Parallelized 2048 Solver

Alex Bala and Michal Hajlasz

12/20/2023

1 Introduction

The game 2048 is a solo puzzle game involving a 4x4 grid in which players merge numbered tiles. Tiles are numbered by powers of 2, and only tiles of the same value can merge to form a single tile with value equal to the sum of the merged tiles. A move consists of sliding all tiles in one direction (up, down, left, or right). After each move, the computer randomly places an additional 2 or 4-tile onto the board. The objective of the game is to score a 2048 tile, although the game goes on until the player is out of valid moves. Multiple methods can be used to program an agent to play this game. We explored a Minimax-based solution that is sped up via parallelization in Haskell.

2 Sequential Solution

We used an ExpectMiniMax algorithm in order to produce our sequential solution without parallelization. In this algorithm, we assume that the player is a maximizing agent, meaning that the player is trying to maximize some utility function on their turn. In a normal minimax algorithm, the opponent tries to minimize that utility function. However, since the opponent (computer) in reality is not antagonistic, we simulate the computer's turn with a random tile placement, thus reducing the computational complexity of the algorithm. In order to quantify the utility of each possible move, we use a set of heuristics to determine the value of a board. The weighted sum of the heuristics is what is passed up through the minimax tree. In normal scenarios, this sequential algorithm could be sped up with alpha-beta pruning. Given the nature of this project, we elected to not to do alpha-beta pruning to make parallelization more straightforward.

3 Heuristics

We used the following set of heuristics to guide our tree search:

- stayAliveHeuristic: Largely penalizes any boards that result in no possible moves.
- finishedHeuristic: Emphasizes movements that get the player closer to 2048 or 4096
- sumHeuristic: Computes the sum of the tiles on the board.
- gradientHeuristic: Uses a gradient filter to score boards with larger tiles in the bottom left corner higher.
- emptyTilesHeuristic: Tries to maximize the number of empty tiles on the board.
- monotonicityHeuristic: Ensures that among each row and column in the
- smoothnessHeuristic: Tries to avoid boards that are "jagged".

4 Parallel Solution

4.1 Strategy I

Our initial strategy was to parallelize the entire tree using parList. This resulted in speed up, but it was inefficient in utilizing cores and resulted in a low spark conversion rate.

```haskell
module Minimax where
...
bestMove :: Board -> IO (Maybe Direction)
bestMove board
    | null validMoves = putStrLn "No valid moves available." >> return
      Nothing
    | otherwise = do
       let best = fst $ maximumBy (comparing snd) validMoves
       putStrLn $ "Best move: " ++ show best
       return $ Just best
  where
    validMoves = filterMoves $ parallelizeMoves [U, D, L, R]
    filterMoves = filter (\(dir, _) -> move dir board /= board)
    parallelizeMoves dirs = map (\dir -> (dir, minimax (move dir board)
      False 0)) dirs `using` parList rdeepseq

minimax :: Board -> Bool -> Int -> Float
minimax board isMaximizer depth
  | depth == maxDepth || not (canMove board) = evalHeuristic board
  | isMaximizer = maximum $ parallelizeMinimax [move dir board | dir <- [U
     , D, L, R], move dir board /= board] False (depth + 1)
  | otherwise = minimum $ parallelizeMinimax [addRandomTile board] True (
     depth + 1)

parallelizeMinimax boards maximizing depth = map (\b -> minimax b
    maximizing depth) boards `using` parList rdeepseq

...
```

4.2 Strategy II

Our second strategy (which we analyze in our figures) was to parallize some of the tree but not all of it by limiting the parallelism to parts of the minimax tree where *depth* < 4. We also used parBuffer indstead of parList. This decreased the number of sparks fizziling and allowed more cores to be utilized at once.

```haskell
bestMove :: Board -> IO (Maybe Direction)
bestMove board
    | null validMoves = putStrLn "No valid moves available." >> return Nothing
    | otherwise = do
       let best = fst $ maximumBy (comparing snd) validMoves
       putStrLn $ "Best move: " ++ show best
       return $ Just best
  where
    validMoves = filterMoves $ parallelizeMoves [U, D, L, R]
    filterMoves = filter (\(dir, _) -> move dir board /= board)
    parallelizeMoves dirs = map (\dir -> (dir, minimax (move dir board) False
      0)) dirs `using` parBuffer 500 rdeepseq

minimax :: Board -> Bool -> Int -> Float
minimax board isMaximizer depth
  | depth == maxDepth || not (canMove board) = evalHeuristic board
  | depth > 4 =
    if isMaximizer
      then maximum [minimax newBoard False (depth + 1) | dir <- [U, D, L, R],
         let newBoard = move dir board, newBoard /= board]
      else minimax (addRandomTile board) True (depth + 1)
  | otherwise =
    if isMaximizer
```

```
        then maximum $ parallelizeMinimax [move dir board | dir <- [U, D, L, R
            ], move dir board /= board] False (depth + 1)
        else minimax (addRandomTile board) True (depth + 1)

parallelizeMinimax boards maximizing depth = map (\b -> minimax b maximizing
    depth) boards `using` parBuffer 500 rdeepseq
```

5 Speedup

We saw noticeable speedup thorugh our second strategy. The table and graph are from an average of 5 trials on each number of cores.

# of Cores	Speed Up	Spark Conversion Rate
1	1	0%
2	1.58	20%
3	1.65	53%
4	1.73	70%
5	1.71	76%
6	1.66	81%
7	1.74	82%
8	1.8	84%

Table 1: Performance Metrics

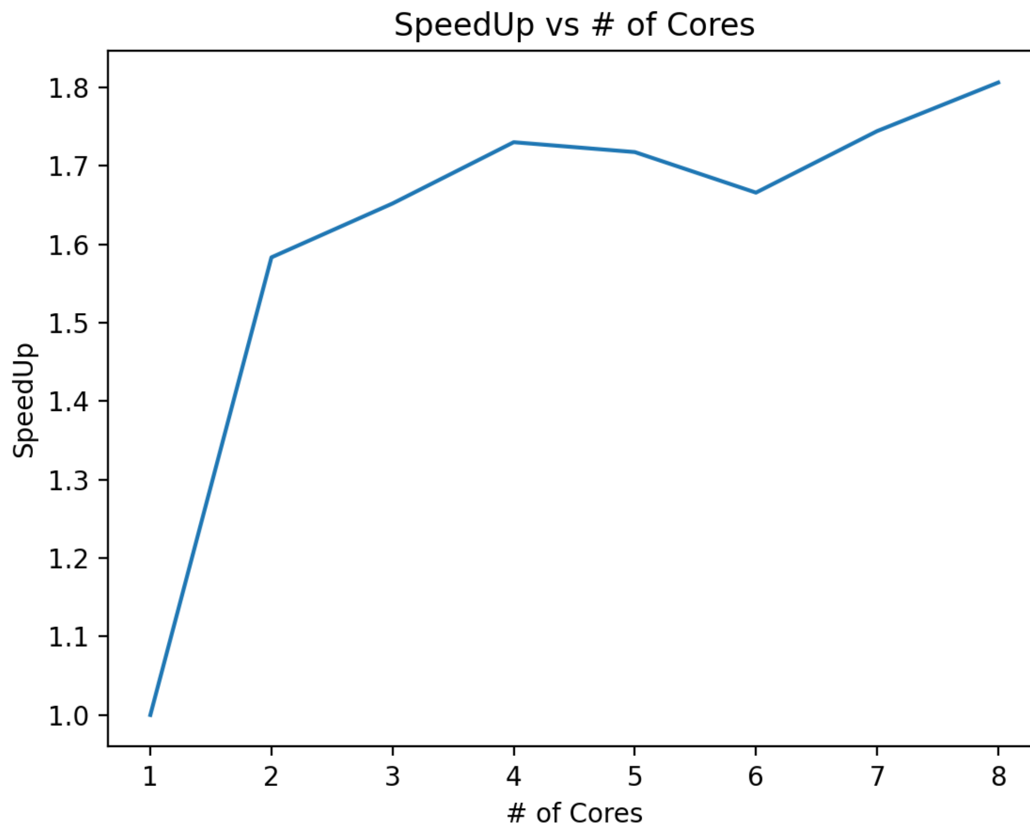

Figure 1: Speedup vs # of Cores

6 Core Usage and Sparks

Overall, we saw that workload was pretty even. Threadscope shows good distribution among the 4 cores and that on average between 2 and 3 cores are being used at a time. Given the nature of our algorithm, we believe that this is decent. In addition, spark conversion rate is above 70%.

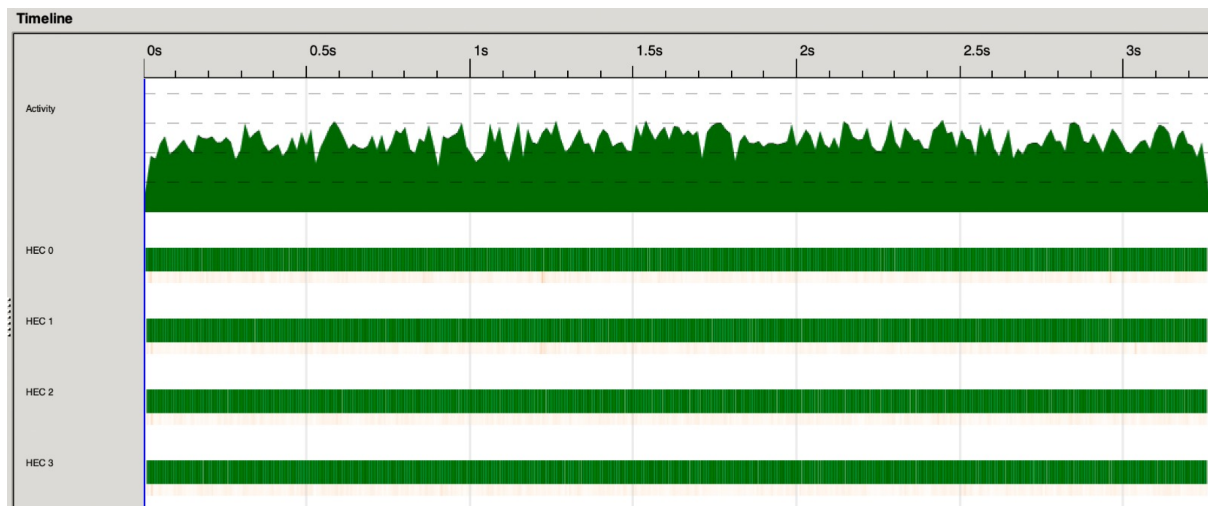

Figure 2: 4 Core Performance

7 Depth Considerations

There is a tradeoff between how many levels we traverse in the decision tree, score, and time taken between moves. While parallel strategies will not affect the score of the game (where we considered score to be the value of the highest tile at the endgame state), we are still concerned with trying to maximize this score. However we are more concerned with speed here, and hence we chose a maximum depth of 10 to balance score performance, speed, and spark conversion rate.

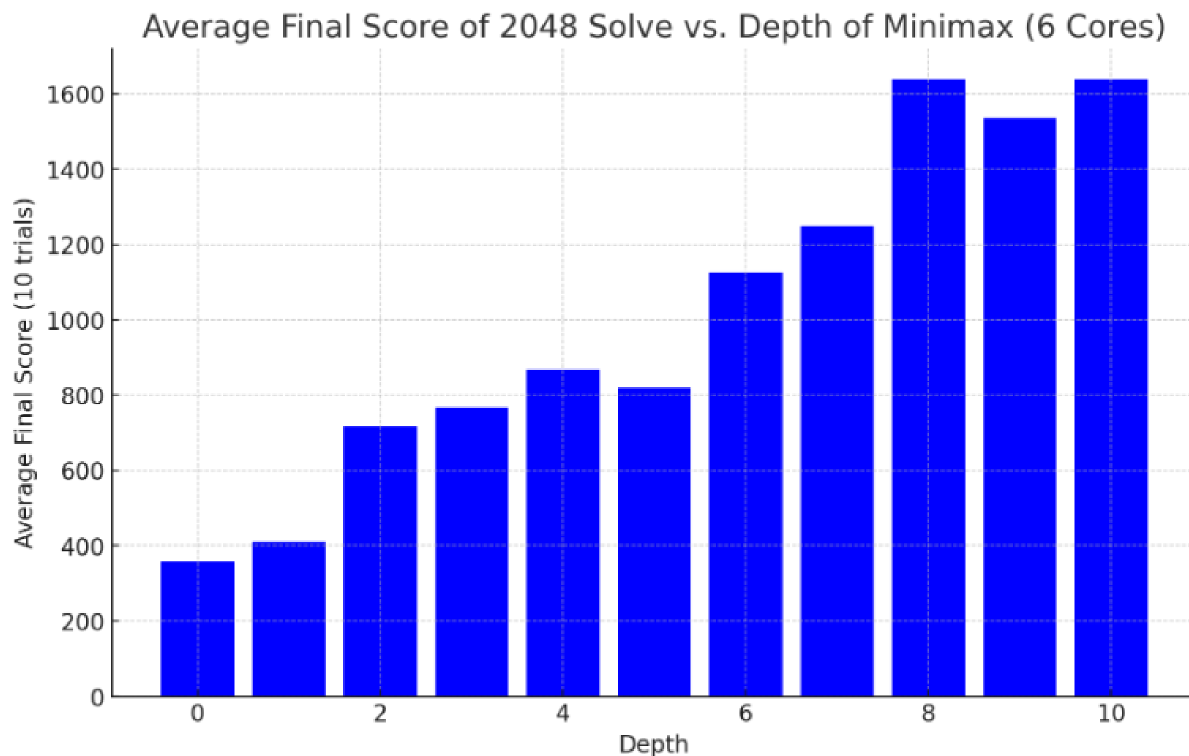

Figure 3: Depth vs Score

8 Further Considerations

While we were able to get noticeable speedups, we think there is still more work to be done on this project. Notably, we think there are other ways to parallelize this program that are more scalable. As we increase the number of cores available, we think that it should be possible to use more cores at a time and continue to speed up the program. In addition, along with parallel methods for efficiency we would like to explore adding sequential methods such as alpha-beta pruning.

9 Board.hs

```
module Board where

import System.Random
import Data.List      (elemIndices, transpose)
import Control.Parallel.Strategies
import Control.DeepSeq

type Board = [[Int]]


data Direction = U | D | L | R
  deriving (Enum, Bounded)

instance NFData Direction where
    rnf _ = ()

instance Show Direction where
    show U = "Up"
    show D = "Down"
    show L = "Left"
    show R = "Right"

instance Read Direction where
    readsPrec _ value =
        case value of
            "Up"     -> [(U, "")]
            "Down"   -> [(D, "")]
            "Left"   -> [(L, "")]
            "Right"  -> [(R, "")]
            _        -> []

instance Eq Direction where
    U == U = True
    D == D = True
    L == L = True
    R == R = True
    _ == _ = False

instance Ord Direction where
    compare U U = EQ
    compare U _ = LT
    compare _ U = GT
    compare D D = EQ
    compare D _ = LT
    compare _ D = GT
    compare L L = EQ
    compare L _ = LT
    compare _ L = GT
    compare R R = EQ

moveLeft :: Board -> Board
moveLeft b = map slideRowLeft b where
        slideRowLeft [] = []
        slideRowLeft [x] = [x]
        slideRowLeft (x:y:zs)
          | x == 0 = slideRowLeft (y : zs) ++ [0]
          | y == 0 = slideRowLeft (x : zs) ++ [0]
          | x == y = (x + y) : slideRowLeft zs ++ [0]
          | otherwise = x : slideRowLeft (y : zs)

move :: Direction -> Board -> Board
```

```
move U b = transpose . moveLeft . transpose $ b
move D b = transpose . map reverse . moveLeft . map reverse . transpose $ b
move L b = moveLeft b
move R b = map reverse . moveLeft . map reverse $ b

getEmptyTiles :: Board -> [(Int, Int)]
getEmptyTiles b = concatMap (\(n, row) -> zip (replicate 4 n) (elemIndices 0
    row)) (zip [0..3] b)

randomTileValue :: IO Int
randomTileValue = do
  x <- randomRIO (1, 10 :: Int) --assumes 90/10 2/4 probability
  return $ if x == 1 then 4 else 2

selectRandomTile :: [(Int, Int)] -> IO (Maybe (Int, Int))
selectRandomTile [] = return Nothing
selectRandomTile tiles = do
  i <- randomRIO (0, length tiles - 1)
  return $ Just (tiles !! i)

updateTile :: (Int, Int) -> Int -> Board -> Board
updateTile (r, c) val b = updateIndex (updateIndex (const val) c) r b where
    updateIndex fn i list = take i list ++ fn (head $ drop i list) : tail (drop
         i list)

addTile :: Board -> IO Board
addTile b = do
  let emptyTiles = getEmptyTiles b
  newValue <- randomTileValue
  maybePoint <- selectRandomTile emptyTiles
  case maybePoint of
    Just newPoint -> return $ updateTile newPoint newValue b
    Nothing -> return b -- No more empty tiles, return the board as is

getAvailableMoves :: Board -> [Board]
getAvailableMoves b = map (\x -> move x b) $ filter (\dir -> validMove dir b)
    moves
  where moves = [U,D,L,R]

validMove :: Direction -> Board -> Bool
validMove dir b = move dir b /= b

-- PRINT
printBoard :: Board -> IO ()
printBoard b = do
    putStrLn "-------------"
    putStrLn $ show (b !! 0)
    putStrLn $ show (b !! 1)
    putStrLn $ show (b !! 2)
    putStrLn $ show (b !! 3)
    putStrLn "-------------"
```

10 Minimax.hs

```
module Minimax where

import Board
import Heuristics
import System.Random
import Data.List (maximumBy)
import Data.Ord (comparing)
import System.IO.Unsafe (unsafePerformIO)
import Control.Parallel.Strategies


bestMove :: Board -> IO (Maybe Direction)
bestMove board
    | null validMoves = putStrLn "No valid moves available." >> return Nothing
    | otherwise = do
      let best = fst $ maximumBy (comparing snd) validMoves
      putStrLn $ "Best move: " ++ show best
      return $ Just best
  where
    validMoves = filterMoves $ parallelizeMoves [U, D, L, R]
    filterMoves = filter (\(dir, _) -> move dir board /= board)
    parallelizeMoves dirs = map (\dir -> (dir, minimax (move dir board) False
       0)) dirs `using` parBuffer 500 rdeepseq

minimax :: Board -> Bool -> Int -> Float
minimax board isMaximizer depth
  | depth == maxDepth || not (canMove board) = evalHeuristic board
  | depth > 4 =
      if isMaximizer
        then maximum [minimax newBoard False (depth + 1) | dir <- [U, D, L, R],
             let newBoard = move dir board, newBoard /= board]
        else minimax (addRandomTile board) True (depth + 1)
  | otherwise =
      if isMaximizer
        then maximum $ parallelizeMinimax [move dir board | dir <- [U, D, L, R
           ], move dir board /= board] False (depth + 1)
        else minimax (addRandomTile board) True (depth + 1)

parallelizeMinimax boards maximizing depth = map (\b -> minimax b maximizing
    depth) boards `using` parBuffer 500 rdeepseq

-- Helper function to simulate adding a random tile
addRandomTile :: Board -> Board
addRandomTile board = unsafePerformIO $ do
  let emptyTiles = getEmptyTiles board
  maybePoint <- selectRandomTile emptyTiles
  newValue <- randomTileValue
  return $ case maybePoint of
    Just point -> updateTile point newValue board
    Nothing -> board  -- Return the board as is if no empty tile is available

maxDepth :: Int
maxDepth = 10 -- Adjust the depth as needed

canMove :: Board -> Bool
canMove b = any (\d -> move d b /= b) [U, D, L, R]
```

11 GameManager.hs

```
module Main where

import System.Random
import Control.Monad (when)
import Board
import Minimax
import Data.Time.Clock
import Data.Csv
import qualified Data.ByteString.Lazy as BL
import qualified Data.Vector as V

data GameState = GameState {
    board :: Board,
    isPlayerTurn :: Bool,
    gameOver :: Bool
}

-- Initialize the game state with two random tiles
initGame :: IO GameState
initGame = do
    initialBoard <- addTile =<< addTile (replicate 4 (replicate 4 0))
    return $ GameState initialBoard True False

-- Check if the game is over
checkGameOver :: Board -> Bool
checkGameOver b = null (getEmptyTiles b) && not (canMove b)

doublesToCsv :: [Double] -> BL.ByteString
doublesToCsv doubles = encode $ map (\x -> [x]) doubles

data TimeRecord = TimeRecord { timeTaken :: Double }

instance ToRecord TimeRecord where
    toRecord (TimeRecord t) = record [toField t]

gameLoop :: GameState -> [Double] -> IO ()
gameLoop state d
    | gameOver state = do
        putStrLn $ "Game Over!"
        putStrLn $ "Score: " ++ show (maximum (map maximum (board state)))
        BL.writeFile "output6-2.csv" $ doublesToCsv d
    | otherwise = do
        printBoard (board state)
        if isPlayerTurn state
            then do
                startTime <- getCurrentTime
                newState <- playerTurn state
                endTime <- getCurrentTime
                let timeTaken = realToFrac $ (diffUTCTime endTime startTime) ::
                     Double
                gameLoop newState $ d ++ [timeTaken]
            else do
                newState <- computerTurn state
                gameLoop newState $ d

-- Handle player's turn
playerTurn :: GameState -> IO GameState
playerTurn state = do
    let b = board state
    moveDir <- bestMove b -- `Maybe Direction`
    case moveDir of
```

```
        Just dir -> do
            let newBoard = move dir b
            return state {board = newBoard, isPlayerTurn = False}
        Nothing ->
            return state

-- Handle computer's turn by adding a random tile
computerTurn :: GameState -> IO GameState
computerTurn state = do
    newBoard <- addTile (board state)
    let newState = state {board = newBoard, isPlayerTurn = True}
    return newState {gameOver = checkGameOver newBoard}

-- Start the game
main :: IO ()
main = do
    a <- initGame
    gameLoop a []
```

12 Heuristics.hs

```haskell
module Heuristics where

import Board
import Data.List (transpose, foldl')

printScore :: Board -> IO ()
printScore b = do
    putStrLn $ show (smoothnessHeuristic b)
    putStrLn $ show (monotonicityHeuristic b)
    putStrLn $ show (emptyTilesHeuristic b)
    putStrLn $ show (gradientHeuristic b)
    putStrLn $ show (finishedHeuristic b)
    putStrLn $ show (stayAliveHeuristic b)
    putStrLn $ show (sumHeuristic b)
    putStrLn $ "Total: " ++ show (evalHeuristic b)
    return ()

evalHeuristic :: Board -> Float
evalHeuristic b = realToFrac $ sum $ zipWith (*) weights funcs
    where weights = [1,1,1,1,0.25,1000000,90000]
          funcs = [logBase 2 $ sumHeuristic b,
                   smoothnessHeuristic b,
                   monotonicityHeuristic b,
                   logBase 2 $ gradientHeuristic b,
                   finishedHeuristic b,
                   stayAliveHeuristic b]

sumHeuristic :: Board -> Double
sumHeuristic b = fromIntegral $ sum $ map sum b

emptyTilesHeuristic :: Board -> Double
emptyTilesHeuristic b = fromIntegral $ length (getEmptyTiles b)

stayAliveHeuristic :: Board -> Double
stayAliveHeuristic b
    | (getAvailableMoves b == []) = -1
    | otherwise = 0

--The following Heuristics are modified from an older project from pfp2048 2021
     project
smoothnessHeuristic :: Board -> Double
smoothnessHeuristic b = (foldr smoothHelper 0 b) + (foldr smoothHelper 0 $ (
    transpose b)) where
  smoothHelper :: [Int] -> Double -> Double
  smoothHelper (x:y:zs) total
    | x == y    = smoothHelper (y:zs) total + 1
    | otherwise = smoothHelper (y:zs) total
  smoothHelper _ total = total

monotonicityHeuristic :: Board -> Double
monotonicityHeuristic b = fromIntegral $ (fst $ foldl' helper  (0,0) b) + (fst
    $ foldl' helper (0,0) (map reverse $ transpose b)) where
    helper :: (Int,Int) -> [Int] -> (Int,Int)
    helper (total, score) (f:s:tl)
        | tl == []              = (total, score)
        | (f == s) && (s==0)    = helper (total, score) (s:tl)
        | f >= s                = helper (total+score, score+1) (s:tl)
        | f < s                 = (total, score)

gradientHeuristic :: Board -> Double
gradientHeuristic b = realToFrac $ sum $ map sum $ zipWith (zipWith (*)) b filt
```

```haskell
            where filt = [[0,0,0,0], [128, 1, 0, 0], [512, 128, 1, 0], [2048,
                512, 128, 1]]

finishedHeuristic :: Board -> Double
finishedHeuristic b
    | maximum (map maximum b) == 4096    = 100
    | maximum (map maximum b) == 2048    = 1
    | otherwise                          = 0
```