Description of the Kalah game:

It's a board game meant for two players. Each player has 6 holes & one kalah store. At the beginning of the game, there are *k (here, k = 6)* stones in each hole except the kalah stores. A player has to get as many stones in his/her Kalah store in orde to win the game. The game terminates when all the holes of a player become empty or when one player gets more than *6k (here 36)* stones in his/her kalah store. A player is supposed to pick all the stones from one of his holes & place them in consecutive holes, his kalah & opponent's hole except the opponent's kalah in a counter clockwise fashion. There are several associated rules with the game which I will talk about in the heuristics function (utility value) definition.

Definition of my heuristic (utility value) function

My heuristic function is derived from the rules associated with the game.
The Game state is defined by:

1. a - AI's holes array
2. b - opponent's holes array
3. a_fin - AI's kalah stones count
4. b_fin - opponent's kalah stones count

```
    value = 0
    holes = [i for i in range(6)]
    # if opponent's kalah contains stones less than 12 (i.e. for the starting 1/3rd part of game, the AI (maximizing player) plays in a defensive mode)
    if state.b_fin < 12:
        value = state.a_fin * 0.75 - state.b_fin * 1
    # for the end 2/3rd part of game, the maximizing player (AI) goes all offensive
    else:
        value = state.a_fin * 1 - state.b_fin * 0.6

    # if last stone of maximizing player lands in his kalah, increase the value by 2
    for hole_num in holes:
        if (state.a[hole_num] == 6 - hole_num):
            value += 2
    # if last stone of minimizing player lands in his kalah, decrease the value by 2
    for hole_num in holes:
        if (state.b[hole_num] == 6 - hole_num):
```

```
            value -= 2

        # decrease the value by 10 if more than 4 of maximizing player's holes become empty
      if state.a.count(0) > 4:
          value -= 10
        # increase the value by 10 if more than 5 of minimizing player's holes become empty
      if state.b.count(0) > 5:
          value += 10

      # increase the value by the opposite hole stone count
      for hole_num in holes:
          extra_stones = state.b[5 - hole_num]
          value += extra_stones
          value += state.a[hole_num]

        # decrease the value by your own hole's stone count since the opponent will try to make moves in order to capture them
      for hole_num in holes:
          extra_stones = state.a[5 - hole_num]
          value -= extra_stones
          value -= state.b[hole_num]
```

Typically in order to maximize AI's kalah stones count & win, one would simply use the difference, a_fin - b_fin, as the utility value.
But I made my AI to play a bit defensively in the starting one-third part of the game, hence until b_fin < 12 (12 since 12 = 36/3), I would assign a weight of 0.75 to a_fin & -1 to b_fin. While once b_fin > 12, I assign a weight of 1 to a_fin & -0.6 to b_fin.

Since the landing of the last stone in his/her own kalah gives the player an extra move, I tuned up the utility value by 2. But when his opponent gets an extra move like this, I tune down the utility value by 2.

If the player's last stone lands in his/her empty hole, that stone plus the opponent's opposite side hole stones are moved to the player's kalah. Hence, I tuned up the utility value by that extra stones count when this scenario was applicable. On the contrary, when this was possible with the opponent’s state too, I tuned down the utility value by that extra stones count.

The game ends when all the holes of a player become empty. So, when more than 4 holes of the player get empty, I penalize the utility value by 10. While if more than 5 holes of the opponent become empty, I tune up the utility value by 10.

IN SIMPLE SCENARIO, the simple heuristic function could be just ***state.a_fin - state.b_fin.***

Out of 5 games played at different depths, the win(W)-lose(L)-draw(T) counts for the AI are (at t = 1000 ms):

Depth	*Simple heuristic*	*Complex final heuristic*
3	4L, 1T	3W, 1L, 1T
5	3L, 2W	4W, 1L
7	4W, 1T	5W

In order to show the role of heuristic function in choosing moves, I have tried to use the search states possible for a depth of 3 which is explained as below (B is the AI, upper row state in the table):

<--5-4-3-2-1-0-------------------------------for player B

0	6	6	6	6	6	6	B
A	6	6	6	6	6	6	0

For player A--------------------------------0-1-2-3-4-5--->

When A0 moves, followed by an extra move from A2, resulting state is

0	6	6	6	7	7	7	B
A	0	7	0	8	8	8	2

For B0 movement, this is the graph based on depth = 3

7 7 7 8 8 0
1 2
1 7 0 8 8 8

----

7 7 7 8 8 0
1 2
0 8 0 8 8 8
Heuristic Value: -1.25
no cut-offf, -9223372036854775807, -1.25

7 7 7 8 9 1
1 3
1 0 1 9 9 9
Heuristic Value: -2.25
no cut-offf, -9223372036854775807, -2.25

7 8 8 9 9 1
1 3
1 7 0 0 9 9
Heuristic Value: -2.25
no cut-offf, -9223372036854775807, -2.25

8 8 8 9 9 1
1 3
1 7 0 8 0 9
Heuristic Value: -2.25
no cut-offf, -9223372036854775807, -2.25

8 8 8 9 9 1
1 3
2 7 0 8 8 0
Heuristic Value: -2.25
no cut-offf, -9223372036854775807, -2.25

Since, at depth 2, we have to minimize the heuristic value, so for this movement, the heuristic value chosen is -2.25 which is chosen as alpha for other possible move from B1.

For B1,

```
  7 7 7 8 0 7
1             2
  1 8 0 8 8 8
------------------------------------------

  7 7 7 8 0 7
1             2
  0 9 0 8 8 8
```

Heuristic Value: -1.25
no cut-offf-2.25-1.25

```
  7 7 7 9 1 8
1             3
  1 0 1 9 9 9
```

Heuristic Value: -2.25
alpha cut-off-2.25-2.25

At depth =2, the alpha beta cutoff comes into action & the further possible moves for the minimizing player are discarded.

Similarly for next possible moves of B, the best (max) heuristic value comes to -2.25. Hence the first move with this heuristic value is chosen which is B0.

PROGRAM DESIGN

I have implemented basic minimax search algorithm with alpha beta pruning to discard non-contributing states in order to improve time complexity.

The make_moves_by_rules method simply updates a state based on the rules of the game.

Before starting the minimax search using the minimax method, the leaf node condition is checked, if True, it just returns the heuristic value and accordingly choses the move. If not, the function calls minimax_with_alphabeta at depth - 1 and switching the maximizing/ minimizing player condition using the max_player flag returned by make_move_by_rules call.

The minimax_with_alphabeta method checks for a terminal state, using the following OR conditions:

- depth == 0
- state.a.count(0) == 6
- state.b.count(0) == 6
- state.a_fin > 36
- state.b_fin > 36)

If not terminal state, then it recursively calls itself at depth - 1 & adjusts the alpha & beta values while going down the state search tree. At state where beta <= alpha, it prunes that part of the tree & retracts up.

<u>EXPERIMENTS & RESULTS</u>

I tried playing against my AI at different depths {3, 5, 7} & I have set *t* to 1000 ms (i.e. 1 second)

Following is my system's CPU clock stats:
lscpu | grep MHz
CPU MHz: 800.014
CPU max MHz: 4000.0000
CPU min MHz: 400.0000

Please note that all the time difference outputs in the results following the depth lines are in seconds.

<u>Results (time in seconds) at depth = 3:</u>

depth = 3
0.00565409660339
depth = 3
0.00422191619873
depth = 3
0.00744390487671

depth = 3
0.00560116767883
depth = 3
0.00150012969971
depth = 3
0.0075831413269
depth = 3
0.00579500198364
depth = 3
0.0022120475769
depth = 3
0.00402402877808
depth = 3
0.00107789039612
depth = 3
0.00329995155334
depth = 3
0.00916910171509
depth = 3
0.00772595405579
depth = 3
0.00440382957458
depth = 3
0.00126600265503
depth = 3
0.00107502937317

Result - AI won (3 times Win, 1 Lose, 1 tie)

Results (time in seconds) at depth = 5:

depth = 5
0.0697519779205
depth = 5
0.0580379962921
depth = 5
0.0599839687347
depth = 5
0.0325100421906

depth = 5
0.0283300876617
depth = 5
0.0275869369507
depth = 5
0.0188610553741
depth = 5
0.0550410747528
depth = 5
0.0351660251617
depth = 5
0.0325908660889
depth = 5
0.0331130027771
depth = 5
0.0162739753723
depth = 5
0.0437178611755
depth = 5
0.0203170776367
depth = 5
0.0362548828125

Result - AI won (4 times Win, 1 lose)

Results (time in seconds) at depth = 7:

depth = 7
0.693994998932
depth = 7
0.365561008453
depth = 7
0.601978063583
depth = 7
0.330519914627
depth = 7
0.44337105751
depth = 7
0.357707023621

depth = 7
0.248655080795
depth = 7
0.179388046265
depth = 7
0.125916004181
depth = 7
0.29034614563
depth = 7
0.0199999809265

Result - AI won (5 times Win, also played against 4 random AI's over Intranet & won all)