Assignment 2 Games

Due Oct 24 by 12p.m. **Points** 100

Table of Contents

Warning

We know that solutions to this or related problems may exist online. *Do not use these solutions, as this would be plagiarism.* To earn marks on this assignment, you must develop your own solutions.

Also, please consider the following important points:

- *Do NOT add any non-standard imports*. All imports already in the starter code must remain. When in doubt, post a question on Piazza.
- *Do NOT modify any provided code*. Modifying the provided code may cause you to *fail* all the tests.
- Submit your program on MarkUs and run the provided tests *well before* the deadline. The fact that your code runs on your own system but not on MarkUs is *NOT* a legitimate reason for a regrade request.
- The provided tests on MarkUs are an *extremely basic* sanity check. Passing the provided tests only means that your program runs without errors; it does *NOT* mean that you will receive full marks on the assignment.

The Mancala Game

The goal of this assignment will be to implement a program to play the Mancala game, shown in the picture below.

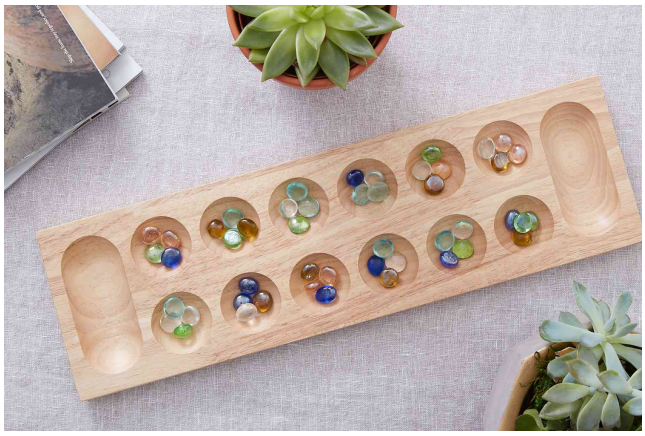
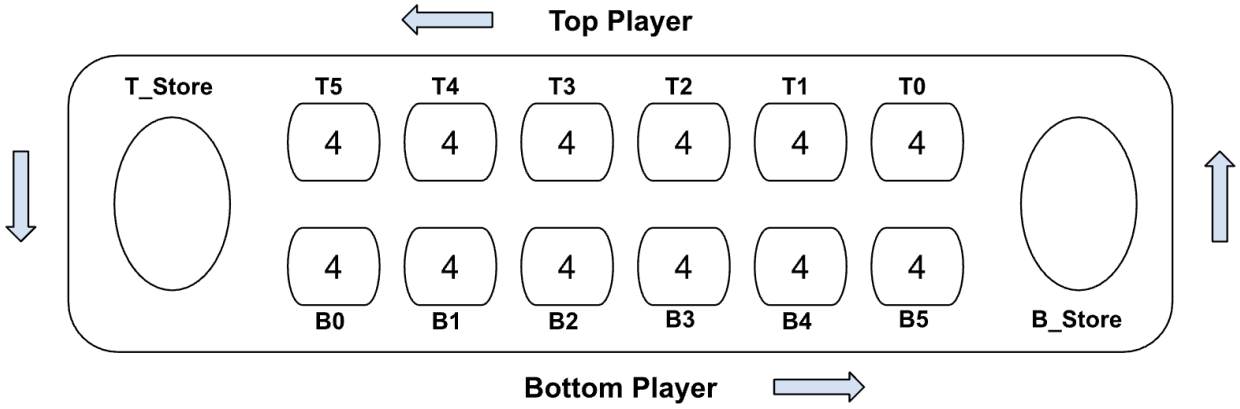


Figure: A picture of the Mancala game

Mancala is a board game played in various cultures for thousands of years.

Setup: Mancala is a two-player game. The board has two rows of six pockets and a Mancala store on each side. Start with four stones in each pocket.



Objective: to collect more stones in their Mancala store than their opponent.

Game Play:

- The game begins with one player picking up all the stones in any one non-empty pocket on their side.
- Moving counter-clockwise, the player deposits one stone in each pocket until the stones run out.
- If the player runs into their own Mancala store, deposit one stone in it.
- If the player runs into the opponent's Mancala store, skip it and continue moving to the next pocket.
- Capture: If the last stone drops in an empty pocket on the player's side, take all the stones in the directly opposite pocket (belonging to your opponent) and put them in the player's store.
- The game ends when all the pockets on one side are empty.
- The player who still has stones on their side when the game ends puts all the stones in their Mancala store.
- The winner of the game is the player with more stones in their Mancala store than the opponent.

Our version of Mancala **differs** from the common version of Mancala in two ways.

1. In some versions of Mancala, if the player places the last stone in their store, they get a free turn to play again. Our version of Mancala removes this rule. In each turn, the player can only play once.
2. In some versions of Mancala, after a player places the last stone in an empty pocket on their side, the player puts it in their store. In our version, this last stone will stay.

An example of a player's depositing a stone in their store

The two figures below show an example of the bottom player taking a move and depositing a store in their store. The second figure shows the results of the bottom player taking the six stones from B4 and depositing them counterclockwise.

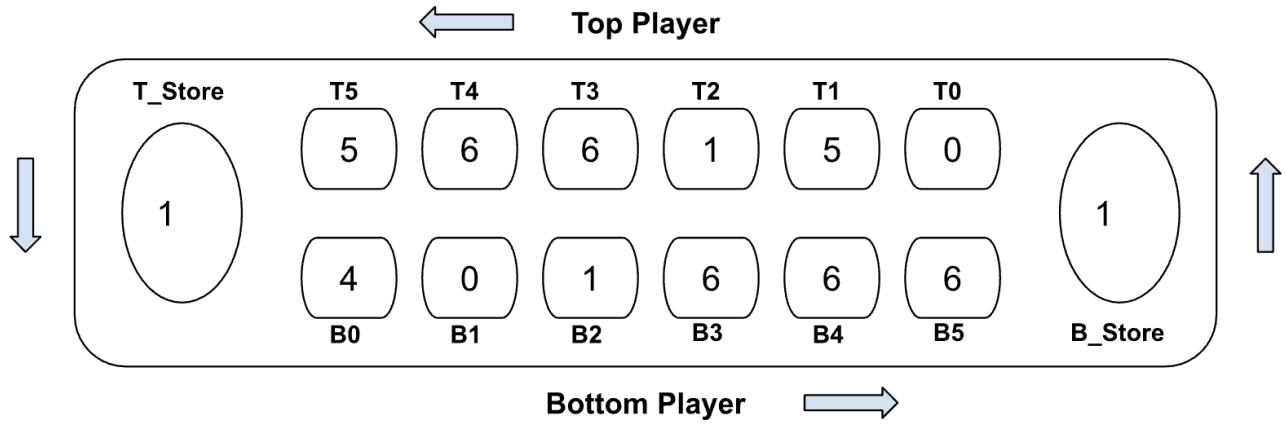


Figure: Board before the bottom player's move

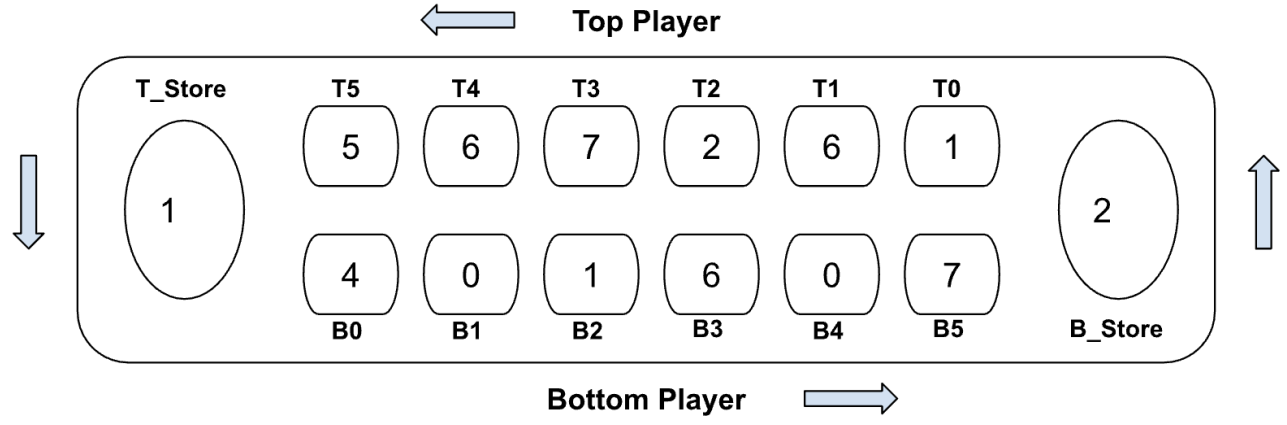


Figure: Board after the bottom player's move. The bottom player took six stones from B4 and deposited them counterclockwise.

An example of a move skipping your opponent’s store

The figures below show an example of a move skipping the opponent's store. The bottom player takes eight stones in B5 and deposits them counterclockwise, skipping the top player's store.

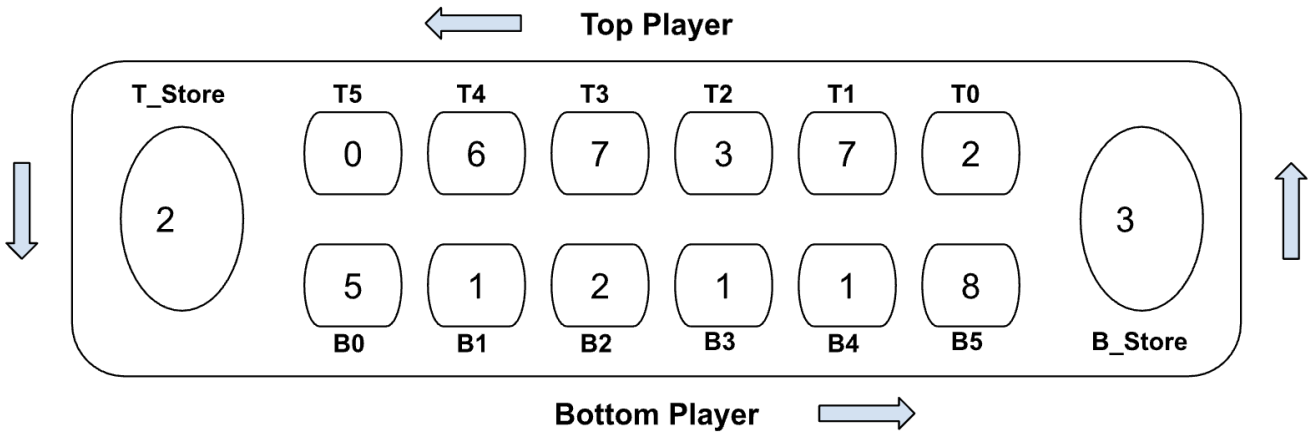


Figure: Board before the bottom player's move

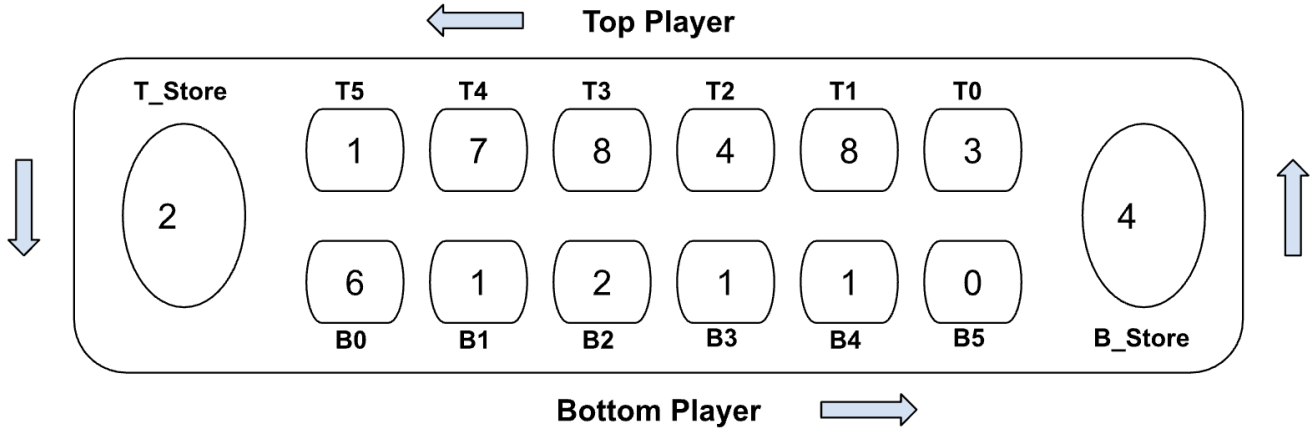


Figure: Board after a move by the bottom player.

The bottom player picked up eight stones in B5 and deposited them counterclockwise, skipping the top player's store.

An example of a capture

The figures below show the before and after boards for a Capture. The bottom player took the four stones in B0, deposited them counterclockwise, and captured the six stones in T1. After this move, the bottom player’s store has eight stones.

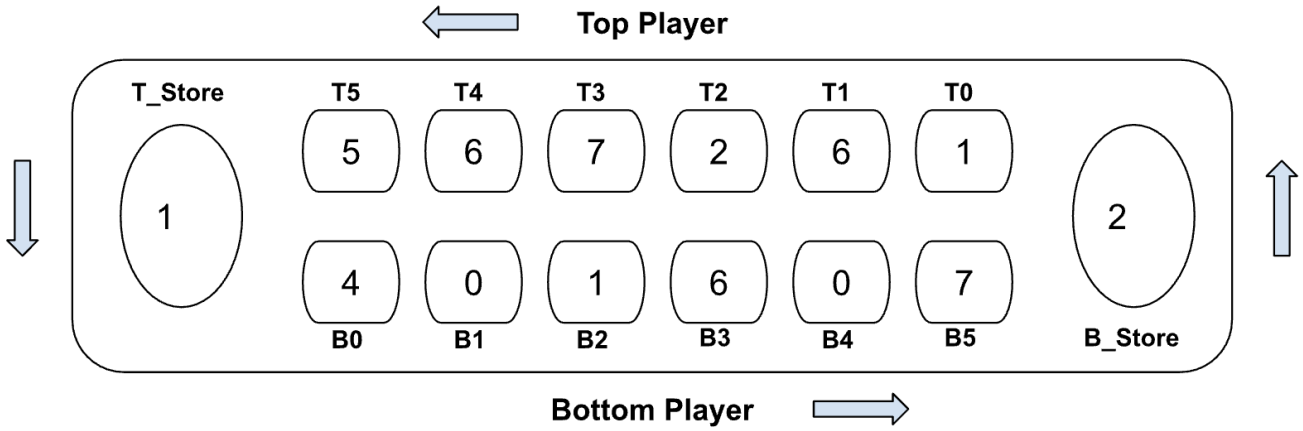


Figure: Board before a capture

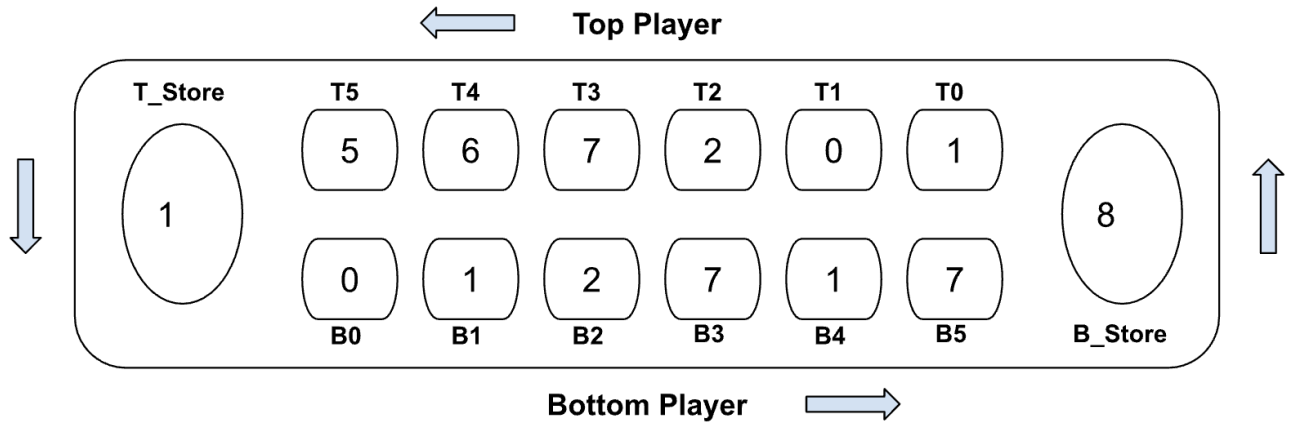


Figure: Board after a capture. The bottom player took the four stones in B0, deposited them counterclockwise, and captured the six stones in T1.

Your Tasks

You will implement two search algorithms: Depth-First Minimax and Alpha-Beta Pruning. In addition, you will implement several extensions, including depth limit and caching states.

You will also implement two heuristic functions: the basic heuristic function and an advanced heuristic function of your invention.

The Basic Heuristic Function

For a state and the current player, the basic heuristic function returns the number of stones in the current player's Mancala store *minus* the number of stones in the opponent's Mancala store.

Consider the state below. If the current player is the bottom player, the heuristic value is 8 - 1 = 7. If the current player is the top player, the heuristic value is 1 - 8 = -7.

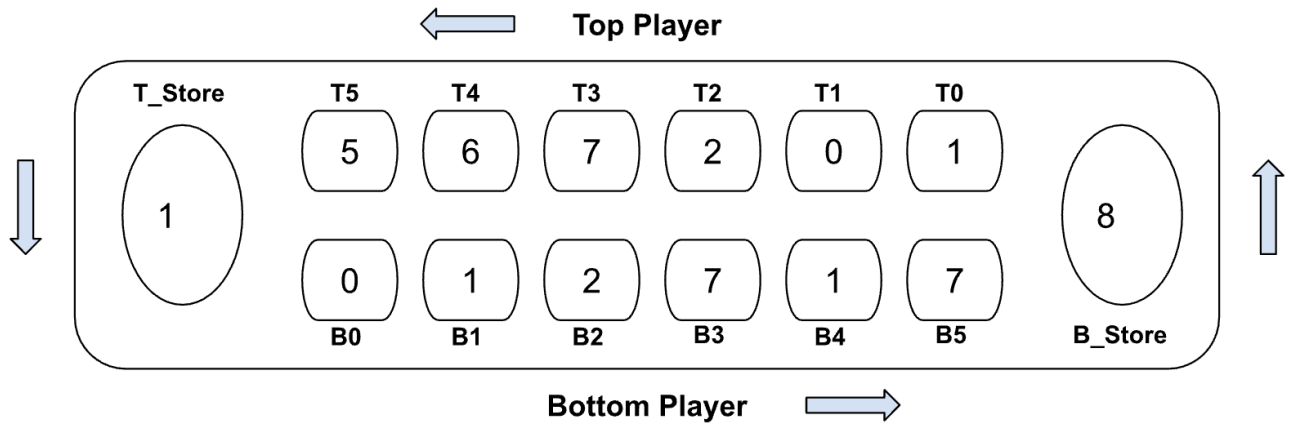


Figure: An example state to calculate the basic heuristic value.

An Advanced Heuristic Function

You will design and implement an advanced heuristic function. Here are some ideas to consider when designing your advanced heuristic function.

1. Consider the number of empty pockets on each side.
2. Consider the arrangement of empty pockets and possible captures.
3. Use a different strategy in the opening, mid-game and end-game.

We will test your advanced heuristic function by playing the two agents against each other.

- The top player is our alpha-beta pruning agent with our basic heuristic function.
- The bottom player is our alpha-beta pruning agent with your advanced heuristic function.

If we play these two agents against each other for many games, the bottom player (with your advanced heuristic) should win more than 50% of the time.

Starter Code

To support you in completing this assignment, we have provided several files to get you started:

- utils_starter.py,
- mancala_game.py,
- mancala_cmdline.py,
- mancala_gui.py,
- agent_random.py,
- agent_minimax_starter.py,
- agent_alphabeta_starter.py.

Below, we provide a brief explanation of the important elements. The files contain detailed documentation. Please read through the documentation carefully before getting started.

The focus of this assignment is to practice implementing adversarial search algorithms. Therefore, you don't have to write code to implement the Mancala game. We have provided code that implements the Mancala game in several files below.

- *mancala_game.py* implements the logic of the Mancala game.
- *mancala_cmdline.py* implements code to play the Mancala game via the command line. This file will be useful for testing your program through the command line.
- *mancala_gui.py* implements a graphical user interface for the Mancala game. This file is not required to complete the assignment. You may use it to debug your program using the GUI.

Heuristic Functions

You can start by implementing the basic heuristic function in *utils_starter.py*. Both minimax and alpha-beta pruning agents will use this heuristic function.

- *heuristic_basic* is the basic heuristic function.
- *heuristic_advanced* is the advanced heuristic function.

This file also defines *two important global constants.* The TOP player is represented by 0, and the BOTTOM player by 1. Please use these constants instead of hardcoding 0 and 1 throughout your program.

Minimax Search

Next, implement the minimax search algorithm in *agent_minimax_starter.py*. We have broken this down for you into three steps.

1. Implement the **basic** minimax search **without depth limit or caching**. You will need to complete *minimax_max_basic* and *minimax_min_basic*.
2. Implement the minimax search with a **depth limit**. You will need to complete *minimax_max_limit* and *minimax_min_limit*.
3. Implement the minimax search with a depth limit and **caching states**. You will need to complete *minimax_max_limit_caching* and *minimax_min_limit_caching*.

Note that most of your code in these three versions of the functions should be identical. We realize that this code duplication is not a good practice. However, we separated the code this way to test the three versions of the functions separately. This separation avoids potential issues if the added code for depth limit or caching breaks your basic minimax implementation.

We recommend completing the minimax search implementations through the process outlined below.

1. **Implement** the basic minimax search without depth limit or caching.
2. **Test** the basic minimax search thoroughly. You will need to construct starting boards that are quite close to an end game.
3. **Copy and paste** the basic minimax search code into the functions with the depth limit.
4. **Modify** the code to add the depth limit.

5. **Test** the minimax search with a depth limit thoroughly.
6. **Copy and paste** the code for the minimax search with depth limit to the functions with the depth limit and caching.
7. **Modify** the code to add caching.
8. **Test** the minimax search with a depth limit and caching thoroughly.

Be careful if you ever copy the code from the functions with a depth limit or caching to the earlier functions. Doing this may introduce bugs to the earlier functions.

Alpha-Beta Pruning

Next, implement the alpha-beta pruning search algorithm in *agent_alphabeta_starter.py*.

This file has the same structure as *agent_minimax_starter*.py.

1. Implement the basic alpha-beta pruning without depth limit or caching.
2. Implement alpha-beta pruning with a depth limit.
3. Implement alpha-beta pruning with a depth limit and caching states.

Similarly, we recommend that you test the simpler versions of the functions thoroughly before implementing the more complex versions of the functions.

The Random Agent

We implemented an agent that takes a random move at each turn in *agent_random.py*. You can use this agent to play the Mancala game through the GUI or the command line. This random agent will be useful before your minimax/alpha-beta agent is fully functional.

Running Your Program via the Command Line

We encourage you to play the Mancala game and test your program via the command line. To facilitate this process, we have implemented several command line arguments in *mancala_cmdline.py*.

Let me give you a few examples.

The following command allows you to play the Mancala game by controlling both players via the command line.

```
python mancala_cmdline.py --dimension 6
```

The following command allows you to play Mancala with two minimax agents using a depth limit of 8 and caching.

```
python mancala_cmdline.py -i test_board1.txt -t agent_minimax.py -b agent_minimax.py -l 8 --caching
```

The following command allows you to play Mancala with the random agent as the top player and the alpha-beta agent as the bottom player using the advanced heuristic and caching.

```
python mancala_cmdline.py -d 7 -t agent_random.py -b agent_alphabeta.py -ht 1 --caching
```

A summary of the command line parameters is provided below.

Command Line Parameters

Parameter	Type	Required?	Description
`--dimension`	int	optional	The number of pockets for each player. The default value is 6.
`--initialBoard`	str	optional	This is the name of a file containing the initial board of the game. If the initial board is not provided, then the initial board is the beginning of the game.
`--agentTop`	str	optional	A Python program to be the TOP player in the game. If not provided, the user must input each move.
`--agentBottom`	str	optional	A Python program to be the BOTTOM player in the game. If not provided, the user must input each move.
`--heuristicTop`	int	optional	The heuristic function for the TOP player. 0 = heuristic_basic, 1 = heuristic_advanced. The default value is 0.
`--heuristicBottom`	int	optional	The heuristic function for the BOTTOM player. 0 = heuristic_basic, 1 = heuristic_advanced. The default value is 0.
`--limit`	int	optional	Specify the depth limit as a non-negative integer. The default value is -1, meaning no limit is in use.
`--caching`	boolean	optional	This boolean value indicates whether state caching is included. The default value is

Input File Format

If the initial board is not provided, the game starts from the beginning. To test your program with a different initial board, you can provide the board in a text file. Below is an example of an input file.

```
4,5,4,4,6,4
4,4,3,4,4,4
2
0
```

The format of this input file is described below.

- The first two lines describe the number of stones in each player's pockets.

  - The first line describes the stones in the pockets of the top player. Note that the order of the pockets is from the perspective of the bottom player.
  - The second line describes the stones in the pockets of the bottom player.
- The next two lines describe the number of stones in each player's Mancala store.
  - The third line is the number of stones in the top player's Mancala store.
  - The fourth line is the number of stones in the bottom player's Mancala store.

While playing the game via the command line, the program will print out the current state of the game using the format below.

```
  4 5 4 4 6 4

2           0

  4 4 3 4 4 4
```

Submission and Grading Scheme

To earn marks for this assignment, you must submit completed **utils.py, agent_minimax.py, and agent_alphabeta.py** to the **A2** project on MarkUs. You will need to rename utils_starter.py -> utils.py, agent_minimax_starter.py -> agent_minimax.py, agent_alphabeta_starter.py -> agent_alphabeta.py. Please remember to update anywhere that imports utils_starter to utils.

Your final grade depends on the results of the automated tests on MarkUs only. We will not be inspecting your code manually. Specifically, your final grade depends on the results of the tests in the **A2** project on MarkUs. In the table below, we have outlined the number of tests, the types of tests and the marks assigned to each set of tests.

There are two types of tests: **public** and **hidden**. The public tests in A2 are available during the entire three-week period. You can run them as many times as you like until the deadline. The hidden tests will not be released to you. After the deadline, we will use the public and hidden tests to determine your assignment's final grade and release the results of the hidden tests on MarkUs.

Remember that the public tests are extremely basic sanity checks. They merely ensure that your program runs. Passing all the public tests does not mean you will get a good or perfect final mark on the assignment. By providing you with a small number of public tests, we want to encourage you to test your program yourself. You can either test your program via the command line or by writing your own unit tests.

Due to technical constraints on MarkUs, we need to specify a time limit for each test case on MarkUs. For this assignment, your program must terminate within **3 minutes** for each test case to earn marks.

MarkUs Test Availability and Grading Scheme

Category	Test Descriptions	A2 - Week 1 Oct 3 - 10	A2 - Week 2 Oct 10 - 17	A2 Oct 3 - 24	Marks
Basic Heuristic	heuristic_basic	1 public test		1 public test + 4 hidden tests = 5 tests	5
Minimax	minimax	1 public test		1 public test + 9 hidden tests = 10 tests	20
	minimax + depth limit		1 public test	1 public test + 9 hidden tests = 10 tests	20
	minimax + depth limit + caching			1 public test + 9 hidden tests = 10 tests	10
Alpha-Beta Pruning	alpha-beta pruning	1 public test		1 public test + 9 hidden tests = 10 tests	15
	alpha-beta pruning + depth limit		1 public test	1 public test + 9 hidden tests = 10 tests	15
	alpha-beta pruning + depth limit + caching			1 public test + 9 hidden tests = 10 tests	10
	our alpha-beta pruning agent + your advanced heuristic wins against our alpha-beta pruning agent + our basic heuristic at least 50% of the time			the two agents will play 20 games.	5

The Optional Early Feedback Deadlines

To encourage you to start working on the assignments early and tackle them incrementally, we have introduced *two optional early feedback deadlines* for each assignment. During the three weeks, we will provide some extra unit tests during the first two weeks. If you submit during either week, you can get additional feedback from these extra unit tests. Note that these extra unit tests will be *exclusively* available during their respective weeks and will no longer be available in the third week. These optional early feedback deadlines are not worth any marks.

We created these optional early feedback deadlines for the following reasons.

- *Encourage you to start the assignments early*: If you start early, you can take advantage of these extra unit tests, which won't be available later on.
- *Help you tackle the assignment incrementally*: We break down the assignment into smaller tasks so that the task for each week is more manageable.
- *Provide extra support*: The extra unit tests make it easier to debug and test your program by identifying problems in your program.
- *Avoid adding extra stress*: The early deadlines are not worth any marks. Therefore, if you are busy with other obligations and cannot take advantage of the early deadlines, you are not missing out on any marks, and you can still potentially get full marks on the assignments.

We have set up three projects on MarkUs: A2-Week1, A2-Week2 and A2. The first two projects (A2-Week1 and A2-Week2) contain the extra unit tests for the optional early deadlines. Please check the table above for the public tests available. Remember that the public tests in A2-Week1 and A2-Week2 are different from those in the A2 project on MarkUs.