2020 Algorithm II Examination Solution

1. a. i) The definition of `closest` is as follows

```
closest :: [Integer] -> (Integer, Integer)
closest []       = error "Empty list supplied!"
closest [x]      = error "List only has one element!"
closest (x:y:[]) = (x, y)
closest (x:y:xs) = (x, y) ∏ (closest (y:xs))
```

The time complexity of `closest` is linear since it only traverses the entire list once. It will recursively compare the differencee of the first two elements with the difference of the closest pair in the rest of the list, which only travel the entire list once. (e.g. level of recursion is the same as the length of the list)

ii) Divide and conquer algorithm first divide problems into subproblems. When the problem becomes small enough, it uses the same strategy across all subproblems to generate a solution. Finally, the algorithm will conquer the solutions together and return a final result.

iii) The definition of `closest'` is as follows

```
closest' :: [Integer] -> (Integer, Integer)
closest' []  = error "Empty list supplied!"
closest' [x] = = error "List only has one element!"
closest' (x:y:[])   = (x, y)
closest' (x:y:z:[]) = (x, y) ∏ (y, z)
closest' list       = (closest' fh) ∏ (closest' sh)
  where (fh, sh) = split ((length list) / 2) list

--You can also define split with signiture `[a] -> ([a], [a])` using a helper
function
split :: Int -> [a] -> ([a], [a])
split 0 list   = ([], list)
split n (x:xs) = (x:f, s)
  where f = fst rec
        s = snd rec
        rec = split (n - 1) xs
```

The complexity of `closest'` is $O(n \log n)$.

b. i) The definition of these functions in `AList` version is as follows

```
head :: AList a -> a
head AList fi _ arr = arr ! fi

last :: AList a -> a
last AList _ li arr = arr ! li

split :: AList a -> (AList a, AList a)
split AList fi li arr = (AList fi n arr, AList (n+1) li arr)
  where n = (li - fi) / 2
```

ii) The definition of `closest''` is as follows

```
closest'' [Integer] -> (Integer, Integer)
closest'' []  = error "Empty list supplied!"
closest'' [x] = error "List only has one element!"
closest'' list = helper (fromList list)
  where helper :: AList Integer -> (Integer, Integer)
        helper AList fi li arr
          | li - fi == 1 = (arr ! fi, arr ! li)
          | li - fi == 2 = (arr ! fi, arr ! fi + 1) ∏ (arr ! fi + 1, arr ! li)
        helper list = (helper fh) ∏ (helper sh)
          where (fh, sh) = split arr
```

Since the `split` operation on `AList` is operating at constant time, the overall time complexity of `closest''` is therefore $O(n)$. (Notice that the complexity of `helper` is $O(\log n)$, but the `fromList` has complexity $O(n)$)

In terms of time complexity, both `closest` and `closest''` have $O(n)$ as the time complexity, but the latter one takes less space (and even faster) because the deepest recursion level of `closest''` is $\log n$ while the deepest recursion level of `closest` is n, where n is the list length. In other words, the time complexity means different things: $O(n)$ in `closest` indicates the level of recursion, while $O(n)$ in `closest''` indicates the `fromList` operation. Clearly, recursion takes more stack space/more time to run than `fromList`.

1. a. i) The definition of `catalan'` is as follows

```
catalan' :: Int -> Integer
catalan' 0 = 1
catalan' n = arr ! n
  where arr = tabulate (0, n) memo
        memo :: Int -> Integer
        memo 0 = 1
        memo 1 = 1
        memo n = sum [arr ! i * arr ! (n - i - 1) | i <- [0..n-1]]
```

ii) The overall complexity of `catalan'` is $O(n^2)$ because each time when calculating the value of `memo n`, we need to calculate `sum [arr ! i * arr ! (n - i - 1) | i <- [0..n-1]]` where the value of `arr ! (n - i - 1)` is another summation up to `arr ! (n - i - 2)`, etc.

b. i) Suppose we calculate the result using the order $A_0 A_1 A_2$

The number of scalar operations when calculating $A_0 A_1$ is 8.

Similarly, the number of scalar operation when calculating $(A_0 A_1) A_2$ is 12. Hence the total number of scalar operations is $8 + 12 = 20$

Suppose we calculate the result using the order $A_0 (A_1 A_2)$ instead

The number of scalar operations of is 12 for both $A_1 A_2$ and $A_0 (A_1 A_2)$. Hence the total number of scalar operations is $12 + 12 = 24$

ii) The definition of `chain` is as follows

```
-- Idea: divide the chain of matrix multiplication into two parts, then choose
the one with min amount of operations

chain :: Array Int Int -> (Int, Int) -> Int
chain a (i, j)
  | i == j    = 0
  | otherwise = minimum [
                  let count = (a ! i) * (a ! (k + 1)) * (a ! (j + 1))
                  in chain a (i, k) + chain a (k + 1, j) + count |
                                                      k <- [i..j-1]]
```

iii) The definition of `chain'` is as follows

```
chain' :: Array Int Int -> (Int, Int) -> Int
chain' a (i, j) = arr ! (i, j)
  where arr = tabulate ((0, 0), (j, j)) (uncurry memo)
        memo :: Int -> Int -> Int
        memo m n =
          | m == n    = 0
          | otherwise = minimum [
              let count = (a ! m) * (a ! (k + 1)) * (a ! (n + 1))
              in arr ! (m, k) + arr ! (k + 1, n) + count
                                                  | k <- [m..n-1]]
```

You can test all the programs in this solution with the following `tabulate` definition and `import Data.Array`

```
tabulate :: Ix i => (i, i) -> (i -> a) -> Array i a
tabulate (u,v) f = array (u,v) [(i,f i) | i <- range (u,v)]
```