Dynamic Programming Done Right

1 A tale of few Pingda's

Pingda numbers (or more populerly called Fibonacci numbers) are a sequence of numbers defined as $1, 1, 2, \ldots$ or the recursion $F(n) = F(n-1) + F(n-2)$. How can we generate the n^{th} Pingda in haskell?

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

Time Complexity

The time complexity of a plain recursion is

number of function calls $\times$ time complexity of a single call

We can clearly note that we first call `ping (n-1)` and `ping (n-2)`. Then each of them calls 2 more functions. This keeps going till we call `ping 1` or `ping 0`. This takes at-least $n-1$ steps.
Thus, $T(n) = O(2^{n-1}) = O(\frac{1}{2}2^n) = O(2^n)$.

The thing is, our code wastefully calculates a lot of values multiple times. If we just knew the last two values, we could be done. What if we store them?

```
ping :: Integer -> Integer
ping n = fst (helper n) where
    helper 0 = (1,1)
    helper n = (a+b, a) where helper (n-1) = (a,b)
```

This code is more optimized as it is storing the last two values and doesn't recompute the same values many, many times. How much more optimized?

Time Complexity

We immediately call `helper (n-1)` which calls `helper (n-2)` and so on. Notice, each of these calls is of $O(1)$ times and we make n calls.
Thus, $T(n) = O(n)$

This is the main idea of dynamic programming.

Idea

Dynamic Programming is a way to optimize recursion by storing previous values to prevent computing the same value multiple times.
One way of doing DP is to store the previous values in a data structure (we will use lists) which is refereed to as memo. Thus, this method is also called memoization.

While smart in its own right, we will use a more involved method for this.

```
ping :: Integer -> Integer
ping 0 = 1
ping 1 = 1
ping n = dp !! (n-1) + dp !! (n-2)
dp = [ping n | n <- [0..]]
```

This may seem like worse, but it is equally good as **haskell is lazy**. The elements of the list are not calculated till required and even when called, haskell only calculates the called element, and no others.

Idea

While there are many ways to solve DP problems, the simplest and most easy to do way is to write a brute force recursion and then memoize.

2 Frogs

Problem

There are N stones, numbered $1, \ldots, N$. For each $1 \leq i \leq N$, the height of Stone i is h_i.
There is a frog who is initially on Stone 1. He will repeat the following action some number of times to reach Stone N: If the frog is currently on Stone i, jump to Stone $i+1$ or Stone $i+2$. Here, a cost of $|h_j - h_i|$ is incurred, where j is the stone to land on.
Find the minimum possible total cost incurred before the frog reaches Stone N.

Let's begin with the most brute force technique.
If I am at stone i, my cost to travel to N, ie `cost i` will be minimum of $|h_i - h_{i+1}|$ `+ cost (i+1)` and $|h_i - h_{i+2}|$ `+ cost (i+2)`.
Let's now write it in code.

```
minCost :: [Int] -> Int -> Int
minCost heights n = go n where
    go 1 = 0
```

```
    go 2 = abs (heights !! 1 - heights !! 2)
    go n = min (go (n-1) + abs (heights !! (n-1) - heights !! n
         )) (go (n-2) + abs (heights !! (n-2) - heights !! n ))
```

This code has exponential time complexity. While, we will not compute it explicitly, it is clearly not optimal.
This code surpasses the haskell playground time limit at $N = 39$.
So how do we do it better? We first look at the dynamic variables. What is causing the values to change? In this case, it is n.
So we now start storing values. This looks like:

```
minCost :: [Int] -> Int -> Int
minCost heights n = go n where
    go 1 = 0
    go 2 = abs (heights !! 1 - heights !! 2)
    go n = min (gom !! (n-2) + abs (heights !! (n-1) - heights
        !! (n-2) )) (gom !! (n-3) + abs (heights !! (n-3) -
        heights !! (n-1) ))
    gom = [go x | x <- [1.. length(heights)]]
```

Here `gom` is memoizing due to haskell's lazy computation. The is list is extended only when called for and it is only called for when the list doesn't have the required index.
This brings the time-complexity down straight to $O(n)$.[1]

Idea

Use lists to memoize lazily.
Create a list with the values you need in the order they are generated in, this is important. If your list is in reverse of the generation order, you actually increase the time complexity.
That's all. With a little maneuvering here and there, one can use this everywhere.

3 Longest Common Subsequence

Problem

Given two strings, S1 and S2, the task is to find the length of the Longest Common Subsequence. If there is no common subsequence, return 0. Moreover, find a common subsequence of the above length. Return "" if there is no common subsequence.

The simple recursion solution is

[1] On Haskell Playground, this got me up to $N \geq 10000$.

```
    lcsCount :: String -> String -> Int
    lcsCount "" _ = 0
    lcsCount _ "" = 0
    lcsCount (x:xs) (y:ys) = if x == y then 1 + lcsCount xs ys
        else max (lcsCount (x:xs) ys) (lcsCount xs (y:ys))

    lcsString :: String -> String -> String
    lcsString "" _ = ""
    lcsString _ "" = ""
    lcsString (x:xs) (y:ys) =
        |x == y  = x : lcsString xs ys
        |length ((lcsString (x:xs) ys)) >  length ((lcsString
            xs (y:ys))) = lcsString (x:xs) ys
        |otherwise lcsString xs (y:ys)
```

This has time complexity of $O\left(2^{\min(m,n)}\right)$.
But with no numbers anywhere in sight, how do we even even memoize it?
Let's write the code a little different...

```
lcsCount :: String -> String -> Int
lcsCount s1 s2 = go (length s1) (length s2)
  where
    go 0 _ = 0
    go _ 0 = 0
    go i j
      | s1 !! (i - 1) == s2 !! (j - 1) = 1 + go (i - 1) (j - 1)
      | otherwise = max (go i (j - 1)) (go (i - 1) j)
```

Idea

We can use our DP idea if we have numbers. While strings are not numbers, there indecies are.
So in such cases, we simply use treat the strings as global variables and then make a numberical code running on them.

Well, now we can simply use out list idea.

```
lcsCount :: String -> String -> Int
lcsCount s1 s2 = go (length s1) (length s2)
  where
  go 0 _ = 0
  go _ 0 = 0
  go i j
    | s1 !! (i - 1) == s2 !! (j - 1) = 1 + gom !! (i - 1) !! (j
        - 1)
    | otherwise = max (gom !! i !! (j - 1)) (gom !! (i - 1) !!
       j)
  gom = [[go m n | n <- [0..(length s2)]] | m <- [0..(length s1
     )]]
```

Let's now solve for `lcsString`. The alternate code is

```
lcsString :: String -> String -> String
lcsString s1 s2 = go (length s1) (length s2)
  where
    go 0 _ = ""
    go _ 0 = ""
    go i j
      | s1 !! (i - 1) == s2 !! (j - 1) = go (i - 1) (j - 1) ++
          [s1 !! (i - 1)]
      | length (go i (j - 1)) > length (go (i - 1) j) = go i (j
          - 1)
      | otherwise = go (i - 1) j
```

So we can now use our list memoization.

```
lcsString :: String -> String -> String
lcsString s1 s2 = go (length s1) (length s2)
  where
    go 0 _ = ""
    go _ 0 = ""
    go i j
      | s1 !! (i - 1) == s2 !! (j - 1) = gom !! (i - 1) !! (j -
          1) ++ [s1 !! (i - 1)]
      | length (go i (j - 1)) > length (go (i - 1) j) = gom !!
          i !! (j - 1)
      | otherwise = gom !! (i - 1) !! j
    gom = [[go m n | n <- [0..(length s2)]] | m <- [0..(length
        s1)]]
```

And we are done.
While the naive code can't even deal with two strings of length 15, our improved code goes up-to 465.[2]

4 Edit Distance

Problem

Given two strings `word1` and `word2`, return the minimum number of operations required to convert `word1` to `word2`.
You have the following three operations permitted on a word:

- Insert a character - 2 cost
- Delete a character - 2 cost
- Replace a character - 1 cost

[2] On haskell playground.

Given two words, figure out the cost to convert from the first word to other. Example:
`Input: word1 = "horse", word2 = "ros"`
`Output: 5`
Explanation:
`horse -> rorse (replace 'h' with 'r')`
`rorse -> rose (remove 'r')`
`rose -> ros (remove 'e')`
`1+2+2 = 5`

Here the naive code is a bit harder to guess, but here is the idea.
The base cases are easy to guess, if the first word is empty, we just insert letters till we have the second word.
If the second word is empty, we delete all the letters till we empty it as well.
Now, if two words have the same letter at the same position, we can simply keep them as it is and let it go.
If they are different, say x and y we have some options.

```haskell
editDistance :: String -> String -> Int
editDistance "" s2 = length s2 * 2  -- Cost of inserting all
    characters of s2
editDistance s1 "" = length s1 * 2  -- Cost of deleting all
    characters of s1
editDistance (x:xs) (y:ys)
    | x == y   = editDistance xs ys  -- No cost if last
        characters are the same
    | otherwise = minimum [ insertCost, deleteCost, replaceCost
        ]
  where
    insertCost = editDistance (x:xs) ys + 2        -- Insert y
        before x in x:xs
    deleteCost = editDistance xs (y:ys) + 2        -- Delete x
        from x:xs
    replaceCost = editDistance xs ys + 1        -- Replace x
        with y
```

With our naive solution in place, we can use the logic from LCS problem to get the following code.

```haskell
editDistance :: String -> String -> Int
editDistance s1 s2 = go (length s1) (length s2)
  where
    go 0 j = 2*j  -- Cost of inserting all characters of s2
    go i 0 = 2*i  -- Cost of deleting all characters of s1
    go i j
        | s1 !! (i-1) == s2 !! (j-1) = go (i-1) (j-1)  -- No
            cost if last characters are the same
        | otherwise = minimum [ go i (j-1) + 2,   -- Insertion
            cost
```

```
                                    go (i-1) j + 2,    -- Deletion
                                        cost
                                    go (i-1) (j-1) + 1 --
                                        Substitution cost
                                ]
```

And from here, getting to a DP is just making the memo.

```
editDistance :: String -> String -> Int
editDistance s1 s2 = go m n
  where
    m = length s1
    n = length s2
    go 0 j = 2*j  -- Cost of inserting all characters of s2
    go i 0 = 2*i  -- Cost of deleting all characters of s1
    go i j
        | s1 !! (i-1) == s2 !! (j-1) = gom !! (i-1) !! (j-1)
            -- No cost if last characters are the same
        | otherwise = minimum [ gom !! i !! (j-1) + 2,    --
            Insertion cost
                                    gom !! (i-1) !! j + 2,    --
                                        Deletion cost
                                    gom !! (i-1) !! (j-1) + 1 --
                                        Substitution cost
                                ]
    gom = [[go i j | j <- [0..n]] | i <- [0..m]]
```

Time Complexity

The naive cases both have a time complexity of $O\left(3^{\min(m,n)}\right)$ where m and n are the length of the words.
The DP case brings the time-complexity down to $O(m * n)$.

5 Partitions

Problem

Given a set of integers, the task is to divide it into two sets S1 and S2 such that the absolute difference between their sums is minimum.
If there is a set S with n elements, then if we assume Subset1 has m elements, Subset2 must have n-m elements and the value of abs(sum(Subset1) – sum(Subset2)) should be minimum.
Examples
`Input: [1, 6, 11, 5]`
`Output: 1`
Explanation: `S1 = [1, 5, 6], sum = 12, S2 = 11, sum = 11,`

```
Absolute Difference (12 - 11) = 1
Input: [1, 5, 11, 5]
Output: 0
Explanation:    S1 = [1, 5, 5], sum = 11, S2 = [11], sum = 11,
Absolute Difference (11 - 11) = 0
```

Here instead of a string we have been given a list of integers. Our stratergy, however, remains the same.
The naive code at every index considers what would happen if the element at that index would be in $S1$ or $S2$ and takes the case which reduces the difference between sums.
We are basically generating all the subsets of the given list.

```haskell
minSum :: [Int] -> Int
minSum [] = 0
minSum [a] = a
minSum xs = go (length xs - 1) 0
  where
    sumTotal = sum xs
    go :: Int -> Int -> Int
    go 0 h = abs (2*h - sumTotal)
    go n h = min (go (n-1) (h+ (xs !! n))) (go (n-1) h)
```

Memmoizing it is just a formality from here.

```haskell
minSum :: [Int] -> Int
minSum [] = 0
minSum [a] = a
minSum xs = gom !! (length xs - 1) !! 0
  where
    sumTotal = sum xs
    go :: Int -> Int -> Int
    go 0 h = abs (2*h - sumTotal)
    go n h = min (gom !! (n-1) !! (h+ (xs !! n))) (gom !! (n-1)
         !! h)
    gom = [[go n h | h <- [0..]]| n <- [0..]]
```

A harder question here would be about the time complexity of the memoized code.

Time Complexity

An rather simple way of guessing the worst case time complexity (the O thingy) of a memoized code is to look at the number of elements in the memoized list. Basically, we may have some worst case where every element is accessed.
Note: This works as calculating each entry in memo takes constant time

due to memoization and we can 'ignore' constents in time complexity. You can confirm that this is true for all the codes we saw till now. However, this doesn't work here as we are dealing with an infinite list of infinite list. The above method would indicate that this code never terminates, which is obviously untrue. So what do we do here?
In this case, we ask till where can we actually generate the lists?
`h` is at worst equal to sum of all elements of the given list. `n` is at worst equal to the number of elements in the list.
This gives us the time complexity $O(n * S)$ where n is the number if elements in the list and S is the sum of these elements.

6 0-1 Knapsack

Problem

Given N items where each item has some weight and profit associated with it and also given a bag with capacity W, [i.e., the bag can hold at most W weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible.

Possibly the most famous problem in DP. The naive solution

```
knapsack :: [(Int, Int)] -> Int -> Int
knapsack [] _ = 0
knapsack _ 0 = 0
knapsack ((a,b):xs) w
  | w < a     = knapsack xs w  -- Item can't be included if its
      weight is more than available capacity
  | otherwise = max (b + knapsack xs (w - a)) (knapsack xs w)
```

Making it better is equally simple

```
knapsack :: [(Int, Int)] -> Int -> Int
knapsack xs bag = go (length xs) bag where
    weights = map fst xs
    values = map snd xs
    go :: Int -> Int -> Int
    go 0 _ = 0
    go _ 0 = 0
    go i w
        | w < (weights !! (i-1)) = dp !! (i-1) !! w
        | otherwise = max (values !! (i-1) + dp !! (i-1) !! (w
            - weights !! (i-1)))
                          (dp !! (i-1) !! w)

    dp = [ [ go i w | w <- [0..bag] ] | i <- [0..(length xs)] ]
```

So why did we even study this simple thing? Because Knapsack is not a single problem, it is a class of problems. We can solve a lot of problems by making only cosmetic changes to this algorithm. Two famous examples are left as exercise.

Problem

Given a rod of length n inches and and a list of prices that includes prices of all pieces of size smaller than n. Determine the maximum value obtainable by cutting up the rod and selling the pieces. We are mainly given a price array such that the size of the price array is the same as rod length, and `price !! i` represents the price of length $i+1$.
`Input: price = [1, 5, 8, 9, 10, 17, 17, 20], length = 8`
`Output: 22`
Explanation: The rod length is 8, and the values of different pieces are given as follows:

Length	1	2	3	4	5	6	7	8
Price	1	5	8	9	10	17	17	20

The maximum obtainable value is 22 (by cutting the rod as $8 = 2 + 6$)

Problem

Given an list `coins` representing different types of denominations and an sum, the task is to count all combinations of coins to make a given value sum.
Note: Assume that you have an infinite supply of each type of coin.
Examples:
`Input: sum = 4, coins = [1,2,3]`
`Output: 4`
Explanation: there are four solutions: $1+1+1+1; 1+1+2; 2+2; 1+3$
`Input: sum = 10, coins = [2, 5, 3, 6]`
`Output: 5`
Explanation: There are five solutions: $2+2+2+2+2; 2+2+3+3; 2+2+6; 2+3+5; 5+5$
`Input: sum = 10, coins = [10]`
`Output: 1`
Explanation: The only is to pick 1 coin of value 10.
`Input: sum = 5, coins = [4]`
`Output: 0`
`Explanation: We cannot make sum 5 with the given coins`

7 A Game

Problem

Consider a row of N coins of values $V_1, \ldots, V_n$, where N is even. We play a game against an opponent by alternating turns. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Determine the maximum possible amount of money we can definitely win if we move first.
Note: The opponent is as clever as the user.
Example:

```
Input: [5, 3, 7, 10]
Output: 15 -> (10 + 5)
Input: [8, 15, 3, 7]
Output: 22 -> (7 + 15)
```

This problem will require us to work a bit more creatively.
Instead of making two functions, one to model our move and the other to model the opponents moves, we can make a singular function. This is a zero-sum game That means, our opponents best move is one which reduces our ideal payoff. So I only need to think of our move.
Doing so naively looks like

```
payOut :: [Int] -> Int
payOut [] = 0
payOut [a] = a
payOut xs = go 0 (length xs - 1) -- Here the second and third
    arguments are just the indexes we start and end at
    where
        go i j
            | i > j  = 0
            | i == j = xs !! i
            | otherwise = let
            takeFirst = xs !! i + min (go (i+2) j) (go (i+1) (j
                -1)) -- The opp may take from the front or the
                back. As they play optimally, they try to
                minimize our winnings.
            takeLast = xs !! j + min (go (i+1) (j-1)) (go i (j
                -2)) in
            max takeFirst takeLast
```

The memoization will take a bit more creativity here. Direct memoization creates errors as `i+2`, `j-1` or `j-2` may not be a logical index.
So what do we do? Create another function to ensure safe access of the list.

```
payOut :: [Int] -> Int
payOut [] = 0
```

```
payOut [a] = a
payOut xs = go 0 (length xs - 1) -- Here the second and third
    arguments are just the indexes we start and end at
    where
      go i j
        | i > j  = 0
        | i == j = xs !! i
        | otherwise = let
          takeFirst = xs !! i + min (safeAccess (i+2) j) (
              safeAccess (i+1) (j-1))
          takeLast  = xs !! j + min (safeAccess (i+1) (j-1)) (
              safeAccess i (j-2))
        in
          max takeFirst takeLast
      gom = [[go i j | j <- [0..(length xs - 1)]] | i <- [0..(
          length xs - 1)]]

      safeAccess :: Int -> Int -> Int
      safeAccess i j
        | i > j = 0
        | i == j = xs !! i
        | otherwise = gom !! i !! j
```

Finally, our code happens to have yet another optimization, we can store the data of the game in double accessed list or a 'dequeue'.
While this doesn't change the asymptotic complexity (the O thingy) it does make it better on average for obvious reasons.
While utilizing is left as exercise, we have included the code for dequeue at the end.

8 Google Interview: Subset Sum

Problem

Given a list of integers `S` and a target number `k`, write a function that returns a subset of `S` that adds up to `k`. If such a subset cannot be made, then return `Nothing`.
Integers can appear more than once in the list. You may assume all numbers in the list are positive.
For example, given `S = [12, 1, 61, 5, 9, 2]` and `k = 24`, return `[12, 9, 2, 1]` since it sums up to 24.

This question was asked in a google interview and was considered hard by a good bunch of people.
Let's hope you don't end up joining that bunch.
The naive solution is slightly difficult to get directly, so here is my attempt to explain it with comments.

```haskell
subset :: [Int] -> Int -> Maybe [Int]
-- Base case: if `k` is 0, the subset is empty (since the sum
    of an empty subset is 0)
subset _ 0 = Just []
-- Base case: if the list is empty but `k` is not 0, no valid
    subset can be found
subset [] _ = Nothing
-- Main recursive case: try to find a subset by invoking the
    helper function `go`
subset s k = go s k []
    where
        -- Helper function `go` performs the actual recursive
            search
        -- `go s k ans` will return `Just ans` if a subset
            summing to `k` is found
        -- `s` is the remaining list of integers to consider
        -- `k` is the target sum we are trying to achieve
        -- `ans` is the current list of integers that form a
            partial subset
        go [] 0 ans = Just ans  -- If we've successfully summed
             up to 0, return the current subset
        go [] _ _ = Nothing     -- If the list is empty but we
            haven't reached 0, return Nothing (no valid subset)

        -- Recursive case: consider each element of the list `s
            ` (starting from `x:xs`)
        go (x:xs) k ans
            -- If `k` is negative, we have gone over the target
                , so return Nothing
            | k < 0 = Nothing
            -- Check if it's possible to find a subset by
                excluding `x` (don't add `x` to `ans`)
            -- This checks the "without the current element"
                scenario by calling `go xs k ans`
            -- If that returns `Nothing`, proceed to try adding
                 `x` to `ans` and call `go xs (k - x) (x : ans)`
            | go xs k ans == Nothing = go xs (k - x) (x : ans)
            -- If the previous step fails, just keep the subset
                 as is and continue with the next elements
            | otherwise = go xs k ans
```

And for your viewing pleasure, here is the code without the comments.

```haskell
subset :: [Int] -> Int -> Maybe [Int]
subset _ 0 = Just []
subset [] _ = Nothing
subset s k = go s k []
    where
        go [] 0 ans = Just ans
        go [] _ _ = Nothing
```

```
        go (x:xs) k ans
        | k < 0 = Nothing
        | go xs k ans == Nothing  = go xs (k-x) (x:ans)
        | otherwise = go xs k ans
```

So how do we memoize this? Clearly, we work with the index of elements of `s`. The naive, dp-able, version is

```
join :: a -> Maybe [a] -> Maybe [a]
join _ Nothing = Nothing
join x (Just xs) = Just (x:xs)

subset :: [Int] -> Int -> Maybe [Int]
subset _ 0 = Just []
subset [] _ = Nothing
subset s k = go (length s) k
    where
      go _ 0 = Just []
      go 0 _ = Nothing
      go i j
        |j < 0 = Nothing
        |go (i-1) j == Nothing = let x = s !! (i-1) in  join x
            (go (i-1) (j-x))
        |otherwise = go (i-1) j
```

And then we memoize it and secure a 'hypothetical' job at google

```
join :: a -> Maybe [a] -> Maybe [a]
join _ Nothing = Nothing
join x (Just xs) = Just (x:xs)

subset :: [Int] -> Int -> Maybe [Int]
subset _ 0 = Just []
subset [] _ = Nothing
subset s k = go (length s) k
    where
      go _ 0 = Just []
      go 0 _ = Nothing
      go i j
        |j < 0 = Nothing
        |gom !! (i-1) !! j == Nothing = let x = s !! (i-1) in
            if (j-x) > 0 then join x (gom !! (i-1) !! (j-x))
            else Nothing
        |otherwise = gom !! (i-1) !! j
      gom = [[go a b | b <- [0..k]]| a <- [0..(length s)]]
```

9 When not to memoize...

Problem

The following iterative sequence is defined for the set of positive integers:
$n \to n/2$ (n is even)
$n \to 3n + 1$ (n is odd)
Using the rule above and starting with 13, we generate the following sequence:

$$13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1.$$

It can be seen that this sequence (starting at 13 and finishing at 1) contains 10 terms. Although it has not been proved yet (Collatz Problem), it is thought that all starting numbers finish at 1. Which starting number, under some input N, produces the longest chain?
NOTE:Once the chain starts the terms are allowed to go above N.

This question starts like any other, we start by simply writing a naive code.

```
colletz :: Int -> Int
colletz n = snd (maximum [(colletzLength x , x) | x <- [1..n]])
    where
        colletzLength 1 = 0
        colletzLength a = if even a then 1 + colletzLength (a `
            div` 2) else 1 + colletzLength (3*a + 1)
```

Time Complexity

While we can't really say much about the time complexity here as the Colletz conjuncture is unsolved at the moment.
And the closed form of length of colletz chain is still another issue.
So what do we do here?
Each call to `colletzLength` can be thought of as having a complexity that might be close to $O(\log a)$ on average for many numbers, but this can vary greatly.
This is not a very good approximation, we are guessing way below. But what else can we do?
Since `colletz` calls `colletzLength` for every number up to n, we can simply say all numbers cause $\log n$ amount of operations.
Thus, in practice, the overall time complexity can be estimated as:

$$O(n \cdot \log n)$$

This is a rough estimate that assumes each call to `colletzLength` on average takes $O(\log n)$ time, leading to a total of $O(n \cdot \log n)$.
NOTE: The actual time can vary significantly based on the character-

istics of the Collatz sequence for each number, and while $O(n \log n)$ is a reasonable estimate, the worst-case behavior could be worse due to the undefined behavior of the sequence lengths for larger inputs.

As $O(n \cdot \log n)$ is anyhow a good time-complexity, one can estimate this code to work till some point. And it indeed does work for even a million.
However, can we make it much better by memoization?

```
colletz :: Int -> Int
colletz n = snd (maximum [(colletzLength x , x) | x <- [1..n]])
    where
        colletzLength 1 = 0
        colletzLength a = if even a then 1 + colletzLengthm !!
            (a `div` 2 - 1) else 1 + colletzLengthm !! (3*a)
        colletzLengthm = [colletzLength x | x <- [1..]]
```

But surprise, surprise, this only works till 9500.

Time Complexity

The memoization makes the `colletzLength` a $O(n)$ operation. This makes `colletz` an $n * O(n) = O(n^2)$ process.

So what do we do? We don't memoize all the time.
In case of `colletz`, memoization doesn't really help as it is already on an average an $O(\log n)$ operation.

Idea

Memoization works to bring exponential time complexity down to polynomial.
Memoization will **never** give something sub-linear.
Do not memoize things which are already sub-linear. It won't end well.

10 Why does this handout exist?

Mainly as I was somewhat sad that most people, including my batch mates, TAs and most of the internet dispised DP in haskell.
One of the first things I noticed while trying out the class examples was the technique of lists detailed in this handout. I was somewhat shocked to not find a good treatment of this technique anywhere else (the only somewhat decent one being the one on Haskell wiki). So I decided to make this handout.
Every other place and person was quick to jump to arrays or maps or monads or

the memo package which is basically overkill for most DP problems.This frustrated me even more eventually motivating me to speand way more time than neccesary on this.
The problems covered were from the Geek For Geek list of "Top 20 Dynamic Programming Interview Questions". This source was chosen to illustrate that the most popular problems are easily doable.

I hope you enjoyed this short journey through list only DP. As a great man once said, this is the hack job by a novice. Please feel free to use or discard any part of this. If possible, report any errors or feedback to arjuna.ug2024@cmi.ac.in.
Thanks for reading.

Appendix: Dequeue

```haskell
data Deque a = Deque [a] [a]
emptyDeque :: Deque a
emptyDeque = Deque [] []

-- Add an element to the front
pushFront :: a -> Deque a -> Deque a
pushFront x (Deque front back) = Deque (x:front) back

-- Add an element to the back
pushBack :: a -> Deque a -> Deque a
pushBack x (Deque front back) = Deque front (x:back)

-- Remove an element from the front
popFront :: Deque a -> Maybe (a, Deque a)
popFront (Deque [] []) = Nothing  -- empty deque
popFront (Deque (x:xs) back) = Just (x, Deque xs back)
popFront (Deque [] back) = popFront (Deque (reverse back) [])

-- Remove an element from the back
popBack :: Deque a -> Maybe (a, Deque a)
popBack (Deque [] []) = Nothing  -- empty deque
popBack (Deque front (x:xs)) = Just (x, Deque front xs)
popBack (Deque front []) = popBack (Deque [] (reverse front))
```