[redacted]

[redacted]

[redacted].

Game Rules

The game starts with n stones numbered 1, 2, 3, ..., n. Players take turns removing one of the remaining numbered stones. At a given turn there are some restrictions on which numbers (i.e., stones) are legal candidates to be taken. The restrictions are:

- At the first move, the first player must choose an odd-numbered stone that is strictly less than n/2. For example, if n = 7 (n/2 = 3.5), the legal numbers for the first move are 1 and 3. If n = 6 (n/2 = 3), the only legal number for the first move is 1.
- At subsequent moves, players alternate turns. The stone number that a player can take must be a multiple or factor of the last move (note: 1 is a factor of all other numbers). Also, this number may not be one of those that has already been taken. After a stone is taken, the number is saved as the new last move. If a player cannot take a stone, he/she loses the game.

An example game is given below for n = 7:

```
Player 1: 3
Player 2: 6
Player 1: 2
Player 2: 4
Player 1: 1
Player 2: 7
Winner: Player 2
```

Program Specifications

There are 2 players: player 1 (called Max) and player 2 (called Min). For a new game (i.e., no stones have been taken yet), the Max player always plays first. Given a specific game board state, your program is to compute the best move for the current player using the given heuristic static board evaluation function. That is, only a single move is computed.

Input

A sequence of positive integers given as command line arguments separated by spaces: <#stones> <#taken_stones> <list_of_taken_stones> <depth>

- #stones: the total number of stones in the game
- #taken_stones: the number of stones that have already been taken in previous moves. If this number is 0, this is the first move in a game, which will be played by Max. (Note: If this number is even, then the current move is Max's; if odd, the current move is Min's)
- List_of_taken_stones: a sequence of integers indicating the indexes of the already-taken stones, ordered from first to last stone taken. Hence, the last stone in the list was the stone

taken in the last move. If #taken_stones is 0, this list will be empty. You can safely assume that these stones were taken according to the game rules.
- depth: the search depth. If depth is 0, search to end game states (i.e., states where a winner is determined).

For example, with input: 7 2 3 6 0, you have 7 stones and 2 stones, numbered 3 and 6, have already been taken — so it is the Max player's turn — and you should search to end game states.

Output
You are required to print out the following information to the console, after your alpha-beta search has completed:
- The best move (i.e., the stone number that is to be taken) for the current player (as computed by your alpha-beta algorithm)
- The value associated with the move (as computed by your alpha-beta algorithm)
- The total number of nodes visited
- The number of nodes evaluated (either an end game state or the specified depth is reached)
- The maximum depth reached (the root is at depth 0)
- The average effective branching factor (i.e., the average number of successors that are not pruned)

For example, here is sample input and output when it is Min's turn to move (because 3 stones have previously been taken), there are 4 stones remaining (3 5 6 7), and the Alpha-Beta algorithm should generate a search tree to maximum depth 3. Since the last move was 2 before starting the search for the best move here, only one child is generated corresponding to removing stone 6 (since it is the only multiplier of 2). That child node will itself have only one child corresponding to removing stone 3 (since it is the only factor of 6 among the remaining stones). So, the search tree generated will have the root node followed by taking 6, followed by taking 3, which leads to a terminal state (Max wins). So, returning from these nodes, from leaf to root, we get the output below.

Input:
```
7 3 1 4 2 3
```

Output:
```
Move: 6
Value: 1.0
Number of Nodes Visited: 3
Number of Nodes Evaluated: 1
Max Depth Reached: 2
Avg Effective Branching Factor: 1.0
```

All doubles/floats should be printed with 2 decimal places.

Static Board Evaluation

- The static board evaluation function should return values as follows:
  - At an end game state where Player 1 (MAX) wins: 1.0
  - At an end game state where Player 2 (MIN) wins: -1.0
- Otherwise,
  - if it is Player1(MAX)'s turn:
    - If stone 1 is not taken yet, return a value of 0 (because the current state is a relatively neutral one for both players)

    - If the last move was 1, count the number of the possible successors (i.e., legal moves). If the count is odd, return 0.5; otherwise, return -0.5.
    - If last move is a prime, count the multiples of that prime in all possible successors. If the count is odd, return 0.7; otherwise, return -0.7.
    - If the last move is a composite number (i.e., not prime), find the largest prime that can divide last move, count the multiples of that prime, including the prime number itself if it hasn't already been taken, in all the possible successors. If the count is odd, return 0.6; otherwise, return -0.6.
  - if it is Player2(MIN)'s turn, perform the same checks as above, but return the opposite (negation) of the values specified above.

Other Important Information

- Set the initial value of alpha to be $-\infty$ and beta to be ∞
- To break ties (if any) between child nodes that have equal values, pick the smaller numbered stone. For example, if stones 3 and 7 have the same value, pick stone 3.

[redacted]

[redacted]

[redacted]

[redacted]