# Mahjong Analyzer

Progress 5

Muhammad Jilan Wicaksono

November 24, 2025

Universitas Indonesia

# Target dan Pencapaian i

**Membuat Suite Test menggunakan QuickCheck:** 70%

- Membuat generator dari QuickCheck untuk Data Type Tile.
- Membuat generator untuk Data Type AgariHand dan Meld berdasarkan genTile.
- Membuat daftar properti yang secara matematis pasti terpenuhi oleh fungsi-fungsi yang sudah dibuat.

**Menulis README:** 30%

- Berisi penjelasan singkat algoritma dari penghitung shanten dan cara kerja memoTree.

# Target dan Pencapaian ii

```haskell
instance Arbitrary Suit where
  arbitrary :: Gen Suit
  arbitrary = elements [Manzu, Pinzu, Souzu, Honor]

genTile :: Gen Tile
genTile = do
  suit <- arbitrary
  n <- case suit of
    Honor -> choose (1,7)
    _ -> choose (1,9)
  return (Tile suitTile= suit numberTile= n)

instance Arbitrary Tile where
  arbitrary :: Gen Tile
  arbitrary = genTile
```

```haskell
genSequence :: Gen Meld
genSequence = do
  suit <- elements [Manzu, Pinzu, Souzu]
  n    <- choose (1,7)
  return (Sequence meldTile= (Tile suit n))

genTriplet :: Gen Meld
genTriplet = Triplet <$> genTile

instance Arbitrary Meld where
  arbitrary :: Gen Meld
  arbitrary = oneof [genSequence, genTriplet]

instance Arbitrary KMeld where
  arbitrary :: Gen KMeld
  arbitrary = KMeld <$> arbitrary <*> arbitrary
```

# Target dan Pencapaian iii

```haskell
genStandard :: Gen AgariHand
genStandard = Standard <$> vectorOf 4 arbitrary <*> arbitrary

genChiitoi :: Gen AgariHand
genChiitoi = do
  ts <- nub <$> vectorOf 7 arbitrary
  return (Chiitoi (map Pair ts))

genKokushi :: Gen AgariHand
genKokushi = do
  terminals <- shuffle kokushiList
  let pair = head terminals
  return (Kokushi (pair:terminals))

kokushiList :: [Tile]
kokushiList =
  [ Tile suitTile=s numberTile=n | s <- [Manzu, Pinzu, Souzu], n <- [1,9] ] ++
  [ Tile suitTile=Honor numberTile=n | n <- [1..7] ]

instance Arbitrary AgariHand where
  arbitrary :: Gen AgariHand
  arbitrary = frequency
    [ (8, genStandard)
    , (1, genChiitoi)
    , (1, genKokushi)
    ]
```

# Target dan Pencapaian iv

```haskell
-- ------------------------------------------------------------
-- Property: HandCount round-trip
-- ------------------------------------------------------------

prop_roundTrip :: AgariHand -> Bool
prop_roundTrip ag =
  let h  = agariToHand ag
      hc = handToCount h
  in sort (countToHand hc) == sort h

-- ------------------------------------------------------------
-- Property: findPartition reconstructs hand
-- ------------------------------------------------------------

prop_partitionCorrect :: AgariHand -> Bool
prop_partitionCorrect ag =
  let h  = agariToHand ag
      hc = handToCount h
      ag' = normalizeAgari ag
  in ag' `elem` map normalizeAgari (findPartition hc [])
```

```haskell
-- ------------------------------------------------------------
-- Property: Fu properties
-- ------------------------------------------------------------

prop_fuMultipleOf10 :: HandContext -> AgariHand -> Bool
prop_fuMultipleOf10 ctx ag =
  case ag of
    Chiitoi _ -> calcFu ctx ag == 25
    _         -> calcFu ctx ag `mod` 10 == 0

prop_fuAtLeast20 :: HandContext -> AgariHand -> Bool
prop_fuAtLeast20 ctx ag =
  let f = calcFu ctx ag
  in case ag of
       Kokushi _ -> f == 0
       Chiitoi _ -> f == 25
       _         -> f >= 20
```

# Target dan Pencapaian v

```haskell
-- ------------------------------------------------------------
-- Property: score invariant under hand permutation
-- ------------------------------------------------------------

prop_scorePermutationInvariant :: HandContext -> AgariHand -> Property
prop_scorePermutationInvariant ctx ag =
  let h  = agariToHand ag
      sc = calcScore ctx ag
  in forAll (shuffle h) $ \h2 ->
       let hc2 = handToCount h2
           ag2s = findPartition hc2 []
       in null ag2s || maximum (map (calcScore ctx) ag2s) == sc
```

# Target dan Pencapaian vi

```
Running 1 test suites...
Test suite mahjong-test: RUNNING...
+++ OK, passed 100 tests.
*** Failed! Falsified (after 1 test):
Standard [Open(5P,6P,7P),Open(4M,4M,4M),Closed(5Z,5Z,5Z),Open(3M,3M,3M)] (8P,8P)
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
*** Failed! Falsified (after 47 tests):
HandContext {openMelds = [], prevalentWind = 1, seatWind = 3, winTile = 9M, winMethod = Ron, isRiichi = False, isDealer = False}
Standard [Closed(4S,5S,6S),Closed(1M,1M,1M),Open(2M,3M,4M),Open(7S,8S,9S)] (7S,7S)
+++ OK, passed 100 tests; 775 discarded.
*** Failed! Falsified (after 8 tests):
HandContext {openMelds = [], prevalentWind = 4, seatWind = 1, winTile = 3P, winMethod = Ron, isRiichi = True, isDealer = True}
Standard [Open(3S,4S,5S),Open(2P,3P,4P),Closed(5S,6S,7S),Closed(1S,2S,3S)] (2P,2P)
[7S,2P,6S,2P,3S,5S,1S,2P,4P,3S,2S,3P,4S,5S]
Test suite mahjong-test: PASS
```

# Link Commit

## Link Commit

- https://github.com/WLan1707/mahjong-analyzer/commit/7de788b591968764c895ec04c64a1afe44b4b6be

# Lesson Learned: Memoization Using Tree i

Secara konsep mirip seperti Fibonacci, jadi akan digunakan contoh ini saja:

$$F(0) = 1, F(1) = 1, F(n) = F(n-1) + F(n-2)$$

Tanpa memoization, dapat memanggil

```
fix fib 30 == 1346269
```

```haskell
import Data.Function (fix)

fib :: (Integer -> Integer) -> (Integer -> Integer)
fib f 0 = 1
fib f 1 = 1
fib f n = f (n - 1) + f (n - 2)
```

# Lesson Learned: Memoization Using Tree ii

Dengan list, dapat dipanggil

```
faster_f 3000 == 90018235110926312 49
```

```haskell
f_list :: [Integer]
f_list = map (fib faster_f) [0..]

faster_f :: Integer -> Integer
faster_f n = f_list !! fromInteger n
```

Jika diperhatikan, `faster_f` memanfaatkan `fmap` dari Functor List dengan fungsi lookup `(!!)` sebesar $O(n)$.

dapat dimanfaatkan Functor lain dengan fungsi lookup sebesar $O(\log n)$, yaitu tree.

# Lesson Learned: Memoization Using Tree iii

```
Visualisasi f_tree

- nats:
        0
      /   \
     1     2
    / \   / \
   3  4  5   6

- f_tree:
          g 0
        /   \
      g 1     g 2
     /   \   /   \
   g 3  g 4  g 5  g 6
dengan g = fib fastest_f
sehingga
g n = fib fastest_f
    = fastest_f (n - 1) + ( fastest_f (n-2) )
    = index f_tree (n - 1) + ( index f_tree (n - 2) )
dan
fastest_f n = index f_tree n
            = g n
```

```haskell
data Tree a = Tree (Tree a) a (Tree a)
instance Functor Tree where
    fmap f (Tree l m r) = Tree (fmap f l) (f m) (fmap f r)

index :: Tree a -> Integer -> a
index (Tree _ m _) 0 = m
index (Tree l _ r) n = case (n - 1) `divMod` 2 of
    (q,0) -> index l q
    (q,1) -> index r q

nats :: Tree Integer
nats = go 0 1
    where
        go !n !s = Tree (go l s') n (go r s')
            where
                l = n + s
                r = l + s
                s' = s * 2

f_tree :: Tree Integer
f_tree = fmap (fib fastest_f) nats

fastest_f :: Integer -> Integer
fastest_f = index f_tree
```

`f_tree` dibuat dulu secara lazy, ketika `fastest_f n` dipanggil, hanya perlu melihat node ke $(n-1)$ dan $(n-2)$ dari `f_tree` menggunakan index, dan keduanya juga hanya melihat kode sebelumnya saja juga, dan seterusnya.

# Lesson Learned: Memoization Using Tree iv

Dengan ini, dapat dipanggil bilangan fibonacci ke 70000 secara cepat.

```
ghci> fastest_f 70000
```

# Lesson Learned: Memoization Using Tree v

Penerapan pada perhitungan shanten

```haskell
shantenSuitTileFirst :: HandCount -> ResultSuit -> ResultSuit
shantenSuitTileFirst hand result
  | Map.null hand     = result
  | otherwise =
    let
      smallestTile = fst $ Map.findMin hand

      pairResult = [ shantenSuitTileFirst (removePair (Pair pairTile=smallestTile) hand) (addResult result (0,1,0)) | isPair
      smallestTile hand ]
      tripletResult = [ shantenSuitTileFirst (removeTriplet (Triplet meldTile=smallestTile) hand) (addResult result (1,0,0)) |
      isTriplet smallestTile hand ]
      sequenceResult = [ shantenSuitTileFirst (removeSequence (Sequence meldTile=smallestTile) hand) (addResult result (1,0,0)) |
      isSequence smallestTile hand ]
      missMidResult = [ shantenSuitTileFirst (removeMissMid (MissMid smallestTile) hand) (addResult result (0,0,1)) | isMissMid
      smallestTile hand ]
      missOutResult = [ shantenSuitTileFirst (removeMissOut (MissOut smallestTile) hand) (addResult result (0,0,1)) | isMissOut
      smallestTile hand ]
      fallbackResult = [ shantenSuitTileFirst (removeOneTile smallestTile hand) result ]
      allResult = pairResult ++ tripletResult ++ sequenceResult ++ missMidResult ++ missOutResult ++ fallbackResult

    in maximumBy (comparing compResult) allResult
```

# Lesson Learned: Memoization Using Tree vi

```haskell
shantenInt :: (Int -> Suit -> ResultSuit -> ResultSuit) -> Int -> Suit -> ResultSuit -> ResultSuit
shantenInt f handInt suit result
  | handInt == 0     = result
  | otherwise =
    let
      hand = decodeSuitMap suit handInt
      smallestTile = fst $ Map.findMin hand

      pairResult = [ f (extractSuit suit $ removePair (Pair pairTile=smallestTile) hand) suit (addResult result (0,1,0)) | isPair
      smallestTile hand ]
      tripletResult = [ f (extractSuit suit $ removeTriplet (Triplet meldTile=smallestTile) hand) suit (addResult result (1,0,
      0)) | isTriplet smallestTile hand ]
      sequenceResult = [ f (extractSuit suit $ removeSequence (Sequence meldTile=smallestTile) hand) suit (addResult result (1,0,
      0)) | isSequence smallestTile hand ]
      missMidResult = [ f (extractSuit suit $ removeMissMid (MissMid smallestTile) hand) suit (addResult result (0,0,1)) |
      isMissMid smallestTile hand ]
      missOutResult = [ f (extractSuit suit $ removeMissOut (MissOut smallestTile) hand) suit (addResult result (0,0,1)) |
      isMissOut smallestTile hand ]
      fallbackResult = [ f (extractSuit suit $ removeOneTile smallestTile hand) suit result ]

      allResult = pairResult ++ tripletResult ++ sequenceResult ++ missMidResult ++ missOutResult ++ fallbackResult

    in maximumBy (comparing compResult) allResult
```

```haskell
memoShantenSuit :: Int -> Suit -> ResultSuit -> ResultSuit
memoShantenSuit = memoTree shantenInt

memoShanten :: HandCount -> Int
memoShanten hand
  | isAgari hand    = -1
  | otherwise       = uncurry3 shantenValue result
    where
      result = foldl addResult (0,0,0) resultPerSuit
      resultPerSuit = [resultMan, resultPin, resultSou, resultHon]
      resultMan = memoShantenSuit ( extractSuit Manzu hand ) Manzu (0,0,0)
      resultPin = memoShantenSuit ( extractSuit Pinzu hand ) Pinzu (0,0,0)
      resultSou = memoShantenSuit ( extractSuit Souzu hand ) Souzu (0,0,0)
      resultHon = memoShantenSuit ( extractSuit Honor hand ) Honor (0,0,0)
```

# Lesson Learned: QuickCheck i

Haskell sudah mempunyai cara untuk membangun domain dari suatu fungsi secara otomatis, yang bisa digunakan untuk melakukan pengecekan suatu sifat dari fungsi itu.

Meskipun secara otomatis, perlu diberikan argumen agar domain tersebut membuat fungsi itu terdefinisi dengan baik di sana.

Pengecekannya juga hanya berupa sifat, yang harus bisa dibuktikan secara matematis (atau dari aturannya) tanpa program.

Sehingga harus berhati-hati dalam pendefinisian domain dan sifat yang ingin diperiksa.

# Lesson Learned: QuickCheck ii

Kelas Arbitrary juga memiliki tipe Gen, yang merupakan monad dan pastinya aplikatif, sehingga ada dua cara penulisan generator:

1. do-notation jika generator bergantung antara variable

```haskell
genSequence :: Gen Meld
genSequence = do
  suit <- elements [Manzu, Pinzu, Souzu]
  n    <- choose (1,7)
  return (Sequence meldTile=(Tile suit n))
```

2. Applicative untuk generator yang independen

```haskell
genStandard :: Gen AgariHand
genStandard = Standard <$> vectorOf 4 arbitrary <*> arbitrary
```

Ini membuat kode generator lebih ekspresif, serta jangan paksakan penggunaan suatu notasi jika ada yang lebih bagus.