Javier Escareno
Lab 7

**Part 1 A**

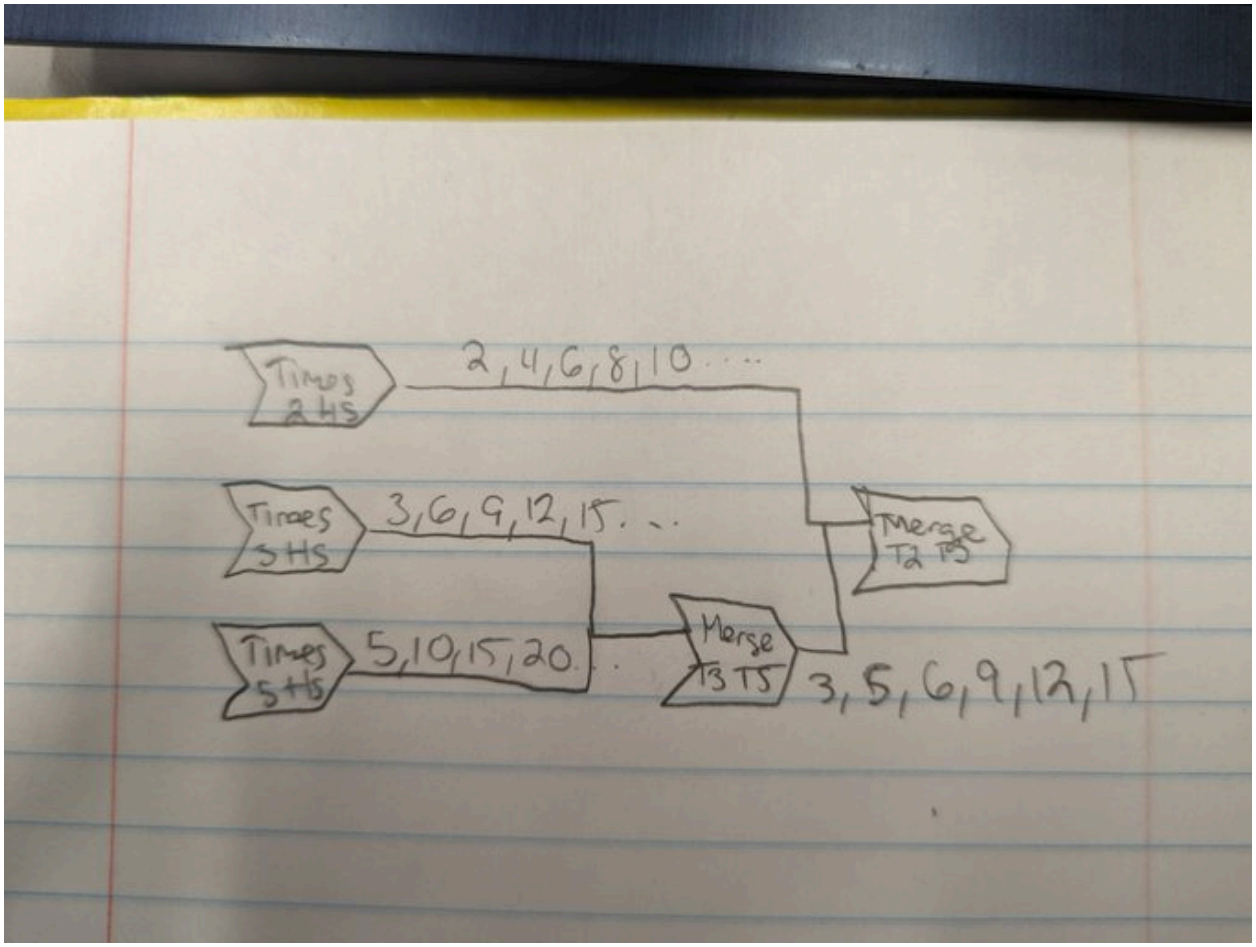
```
fun {Times N Hs}
        fun {$}
                (H # Hr) = {Hs}
                in
                        ((N*H) # {Times N Hr})
        end
end

fun {Merge Xs Ys}
        fun {$}
                (X#Xr) = {Xs}
                (Y#Yr) = {Ys}
                in
                        if (X < Y) then (X # {Merge Xr Ys})
                        elseif (X > Y) then (Y # {Merge Xs Yr})
                        else (X # {Merge Xr Yr})
                        end
        end
end

fun {GenerateHamming Hs}
        fun {$}
                (1 # {Merge {Times 2 Hs} {Merge {Times 3 Hs} {Times 5 Hs}}})
        end
end

fun {Take N Xs}
        if (N > 0) then
                (X # Xr) = {Xs} in
                (X | {Take (N - 1) Xr})
        else
                        nil
        end
end

HammingSequence = {Take 10 {GenerateHamming {Generate 1}}}

skip Browse HammingSequence
```

Terminal Output -

V1 : 3
V2 : 5

V3 : 4

HammingSequence : [ 1 2 3 4 5 6 8 9 10 12 ]




**Part 1 B**

data Gen a = G (() -> (a, Gen a))

generate :: Int -> Gen Int

```
generate n = G (\_ -> (n, generate (n+1)))
times :: Int -> Gen Int -> Gen Int
times n (G f) = let (h, hs) = f() in G (\_ -> (n * h, times n hs))


merge :: Ord a => Gen a -> Gen a -> Gen a
merge g1@(G f1) g2@(G f2) | x < y = G (\_ -> (x, merge xs g2))
                          | y < x = G (\_ -> (y, merge g1 ys))
                          | otherwise = G (\_ -> (x, merge xs ys))
                          where (x, xs) = f1()
                                (y, ys) = f2()


generateHamming :: Gen Int -> Gen Int
generateHamming hs = G (\_ -> let merged2 = merge (times 2 hs) (merge (times 3 hs) (times 5
hs))
                              in (1, merged2))


gen_take :: Int -> Gen a -> [a]
gen_take 0 _ = []
gen_take n (G f) = let (x,g) = f () in x : gen_take (n-1) g

ghci> gen_take 10 (generate 1)
[1,2,3,4,5,6,7,8,9,10]
ghci> gen_take 10 (generateHamming (generate 1))
[1,2,3,4,5,6,8,9,10,12]
ghci>
```

**Part 2 A**

```
fun {IntToNeed L}
        case L
        of nil then nil
        [] (X|Xs) then ByNeedValue in
                byNeed fun {$} X end ByNeedValue
                (ByNeedValue|{IntToNeed Xs})
        end
end
```

**Part 2 B**

```
AndG = {GateMaker fun {$ X Y}
                        if (X == 0) then 0
                        elseif (Y == 0) then 0
```

```
                    else 1
                end
 end}

OrG = {GateMaker fun {$ X Y}
                    if (X == 1) then 1
                    elseif (Y == 1) then 1
                    else 0
                end
 End}
```

**Part 2 C**

```
fun {MulPlex A B S} SelectA SelectB in
        SelectA = {AndG {NotG S} A}
        SelectB = {AndG S B}
        {OrG SelectA SelectB}
 end
```

**Part 2 D1**

The values not needed in A and B will be Highlighted

A = {IntToNeed [0 1 1 0 0 1]}

B = {IntToNeed [1 1 1 0 1 0]}

**Part 2 D2**

Needed: 191 -> 1
Needed: 258 -> 1
Needed: 292 -> 1
Needed: 324 -> 1
Needed: 358 -> 0
Needed: 361 -> 0

The values that were needed to match up did so except for a few cases. The total number of needed variables matches up and the frequency for each value is accurate to the expected output amount.