**Algorithm Design and Analysis 2023**

Correction to 1.b.i announced during exam: in the example, palindist “abXcYbZ” should equal 2, not 3

Aanswers.lhs and Dist.hs contain these answers as working Haskell code that your editor should be able to display with syntax highlighting. Please make changes there too

A picture containing text, person, whiteboard, handwriting

Description automatically generated

1.a.i)

palindrome :: String -> Bool

palindrome xs

= xs == (reverse xs)

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

1.a.ii)

A) 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) 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.

e.g. abcdef to abccba has an edit distance of 3

C) 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)

strings :: String -> Int -> [String]

strings \_ 0 = [""]

strings xs n = [s : st | s <- xs, st <- strings xs (n - 1)]

1.b.ii)

palindromes :: String -> [String]

palindromes xs

= (filter palindrome . concatMap (strings xs)) [0..ml]

where ml = n + (n `div` 2)

n = length xs

1.b.iii)

palindist :: String -> Int

palindist xs

= minimum (map (dist xs) (palindromes xs))

1.c.i)

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)

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)

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,

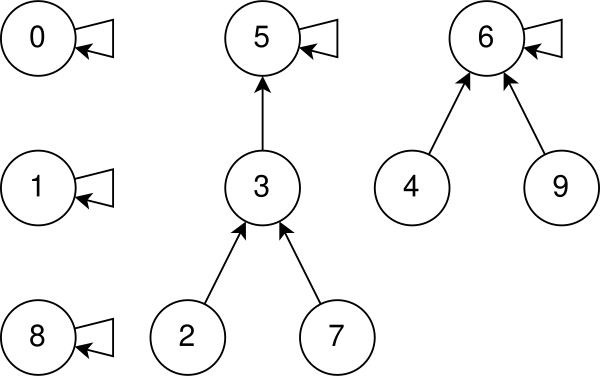
(arr ! (x, y - 1)) + 1,

(arr ! (x + 1, y - 1)) +

if xsArr ! x == xsArr ! y

then 0 else 1]

2.a.i)



|  |  |  |
| --- | --- | --- |
| **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)

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)

ancestors :: [Int] -> Int -> [Int]

ancestors ps x

| x' == x = [x]

| otherwise = x : (ancestors ps x')

where x' = ps ! x

origin :: [Int] -> Int -> Int

origin ps x = last (ancestors ps x)

2.a.iv)

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)

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)

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.

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)

2.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)

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)