QUIT** signal and it will print out what the current score is.

### Shell Script to Run Multiple Times

The file **scripts/run_multiple.sh** is a shell script to run the player on multiple seeds and average the results. Examples:

```
UNIX> sh scripts/run_multiple.sh
usage: sh scripts/run_multiple.sh r c mss colors player nruns starting_seed
UNIX> sh scripts/run_multiple.sh 8 10 5 pbyrg bin/sb-play 10 1
Run   1 - Score:     38  - Average     38.000
Run   2 - Score:      0  - Average     19.000
Run   3 - Score:      0  - Average     12.667
Run   4 - Score:     57  - Average     23.750
Run   5 - Score:      0  - Average     19.000
Run   6 - Score:      0  - Average     15.833
Run   7 - Score:     89  - Average     26.286
Run   8 - Score:     15  - Average     24.875
Run   9 - Score:      0  - Average     22.111
Run  10 - Score:     20  - Average     21.900
UNIX> sh scripts/run_multiple.sh 8 10 5 pbyrg bin/sb-play2 10 1
Run   1 - Score:    855  - Average    855.000
Run   2 - Score:    979  - Average    917.000
Run   3 - Score:    650  - Average    828.000
Run   4 - Score:    833  - Average    829.250
Run   5 - Score:    832  - Average    829.800
Run   6 - Score:   3326  - Average   1245.833
Run   7 - Score:   1507  - Average   1283.143
Run   8 - Score:   3643  - Average   1578.125
Run   9 - Score:    610  - Average   1470.556
Run  10 - Score:    862  - Average   1409.700
UNIX> sh scripts/run_multiple.sh 8 10 5 pbyrg bin/sb-play3 10 1
Run   1 - Score:   2572  - Average   2572.000
Run   2 - Score:   2708  - Average   2640.000
Run   3 - Score:    745  - Average   2008.333
Run   4 - Score:    424  - Average   1612.250
Run   5 - Score:   1888  - Average   1667.400
Run   6 - Score:   7140  - Average   2579.500
Run   7 - Score:   3475  - Average   2707.429
Run   8 - Score:   1701  - Average   2581.625
Run   9 - Score:   2699  - Average   2594.667
Run  10 - Score:   2291  - Average   2564.300
UNIX>
```

Obviously, to get a meaningful average, many more runs (than 10) will be required.

Oh, and make your programs run in reasonable time. Roughly 5 seconds for every thousand points, and if you are burning all that time, your program better be killing mine....

---

## Hints

Play the game for a bit to try to figure out some strategies. However, one good way to write a game player is to figure out a way to come up with a rating for a game board. Then when you are faced with making a move, you analyze all potential moves by trying them out (in other words, enumerate all potential swap operations) and choosing the one that gives you the resulting board with the highest rating.

Ideas for ratings? How about the total number of disjoint sets on the board (do you want that to bif or small)? Maybe some metric related to the sizes of the disjoint sets? Maybe add a fudge factor for a disjoint set that can be scored?

---

## The Superball Challenge

To get credit, your player needs to average over 100 points on runs of 100 games.

I will run a Superball tournament with all of your players with extra lab points going to the winners:

- 1st place: 40 extra lab points.
- 2nd place: 25 extra lab points.
- 3rd place: 10 extra lab points.

Dr. Emrich and I have now performed the challenge eleven times:

- CS140 in Fall, 2007: Plank
- CS302 in Fall, 2010: Plank
- CS302 in Fall, 2011: Plank
- CS302 in Fall, 2012: Plank
- CS302 in Fall, 2013: Plank
- CS302 in Fall, 2014: Plank
- CS302 in Fall, 2015: Plank
- CS302 in Fall, 2018: Plank
- CS302 in Spring, 2019: Emrich
- CS302 in Spring, 2020: Emrich
- CS302 in Fall, 2020: Plank

Here's the Superball Challenge Hall Of Fame (scores over 500):

| Rank | Average | Name | Semester |
|---|---|---|---|

| | | | |
|---|---|---|---|
| 1 | 31814.13 | Grant Bruer | CS302, Fall, 2015 |
| 2 | 24278.49 | Alexander Teepe | CS302, Fall, 2015 |
| 3 | 17367.77 | Joseph Connor | CS302, Fall, 2014 |
| 4 | 17021.37 | Cory Walker | CS302, Fall, 2014 |
| 5 | 16963.40 | Seth Kitchens | CS302, Fall, 2015 |
| 6 | 14555.83 | Ben Arnold | CS302, Fall, 2012 |
| 7 | 14555.83 | Adam Disney | CS302, Fall, 2011 |
| 8 | 13657.79 | Isaac Sikkema | CS302, Fall, 2018 |
| 9 | 12963.47 | Jake Davis | CS302, Fall, 2014 |
| 10 | 12634.29 | Jake Lamberson | CS302, Fall, 2014 |
| 11 | 11722.05 | Parker Mitchell | CS302, Fall, 2014 |
| 12 | 11418.77 | James Pickens | CS302, Fall, 2014 |
| 13 | 11380.74 | Nathan Ziebart | CS302, Fall, 2011 |
| 14 | 11291.39 | Michael Jugan | CS302, Fall, 2010 |
| 15 | 10576.96 | Tyler Shields | CS302, Fall, 2014 |
| 16 | 8770.67 | Nathan Swartz | CS302, Spring, 2019 |
| 17 | 7475.07 | Jared Smith | CS302, Fall, 2014 |
| 18 | 7216.28 | Michael Bowie | CS302, Fall, 2018 |
| 19 | 7003.56 | Andrew LaPrise | CS302, Fall, 2011 |
| 20 | 6100.28 | Chris Nagy | CS302, Fall, 2015 |
| 21 | 5467.56 | Tyler Marshall | CS302, Fall, 2013 |
| 22 | 5262.80 | Harry Channing | CS302, Fall, 2018 |
| 23 | 5116.13 | Kyle Bashour | CS302, Fall, 2014 |
| 24 | 4808.03 | Matt Baumgartner | CS302, Fall, 2010 |
| 25 | 4586.51 | Jeramy Harrison | CS302, Fall, 2013 |
| 26 | 4531.96 | Philip Hicks | CS302, Spring, 2019 |
| 27 | 4057.08 | Phillip McKnight | CS302, Fall, 2015 |
| 28 | 3882.53 | Pranshu Bansal | CS302, Fall, 2013 |
| 29 | 3882.28 | Kemal Fidan | CS302, Fall, 2018 |
| 30 | 3852.87 | Yaohung Tsai | CS302, Fall, 2015 |
| 31 | 3849.24 | Chris Richardson | CS302, Fall, 2010 |
| 32 | 3809.41 | Arthur Vidineyev | CS302, Fall, 2015 |
| 33 | 3588.35 | Kevin Dunn | CS302, Fall, 2014 |
| 34 | 3464.83 | Patrick Slavick | CS302, Fall, 2012 |
| 35 | 3460.00 | Brandan Roachell | CS302, Fall, 2020 |
| 36 | 3436.21 | sb-play3 | CS140, Fall, 2007 |
| 37 | 3400.50 | Kody Bloodworth | CS302, Fall, 2018 |
| 38 | 3080.15 | Andrew Messing | CS302, Fall, 2013 |
| 39 | 2903.38 | Adam LaClair | CS302, Fall, 2013 |
| 40 | 2616.00 | Rus Refait | CS302, Spring, 2020 |
| 41 | 2555.36 | Mohammad Fathi | CS302, Fall, 2014 |
| 42 | 2532.89 | Trevor Sharpe | CS302, Fall, 2015 |
| 43 | 2521.44 | Justus Camp | CS302, Fall, 2018 |
| 44 | 2354.00 | Zach Deguira | CS302, Fall, 2020 |
| 45 | 2335.88 | Mark Clark | CS302, Fall, 2012 |
| 46 | 2307.16 | John Burnum | CS302, Fall, 2012 |
| 47 | 2205.17 | Shawn Cox | CS302, Fall, 2011 |
| 48 | 2163.70 | Alex Wetherington | CS302, Fall, 2011 |
| 49 | 2134.99 | Julian Kohann | CS302, Fall, 2013 |
| 50 | 2011.38 | Wells Phillip | CS302, Fall, 2015 |
| 51 | 1919.72 | Ravi Patel | CS302, Spring, 2019 |
| 52 | 1854.00 | Sam Aba | CS302, Spring, 2020 |
| 53 | 1778.83 | Keith Clinart | CS302, Fall, 2011 |
| 54 | 1740.19 | Luke Bechtel | CS302, Fall, 2014 |
| 55 | 1634.49 | William Brummette | CS302, Fall, 2013 |
| 56 | 1602.83 | Forrest Sable | CS302, Fall, 2014 |
| 57 | 1498.87 | Tom Hills | CS302, Spring, 2019 |
| 58 | 1470.84 | Christopher Tester | CS302, Fall, 2014 |

| | | | |
|---|---|---|---|
| 59 | 1446.00 | Noah Burgin | CS302, Spring, 2020 |
| 60 | 1433.48 | Xiao Zhou | CS302, Fall, 2015 |
| 61 | 1430.54 | Jonathan Burns | CS302, Fall, 2018 |
| 62 | 1340.32 | John Murray | CS302, Fall, 2012 |
| 63 | 1329.34 | Benjamin Brock | CS302, Fall, 2013 |
| 64 | 1257.56 | Dylan Lee | CS302, Fall, 2018 |
| 65 | 1202.06 | Bandara | CS302, Fall, 2014 |
| 66 | 1149.80 | Will Houston | CS302, Fall, 2010 |
| 67 | 1119.85 | Kevin Chiang | CS302, Fall, 2014 |
| 68 | 1096.48 | Daniel Cash | CS302, Fall, 2011 |
| 69 | 1059.91 | Kaleb McClure | CS302, Fall, 2013 |
| 70 | 1058.26 | sb-play2 | CS140, Fall, 2007 |
| 71 | 1029.63 | Lydia San George | CS302, Fall, 2018 |
| 72 | 1019.72 | Justin Langston | CS302, Spring, 2019 |
| 73 | 972.36 | Erik Rutledge | CS302, Fall, 2013 |
| 74 | 959.79 | Daniel Nichols | CS302, Fall, 2018 |
| 75 | 917.92 | Vasu Kalaria | CS302, Fall, 2015 |
| 76 | 908.09 | Chris Rains | CS302, Fall, 2012 |
| 77 | 875.44 | Allen McBride | CS302, Fall, 2012 |
| 78 | 856.00 | Tim Krenz | CS302, Spring, 2019 |
| 79 | 840.94 | Spencer Howell | CS302, Fall, 2018 |
| 80 | 830.79 | David Cunningham | CS302, Fall, 2014 |
| 81 | 826.00 | Kincaid Mcgee | CS302, Fall, 2020 |
| 82 | 810.17 | Collin Bell | CS302, Fall, 2012 |
| 83 | 763.58 | Jacob Lambert | CS302, Fall, 2013 |
| 84 | 703.67 | Scott Marcus | CS302, Fall, 2015 |
| 85 | 703.00 | Don Lopez | CS140, Fall, 2007 |
| 86 | 700.90 | Tony Abston | CS302, Fall, 2015 |
| 87 | 682.56 | Jackson Collier | CS302, Fall, 2014 |
| 88 | 681.00 | Holland Johnson | CS302, Fall, 2020 |
| 89 | 677.83 | KC Bentjen | CS302, Fall, 2011 |
| 90 | 677.74 | Andrew Artates | CS302, Spring, 2019 |
| 91 | 665.60 | Joshua Clark | CS302, Fall, 2012 |
| 92 | 659.96 | Warren Dewit | CS302, Fall, 2010 |
| 93 | 654.67 | Coburn Brandon | CS302, Fall, 2015 |
| 94 | 650.98 | Joaquin Bujalance | CS140, Fall, 2007 |
| 95 | 630.73 | Dylan Devries | CS302, Fall, 2018 |
| 96 | 630.10 | Winston Boyd | CS302, Fall, 2018 |
| 97 | 626.62 | Elliot Greenlee | CS302, Fall, 2014 |
| 98 | 594.02 | James Tucker | CS302, Fall, 2015 |
| 99 | 586.71 | Jonathan Ting | CS302, Spring, 2019 |
| 100 | 586.56 | Andrew Berger | CS302, Spring, 2019 |
| 101 | 571.02 | Rocco Febbo | CS302, Fall, 2018 |
| 102 | 554.94 | Jared Burris | CS302, Fall, 2015 |
| 103 | 539.00 | Braden Butler | CS302, Fall, 2020 |
| 104 | 508.04 | Victoria Florence | CS302, Fall, 2015 |