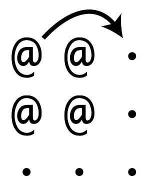
15-122: Principles of Imperative Computation, Spring 2013

Homework 5 Programming: Peglab



Due: Tuesday, March 26, 2013 by 23:59

For the programming portion of this week's homework, you will implement a program to solve peg solitaire puzzles. You have to complete one principal implementation and several small interface files.

- `peg1.c0` (described in Section 1)
- `peg2.c0` (described in Section 2)
- `peg3.c0` (described in Section 3)
- `peg-stacks.c0` (described in Section 1),
- `peg-ht.c0` (described in Section 3)

You should submit these files electronically by 11:59 pm on the due date. Detailed submission instructions can be found below.

- Please read all parts of the assignment carefully.
- Frequently and incrementally test your code. You can solve small boards without backtracking and medium-size boards without hash tables. Take advantage of this to catch conceptual errors early.

The starter code does *not* compile because it is missing the client-side implementations for stacks and hash tables as well as the function to solve a peg solitaire puzzle.

Modification to academic integrity and Piazza policy: The academic integrity policy for this course does not allow you to view other people's C0 code or share your C0 code with others. However, you may share peg solitaire boards that you find useful for testing, draw them on whiteboards, place them in public Piazza posts, whatever.

Questions on Piazza that include Autolab error messages and ask "is there any test board that might help me figure out the problem" are fine, but will *not* be addressed by course staff. Tag these posts on Piazza by writing `#needaboard` in the post. Questions on Piazza about specific code problems will only be addressed by course staff if they reference a test board that demonstrates the problem. (Conceptual questions and clarifications are still fine on Piazza.)

Assignment: Peg Solitaire (25 points in total)

Starter code. Download the file `hw5-handout.tgz` from the course website. When you untar it, you will find several C0 files and a `lib/` directory with some provided C0 libraries.

You should not modify or submit the library code in the `lib/` directory, nor should you rely on the internals of these implementations. When we are testing your code we will use our own implementations of these libraries with the same interface, but possibly different internals.

Compiling and running. You will compile and run your code using the standard C0 tools. You should compile and run the code with

```
% cc0 -d -o peg1 peg1.c0 peg-test.c0 peg-main.c0
% ./peg1 small.txt
```

on small examples (for Task 1, don't forget `-d`), and with

```
% cc0 -r unsafe -c-O2 -o peg<n> peg<n>.c0 peg-test.c0 peg-main.c0
% ./peg<n> english.txt
% ./peg<n> french1.txt
```

for Tasks 2 and 3 with `<n>` = 2 and 3. Removing `-d` runs the code without dynamic checks. The option `-r unsafe` turns off array bounds checks (which can cause your code to do arbitrary bad things instead of causing a segfault). The option `-c-O2`, with the letter `O` not the number `0`, tells the `gcc` compiler to do everything it can to make your code fast.

Submitting. Once you've completed some files, you can submit them to Autolab. There are two ways to do this:

From the terminal on Andrew Linux (via cluster or ssh) type:

```
% handin hw5 peg1.c0 peg2.c0 peg3.c0 peg-stacks.c0 peg-ht.c0
```

Your score will then be available on the Autolab website.

Your files can also be submitted to the web interface of Autolab. To do so, please `tar` them, for example:

```
% tar -czvf sol.tgz peg1.c0 peg2.c0 peg3.c0 peg-stacks.c0 peg-ht.c0
```

Then, go to `https://autolab.cs.cmu.edu/15122-s13` and submit them as your solution to homework 5 (peglab).

You may initially submit your assignment up to 25 times, though we may raise this cap. When we grade your assignment, we will consider the most recent version submitted before the due date. If you get any errors while trying to submit your code, you should contact the course staff immediately.

Annotations. Be sure to include appropriate `//@requires`, `//@ensures`, `//@assert`, and `//@loop_invariant` annotations in your program. You should provide loop invariants and any assertions that you use to check your reasoning. If you write any auxiliary functions, include precise and appropriate pre- and postconditions.

You should write these as you are writing the code rather than after you're done: documenting your code as you go along will help you reason about what it should be doing, and thus help you write code that is both clearer and more correct. **Annotations are part of your score for the programming problems; you will not receive maximum credit if your annotations are weak or missing.**

Unit testing. You should write unit tests for your code. For this assignment, unit tests will not be graded, but they will help you check the correctness of your code, pinpoint the location of bugs, and save you hours of frustration.

Style. Strive to write code with *good style*: indent every line of a block to the same level, use descriptive variable names, keep lines to 80 characters or fewer, document your code with useful comments and contracts, etc. If you find yourself writing the same code over and over, you should write a separate function to handle that computation and call it whenever you need it. We will read your code when we grade it, and good style is sure to earn our good graces. Feel free to ask on Piazza if you're unsure of what constitutes good style.

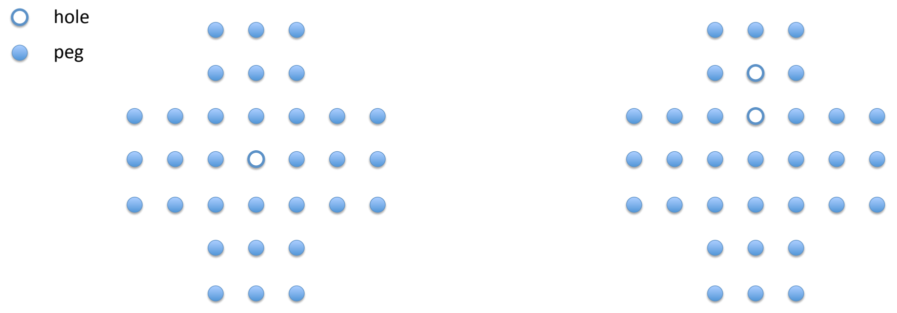
Task 0 (10 points, sort of) At this point in the semester, we expect you to be writing with good style; we have given you feedback on style for multiple assignments and we have shown you many examples of code with reasonable style. This assignment will initially have all 30 points assigned through the autograder, but up to 10 points may be deducted for style issues.

One reason for this is that your code is liable to get pretty messy as you transform `peg1.c0` into `peg2.c0` and `peg2.c0` into `peg3.c0`. In order to succeed at this assignment, you will want to *refactor* your code – notice that it has gotten complicated and restructure it in a way that suits the task better. Refactoring a bit will probably help you write and debug the assignment, and one purpose of style grading this assignment is to help persuade you to do this.

Peg Solitaire

Peg solitaire is a one-player board game with the goal of removing all pegs except one from a board, starting with some initial board configuration consisting of holes, some of which are filled with pegs.

A move is always a vertical or horizontal jump of one peg over another, removing the peg that was jumped over from the board. For example, in the initial configuration of the standard English board on the left, there are 4 possible moves, all ending in the center. The peg arriving from the top leads to the configuration on the right.



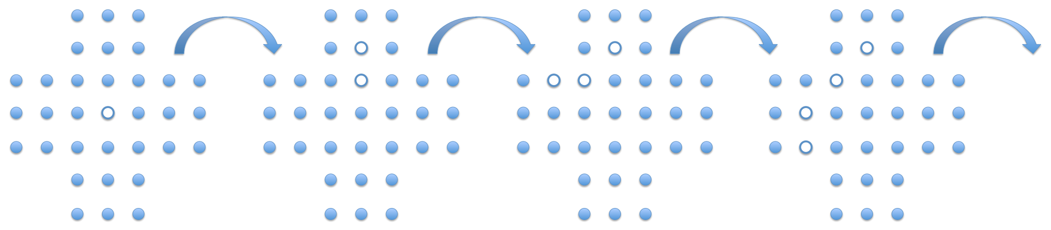
In the position on the right we have now have just 3 possible moves.

The goal of the game is to be left with just one peg. On the standard English board we start with 32 pegs, so any solution will require exactly 31 moves (each jump removes exactly one peg from the board). In some variations of the game we also stipulate where the final peg should come to rest, but in this assignment just reaching a board with a single peg is the only goal.

See `http://en.wikipedia.org/wiki/Peg_solitaire` for more on peg solitaire.

Games computers play

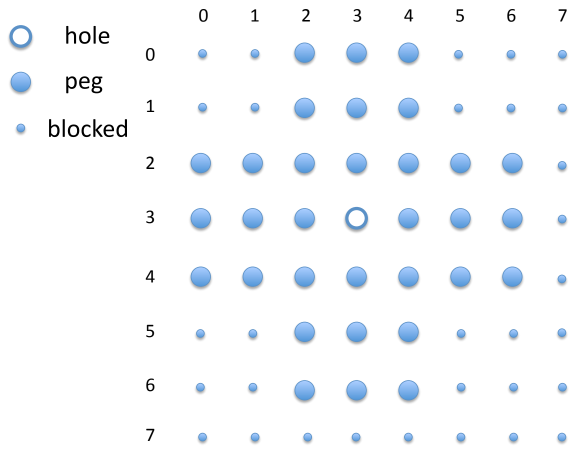
The structure of a peg solitaire is well suited to a solution involving recursion. If we want to play peg solitaire on a board with 32 pegs, then we can enumerate all the valid moves that can be made on that board. Then, after we make one of those moves, we're playing peg solitaire again – only this time on a board with 31 pegs.



If we ever reach a point where we're playing peg solitaire on a board with one peg, it is a very easy game – we've won! A *losing* game of peg solitaire is one where there are no valid moves and more than one peg. An *unsolvable* game of peg solitaire is one where every series of valid moves leads to a losing game.

Representation of the Board

In this programming assignment you will have to make some choices regarding the representation of moves, hash table keys, etc., but we specify the representation of boards. A board is always an array of size $8 \times 8 = 64$ integers, where -1 means the location is blocked (i.e. has no hole and cannot accept a peg), 0 means the location is a hole without a peg, and 1 means the location is a hole occupied by a peg. The board is laid out starting at the top left.



The notation we use for a location on the board is *row* : *col*, starting both *row* and *col* at 0 in the upper left-hand corner. For example, the first move illustrated in the example on the previous page would be from `1:3` to `3:3`. Remember that we represent the board as a one-dimensional array, so `1:3` corresponds to array index $1 * 8 + 3 = 11$ in the board array and `3:3` corresponds to array index $3 * 8 + 3 = 27$ in the board array. Please refer to the definitions in the file `peg-util.c0`, including

```
typedef int[] board;
```

This file also contains functions to read board configurations from a file and print board configurations.

Representation of a Move

It is up to you how to represent moves.

```
typedef ________ move;
```

Some suggestions for the type definition are triples of integers (representing the three board indices involved), pairs of integers (the first and last index of the move), or the row and column values of the peg before and after the move. These could be represented by structs, or could be compressed into something like a single `int`. This involves a tradeoff between compactness and ease or speed of determining the possible moves on a given board. We do **not** combine multiple jumps into a single move.

The testing harness will treat the type `move` as abstract. This means it only uses the following four functions to extract information from a given move m to check whether the move is valid:

```
    int row_start(move m);
    int col_start(move m);
    int row_end(move m);
    int col_end(move m);
```

You can find the testing harness in the files `peg-test.c0` and `peg-main.c0`. You cannot change these files, but you can inspect this code (and the code in `peg-util.c0`) and reuse anything you find useful.

Representation of a Solution

A *solution* to peg solitaire from a given initial configuration consists of a *stack of moves*. The top of the stack should contain the first move, the next element the second move, etc. The function

```
    bool verify_solution(board B, stack S);
```

(provided in the file `peg-test.c0`) verifies that the stack S is a valid solution for the initial configuration given by board B.

Because the solution is a stack of moves, your code needs to define the type of stack element, called `item` in the file `peg-stacks.c0`, before the stack library is used. You can see that from the beginning of the file `peg5.c0` which contains the lines

```
    #use "peg-stacks.c0"
    #use "lib/stacks.c0"
```

A similar remark applies to the hash table (for Task 3), which is included into your code with the lines

```
    #use "peg-ht.c0"
    #use "lib/ht.c0"
```

1 Determinstic Peg Solitaire

For the first task you should assume that from the given initial configuration there is exactly one possible move, and in the configuration resulting from this move there is again exactly one possible move, and so on, until you either reach a winning configuration (i.e. one peg left) or a losing configuration (more than one peg left, but no moves possible).

The point of this task is to make sure that you can correctly determine possible moves, correctly generate solutions, and that you are providing appropriate implementations of the specified interfaces.

Task 1 (10 pts) *Based on the problem description given, determine how you want to represent a move. In the file* `peg1.c0`*, define the type* `move`

```
typedef _____ move;
```

Also, in the file `peg1.c0`*, define the functions*

```
int row_start(move m);
int col_start(move m);
int row_end(move m);
int col_end(move m);
```

to extract row and column information from a move where start refers to the starting position of the jumping peg, and end to its ending position.

Recall that a solution is represented as a stack of moves. In this assignment, our `stack` *is implemented as a linked list of* `item`*s. In the file* `peg-stacks.c0` *define the type* `item`

```
typedef _____ item;
```

so `item` *is consistent with your type* `move`*. This will allow the representation of a solution as a stack of moves. You won't be able to just write* `typedef move item` *because the type* `move` *will not be defined until later in the compilation sequence. Typical answers might be* `struct move_info *` *(for a simple encoding of a move as a struct, whose actual definition is in* `peg1.c0`*) or* `int` *(for a more compact encoding).*

In the file `peg1.c0`*, define the function*

```
int peg_solve(board B, stack S);
```

to play peg solitaire for the given board. The integer that `peg_solve` *returns is the number of pegs left at the end of the game.*

- *If* `peg_solve` *returns* `1`*, then our algorithm played peg solitaire and won. In this case,* `S` *should contain the solution in the form of a stack of moves.*
- *If* `peg_solve` *returns a number greater than* `1`*, then our algorithm played peg solitaire and lost when the board still had* `\result` *pegs left on the board. In this case,* `S` *can contain anything.*

In order to write the `peg_solve` function, you should write a **recursive** helper function `solve` that is called from `peg_solve`. This helper function should take the board, the current stack of moves for a solution, and the number of pegs remaining on the board as parameters. It should return `1` if there is a solution for the given board, the current stack of moves and the given number of pegs remaining. THINK: What is the base case here for this recursive function? When do you know you have a solution for the current board?

One strategy for this helper function is to consider the current board and push all possible first moves onto another stack (possibly using an auxiliary function). It then processes the top move of the additional stack of first moves (we ignore all of the others) and then tries to see if there is a solution to the resulting board recursively. If there is a solution to the resulting board, there is a solution to the current board. In case there is a solution make sure you push the moves of the solution onto the solution stack in the correct order.

Testing

Compile your solution with

```
% cc0 -d -o peg1 peg1.c0 peg-test.c0 peg-main.c0
```

and test it on boards which satisfy the stated determinism requirements with

```
% ./peg1 yourtest.txt
```

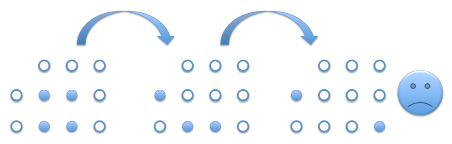
We have only provided one trivial test in the file `german.txt`.

If you would like to use other libraries, either from the course so far or from the website, please include the appropriate files when you hand in your code. Any `#use` directives must come below the indicated line in the files you hand in, otherwise you run the risk of your files not compiling with our test harness on the Autolab server.

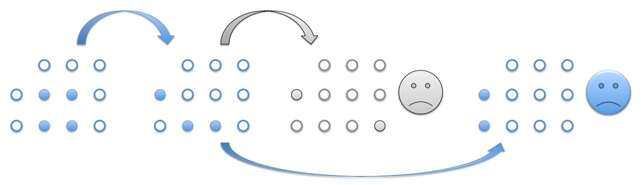
Our testing harness will only test your Task 1 code on peg solitare boards that are deterministic (boards always have either 1 or 0 moves), but you may want to run your solution on more complicated boards. If you do, you should expect your code to sometimes give up too early, failing to find a solution even when one exists.

2 Solving Peg Solitaire

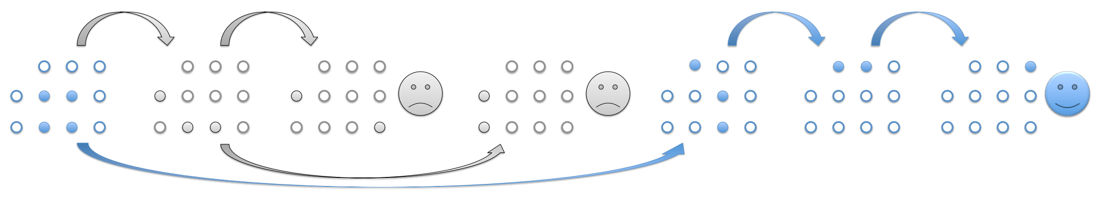
In the previous task, we were content to *play* peg solitaire: with the board in its current state, we generated a stack of all possible immediate moves we could make from the current board, chose one, and played peg solitaire on the resulting board (recursively). Using this approach, if we hit a dead-end in our search for the solution, we're out of luck – we would return the number of pegs left on the board to indicate that we did not win the game. (The first move below is `1:2 to 1:0`, and the second move is `2:1 to 2:3`.)



We can do better with a strategy called *backtracking*. Right now we know that the third peg solitaire board above, with two pegs, is a losing game, which means it is also unsolvable. But we don't know that the *second* peg solitaire board, the one with three pegs, is unsolvable, because there's another valid move we could make – `2:2 to 2:0`. So we backtrack to this old board and try working forward from there.



Unfortunately, this second move also leads to a losing board. This means that the board with three pegs above *is* unsolvable – all the moves starting from that board lead to losing boards. We have to backtrack further, to the board with four pegs, and pick a different way forward. If we pick the valid move `2:1 to 0:1`, we will succeed.



In this task you will generalize the implementation from Task 1 so that it can solve boards that require backtracking for their solution.

Using the process of backtracking, we start with a board in some current state. We try a potential first move on a board, changing the board to a new state. If that move does not work (if the resulting board is unsolvable), we return back to the current state and try another potential first move. We repeat this process as long as a first move does not lead to a solution and there are still potential first moves left to try. If none of the potential first moves lead to a solution, then we know the current board is unsolvable. We return back to the previous state of the board and try any remaining first moves still available, and so on. This backtracking process is also recursive. (Do you see why?)

To summarize, this task differs from the previous task in that your new code will play peg solitaire more like a computer, solving the whole problem rather than trying to play the game in one particular way. If it hits a dead-end with no subsequent moves, it undoes the previous move and tries another move. It that fails, it undoes that move and continues this process.

Task 2 (13 pts) *Copy your (hopefully working) code from* `peg1.c0` *to* `peg2.c0`.

Extend the function `peg_solve` *so that it can handle boards that potentially require backtracking. We will test your code with simple problems requiring backtracking, with the most complex one being* `english.txt` *(the standard English peg solitaire board and initial position).*

If your `peg_solve` *function returns a number greater than* `1`*, indicating that the peg solitare board was entirely unsolvable, then the returned integer should be the* smallest *number of pegs on any board you encountered. The stack* `S` *still only needs to contain a valid set of moves when the result is* `1`*.*

NOTE: For this task *do not use hash tables.* This will allow you to explore how complex the problems can be using this technique before you need a data structure like a hash table to reduce the expanding search space.

Testing

We will test your code with

```
% cc0 -o peg2 peg2.c0 peg-test.c0 peg-main.c0
% ./peg2 mediumboard.txt
```

using several boards of increasing difficulty, culminating in `english.txt`, the standard English board and starting configuration. Depending on the order in which you pick moves, your code may be able to handle `english.txt`, even without the use of a hash table.

We will set timeouts on the Autolab server so that if your code it too slow it may fail some of the more difficult tests. So you should pay some attention to efficiency of your code. If your Task 2 solution cannot solve the English peg solitare board without running out of time, you can still get full points if your Task 3 solution can solve it. You may want to look at the notes at the end of this handout on move selection strategies.

Some Advice

Your solution should probably reuse most of the code from Task 1. You will mainly have to deal with the problem that applying a move *modifies* the board. When you backtrack by returning a number greater than `1` you must make sure to *undo* your change to the board. This can be stated as an invariant of the `solve` function: before it returns the board must be restored to the configuration it was in when it was called.

Efficiency is becoming a minor factor at this point. For example, it is probably worth considering how to *efficiently* check that you have reached a winning position since you may have to do this many times. We recommend sticking with the so-called *brute-force* search rather than trying to rank the possible moves based on their promise.

3 Hash Tables

The basic problem with the backtracking search from Task 2 is that it may visit the same unsolvable peg solitaire boards many, many times. In this task you will add the use of hash tables to the code you used in Task 2. The idea is to save board configurations that have been determined to be unsolvable into a hash table. Then, when trying to solve a position we first check if the board has already been recorded as having no solution and, if so, we immediately return the answer associated with that hash table.

Two crucial factors will determine the efficiency of your implementation: your choice of keys and your choice of hash function.

With respect to keys, we recommend compressing your board into a compact representation for the hash table containing enough information so that two boards in a given problem have the same key if and only if they represent the same position. The `key_equal` function on this representation should be fast. Note that you could not use the board array itself anyway without copying, because then your `solve` function would change the contents of the hash table. If your key is a copy of the whole length 64 array of integers, then your implementation will almost certainly still be too slow.

Regarding hash functions: it will be important to write a hash function for your compact board representation that avoids collisions. We recommend the Wikipedia article on universal hash functions at `http://en.wikipedia.org/wiki/Universal_hash_function`. Also, feel free to use the `ht_stats` function in `lib/ht.c0` to see what the distribution of data in your hash table is like.

Task 3 (7 pts) *Copy your (hopefully working) code from* `peg2.c0` *to* `peg3.c0`.

In the file `peg-ht.c0`*, provide the client-side implementations of the following types and functions that are necessary for the hash table to work correctly.*

```
typedef _____* htelem;
typedef _____ key;
```

```
int hash(key k, int m);
bool key_equal(key k1, key k2);
key htelem_key(htelem e);
```

In the file `peg3.c0`*, extend your code to take advantage of the hash table to reduce the search space for a solution.*

Testing

We will test your code with

```
cc0 -o peg3 peg3.c0 peg-test.c0 peg-main.c0
./peg3 largeboard.txt
```

using several boards of increasing difficulties, culminating in the files `english.txt` and `french*.txt` (a French version of peg solitaire).

As in Task 2, we will set some timeouts to prevent your code from consuming unbounded system resources and verifying that your code is reasonably efficient.

Some Advice

Efficiency is becoming a major factor at this point. There are three main factors that may influence efficiency. Move selection remains an important issue. The efficiency of your hash function, the compactness of your hash key representation, the size of your hash table, and the ability of your hash function to avoid collisions are also important factors. Some of these issues are discussed in `performance-debugging.txt`.

Going Further

Our solution is entirely *brute force*, that is, it does not employ any heuristic ordering among the possible moves. You might consider adding such heuristics to select the most promising moves first once you get this assignment done. All other aspects of the solution should be the same as in the required assignment. If you go beyond the required assignment, please include a `HEURISTIC.txt` file that explains your strategy.

A Move selection

The order in which you consider moves will not make a big difference to how fast you solve unsolvable boards, but it can make a big difference how fast you find the solution to a board that has a solution. Three points on this assignment are for your ability to solve the English pegboard with either Task 2 or Task 3, and to do so you may need to rearrange the order in which you put moves on the stack. If you pick moves by iterating over the board spaces, there are three obvious options:

- Every time you find a peg, see if it can jump in every direction (up, down, left, right).
- Every time you find a peg, see if it can be *jumped over* in every direction.
- Every time you find a hole, see if it can be jumped *into* from every direction.

We recommend you try the first of these three strategies and then experiment with different orderings of directions (up-down-left-right versus up-right-down-left and so on) to find one that works well on the English board. With the right move selection strategy, you can find a solution to `english.txt` in Task 3 very quickly (there can be less than 1100 boards in your hash table after solving the English board).

However, any general solution that solves the English board fast enough will get credit.