ADA Design and Analysis 2023

Generated from answers.lhs and Dist.hs on 2023-07-20

```
import Control.Exception (assert) -- For checking solutions
import Dist (dist, tabulate, fromList) -- For 1.b.iii and 1.c.iii
import Data.Array -- For 1.c.iii
```

1.a.i)

Define `palindrome xs` to return `True` when `xs` is a palindrome, and `False` otherwise. State the complexity of your function.

```
palindrome :: String -> Bool
palindrome xs
  = xs == (reverse xs)
```

Complexity is $O(n)$ where n is string length.

1.a.ii)

Briefly explain why the following properties hold for any string `xs` of length `n`

A) *all the characters in a nearest palindrome to `xs` must be from `xs`*

If we get a nearest palindrome to `xs` that doesn't contain a character in `xs`, we must have added them all in the edit journey. We can therefore remove all of those steps from the edit journey and get a valid palindrome with a shorter edit distance. So we can't have had a nearest palindrome to start off with, and so there can't be a nearest palindrome to `xs` that has characters not in `xs`, and so all nearest palindromes must be made of characters from `xs`.

B) *the edit distance between `xs` and its nearest palindrome is at most floor(n / 2)*

We can always form a valid palindrome of a string `xs` by replacing the second half of the string by the first half reversed. This would take `n` update operations (rounded down, as we can leave the middle character of an odd-length string as it is), so any nearest palindrome must be closer to `xs` than this.

eg. `abcdef` to `abccba` has an edit distance of 3

C) *the length of a nearest palindrome to `xs` is bounded by `n` plus floor(n / 2)*

From B) we have that a nearest palindrome has an edit distance of at most floor(n / 2). The most an update can do to increase the length of a palindrome is to insert a character, increasing length by 1. Therefore, the maximum length that a nearest palindrome to `xs` can have, is the length of `xs`, `n`, plus the maximum edit distance, floor(`n` / 2).

1.b.i)

Define `strings xs n` to produce all the strings of length `n` whose characters are drawn from `xs`. This need not be efficient but its complexity should be bounded by $O(m^n)$ where `m = length xs`.

```
strings :: String -> Int -> [String]
strings _  0 = [""]
strings xs n = [s : st | s <- xs, st <- strings xs (n - 1)]
```

1.b.ii)

Using the `strings` function, define `palindromes xs` to return all the palindromes than can be formed from `xs`. Hint: consider the properties of nearest palindromes and filter appropriate strings with the `palindrome` function.

```
palindromes :: String -> [String]
palindromes xs
  = (filter palindrome . concatMap (strings xs)) [0..ml]
```

```haskell
  where ml = n + (n `div` 2)
        n  = length xs
```

1.b.iii)

Using `palindromes`, define `palindist xs` to calculate the edit distance between `xs` and its nearest palindromes. For example, `palindist "abXcYbZ" = 2`. You may assume `dist :: String -> String -> Int`. This need not be efficient.

```haskell
palindist :: String -> Int
palindist xs
  = minimum (map (dist xs) (palindromes xs))
```

1.c.i)

Consider how `palindist "abcba"` relates to the result of applying `palindist` to the following strings: `"abcbaX"`, `"Xabcba"`, `"XabcbaX"`, `"XabcbaY"`.

Using this relationship, define `palindist'` a recursive version of `palindist`.

This explanation isn't required for the answer. The value of `palindist "abcbaX"` is one more than the value of `palindist "abcba"` as we can remove the `X` at the end to get to `"abcba"`. Similarly, the value of `palindist "Xabcba"` is one more than the value of `palindist "abcba"`. The value of `palindist "XabcbaX"` is equal to the value of `palindist "abcba"`, because the two `X`'s are equal and so are palindromic. The value of `palindist "XabcbaY"` is one more than `palindist "abcba"`, as we can change the character `Y` to `X`. We can check all these cases, and find the minimum amongst them (after adding the extra cost), to define `palindist` recursively.

```haskell
palindist' :: String -> Int
palindist' []  = 0
palindist' [_] = 0
palindist' xs
  = minimum [palindist' (tail xs) + 1,
             palindist' (init xs) + 1,
             palindist' (tail (init xs)) +
               if (head xs) == (last xs)
                 then 0 else 1]
```

1.c.ii)

Consider why strings make bad indices. Complete the definition of `palindist''`, which is a recusive version of `palindist'` that uses indices `i` and `j`:

This explanation isn't required for the answer. Haskell strings make bad indices because removing the last element takes linear time.

```haskell
palindist'' :: String -> Int
palindist'' xs = go 0 (length xs - 1)
  where go :: Int -> Int -> Int
        go x y
          | x >= y    = 0
          | otherwise = minimum [(go (x + 1) y) + 1,
                                 (go x (y - 1)) + 1,
                                 (go (x + 1) (y - 1)) +
                                   if xs !! x == xs !! y
                                     then 0 else 1]
```

1.c.iii)

Define `palindist'''`, an efficient version of `palindist''` that uses dynamic programming. You may use `tabulate :: Ix i => (i, i) -> (i -> a) Array i a`, which takes a range of indices and a function and *creates an array by tabulating the function and* ***fromList :: [a] -> Array Int a****, which returns an array whose elements are from a list.*

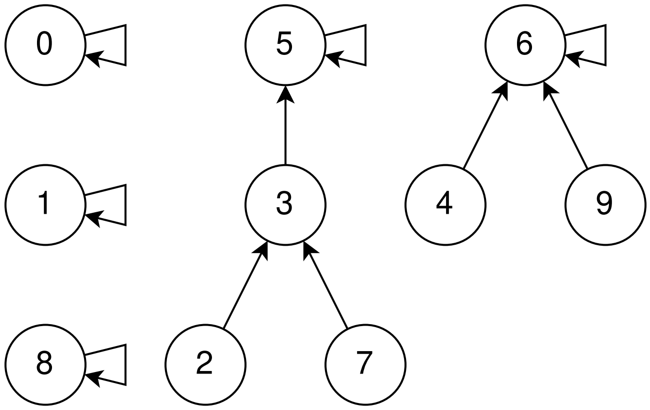
```
palindist''' :: String -> Int
palindist''' "" = 0
palindist''' xs = arr ! (0, end)
  where end = length xs - 1
        xsArr :: Array Int Char
        xsArr = fromList xs

        arr :: Array (Int, Int) Int
        arr = tabulate ((0, 0), (end, end)) memo

        memo :: (Int, Int) -> Int
        memo (x, y)
          | x >= y    = 0
          | otherwise = minimum [(arr ! (x + 1, y)) + 1,
                                 (arr ! (x, y - 1)) + 1,
                                 (arr ! (x + 1, y - 1)) +
                                   if xsArr ! x == xsArr ! y
                                     then 0 else 1]
```

2.a.i)

Consider the case when `n` *is 10, and the parent list is* $[0, 1, 3, 5, 6, 5, 6, 3, 8, 6]$*. Draw a graph where the nodes are all the numbers 0 to 9 and an edge froms from a child to its parent. Write the origin and family of each element.*



Element	Origin	Family
0	0	0
1	1	1
2	5	5, 3, 2, 7
3	5	5, 3, 2, 7
4	6	6, 4, 9
5	5	5, 3, 2, 7
6	6	6, 4, 9
7	5	5, 3, 2, 7
8	8	8
9	6	6, 4, 9

2.a.ii)

Briefly explain why every element $x \in \{0, \ldots, n-1\}$ *has an origin.*

We have a finite number of nodes in this parent list, so an ancestors list for `x` must visit a node already in the ancestors list at some point. If the node that first visits a node in the ancestors list, `a`, has itself as its parent, it is `x`'s origin. If it has some other node in the ancestors list as its parent, `b`, then the parent list isn't valid, as `a` is in the ancestors list for `b`, and `a` is the parent of `b`.

Therefore, every element in a valid parent list has an origin.

2.a.iii)

Define a function `ancestors ps x`*, which returns the ancestors of* `x`*. Use this to define* `origin ps x`*, which returns the origin of* `x`*.*

You may assume the existence of function `(!) :: [a] -> Int -> a`*, where* `xs ! i` *returns the ith element of* `xs` *in constant time.*

Note: I can't give another definition of `!` without it conflicting with the one for arrays we need for the dynamic programming question, so I've used `¡` instead. **Imagine it works in constant time.**

```
(¡) :: [a] -> Int -> a
xs ¡ i = xs !! i -- but like, it's acc quick
```

Then we'll define `ancestors` and `origin`:

```haskell
ancestors :: [Int] -> Int -> [Int]
ancestors ps x
  | x' == x   = [x]
  | otherwise = x : (ancestors ps x')
  where x' = ps ¡ x
```

```haskell
origin :: [Int] -> Int -> Int
origin ps x = last (ancestors ps x)
```

2.a.iv)

Given an element `x`, define `family ps x` to return a list of all the elements in the family of `x`. State the worst-case complexity of your function.

```haskell
family :: [Int] -> Int -> [Int]
family ps x
  = filter (\p -> xo == origin ps p) [0..(length ps - 1)]
  where xo = origin ps x
```

Worst-case complexity is $O(n^2)$

2.a.v)

Given two elements `x` and `y`, define `adopt ps x y` to return the list `ps` modified so that if `xo` is the origin of `x`, and `yo` is the origin of `y`, then the origin `xo` or `yo` that has the biggest family will become the parent of the other origin. If the families are the same size, then `xo` becomes the parent of `yo`. This need not be efficient.

You may assume the existence of `update :: [a] -> Int -> a -> [a]`, where `update xs i x` returns the list `xs` modified so that `xs ! i = x` in constant time

I'll give a definition of `update` here to make the code work. You'll have to **imagine it works in constant time**.

```haskell
update :: [a] -> Int -> a -> [a]
update [] _ _ = error "Update index out of range"
update (o : xs) i x
  | i == 0    = x : xs
  | otherwise = o : (update xs (i - 1) x)
```

Then we can define `adopt`:

```haskell
adopt :: [Int] -> Int -> Int -> [Int]
adopt ps x y
  = if length xf >= length yf
      then update ps yo xo
      else update ps xo yo
  where xo = origin ps x
        yo = origin ps y
        xf = family ps x
        yf = family ps y
```

2.b.i)

Modify your definitions so that `adopt` is more efficient by avoiding the recalculation recalculation of `family`. You will have to change the type of the parent list to accomodate extra information

I'm going to change the parent list data structure so it is a pair, containing both the parent of this node and the size of the family.

```haskell
type Element = (Int, Int)
type PList = [Element]
```

We can convert the old representation, by running family on each one, and storing the length of the family in each element.

```
toPList :: [Int] -> PList
toPList ps = zipWith (\p i -> (p, length (family ps i))) ps [0..]
```

We then modify `ancestors` and `family` to pattern match out the parent before following it. We also need a new `origin` to handle `PList`s.

```
ancestors' :: PList -> Int -> [Int]
ancestors' ps x
  | x' == x   = [x]
  | otherwise = x : (ancestors' ps x')
  where (x', _) = ps ¡ x
```

```
origin' :: PList -> Int -> Int
origin' ps x = last (ancestors' ps x)
```

```
family' :: PList -> Int -> [Int]
family' ps x
  = filter (\p -> xo == origin' ps p) [0..(length ps - 1)]
  where xo = origin' ps x
```

Then we can update `adopt` such that it uses the value stored in the element rather than calling `family`.

```
adopt' :: PList -> Int -> Int -> PList
adopt' ps x y
  = update (update ps xo e') yo e'
  where xo = origin' ps x
        yo = origin' ps y
        (_, xf) = ps ¡ xo
        (_, yf) = ps ¡ yo
        e' = if xf >= yf
          then (xo, xf + yf)
          else (yo, xf + yf)
```

1.b.ii)

`origin` is exactly the same, it just needs redefining on the new type. `ancestors'` produces the list starting with the element passed in, with each subsequent element being the parent of the last. Therefore, the origin is at the end of the list. Its worst-case complexity is $O(n)$

`ancestors'` is changed to pull out the parent from the element, before making the recursive call. It works by adding the current elem to the front of the ancestor list of the parent element, with a base case to handle when we reach an origin. Its worst-case complexity is $O(n)$ in the case where every node is in a big family chain.

`family'` works the same as `family`, only change is to use `origin'` rather than `origin`. Its worst-case complexity is the same, $O(n^2)$.

`adopt'` makes use of the new family size in origin elements so we don't have to make a call to `family`. When we write the new parent to the smaller origin node, we also write the new family size, which we also write to the bigger origin. Its worst-case complexity is now $O(n)$, because we still need to make a call to `origin'` to find `xo` and `yo`.

We only update the size of the family in the origin element because it's faster and it doesn't matter that we don't for non-origins, causing their family size to be wrong over time because there isn't an `unadopt` operation, so an element that isn't an origin can't become an origin, and we only compare sizes of origin elements.

2.b.iii)

Consider a parent list `ps` which is obtained by `k` arbitrary adopt operations on an initial parent list every element is its own parent.

Assuming we mean the new modified definition of `adopt` that uses `PList`. The worst-case complexity of `adopt ps` is $O(k)$. This is because in the worst case, we have to traverse every parent-child relation in a parent list in order to calculate the origin of the `x` and `y`, and `k` adopt operations creates `k` parent-child relations. (ie. in the case where `x` and `y` are at the "bottom" of two family chains that together, use every relation that isn't from an element to itself)

Tests

I've put a couple tests here to check solutions. Some helper values as well:

```haskell
fromPList :: PList -> [Int]
fromPList = map fst

fam = [0,1,3,5,6,5,6,3,8,6]
famPList = [(0,1),(1,1),(3,4),(5,4),(6,3),(5,4),(6,3),(3,4),(8,1),(6,3)]

 main = do
   putStr $ assert (palindrome "abcba") "passed\n"
   putStr $ assert (not (palindrome "abcbef")) "passed\n"
   putStr $ assert (palindist "abc" == 1) "passed\n"
-- Don't run this one, it's _very_ slow
-- putStr $ assert (palindist "abXcYbZ" == 2) "passed\n"
   putStr $ assert (palindist' "abXcYbZ" == 2) "passed\n"
   putStr $ assert (palindist'' "abXcYbZ" == 2) "passed\n"
   putStr $ assert (palindist''' "abXcYbZ" == 2) "passed\n"
   putStr $ assert (ancestors fam 0 == [0]) "passed\n"
   putStr $ assert (ancestors fam 7 == [7, 3, 5]) "passed\n"
   putStr $ assert (origin fam 8 == 8) "passed\n"
   putStr $ assert (origin fam 4 == 6) "passed\n"
   putStr $ assert (adopt fam 0 1 == [0,0,3,5,6,5,6,3,8,6]) "passed\n"
   putStr $ assert (adopt fam 7 4 == [0,1,3,5,6,5,5,3,8,6]) "passed\n"
   putStr $ assert (toPList fam == famPList) "passed\n"
   putStr $ assert (ancestors' famPList 0 == [0]) "passed\n"
   putStr $ assert (ancestors' famPList 7 == [7, 3, 5]) "passed\n"
   putStr $ assert (origin' famPList 8 == 8) "passed\n"
   putStr $ assert (origin' famPList 4 == 6) "passed\n"
   -- we can't compare adopted famPList to toPList of array because there isn't
   -- a canonical representation. ie. PLists only maintain the family value for
   -- origins
   putStr $ assert (map (fst) (adopt' famPList 0 1) == [0,0,3,5,6,5,6,3,8,6]) "passed\n"
   putStr $ assert (map (fst) (adopt' famPList 7 4) == [0,1,3,5,6,5,5,3,8,6]) "passed\n"
```