50001 - Algorithm Analysis and Design - Lecture 7

Oliver Killane

13/11/21

Lecture Recording

Lecture recording is available here

Dynamic Programming

A technique to efficiently calculate solutions to certain recursive problems.

1. Describe an inefficient recursive algorithm.
2. Reduce inefficiency by storing intermediate shared results.

Fibonacci Sequence

Fully Recursive

```haskell
fib :: Int -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

$$
\begin{aligned}
T_{fib}(0) &= 1 \\
T_{fib}(1) &= 1 \\
T_{fib}(n) &= 1 + T_{fib}(n-2) + T_{fib}(n-1)
\end{aligned}
$$

The complexity of this algorithm is $T_{fib}(n) \in O(2^n)$.

Saving Intermediate Results

We can use a helper function which takes the remaining number of additions, and the two previous values.

```haskell
fib :: Int -> Integer
fib n = fibHelper n 0 1
  where
    fibHelper :: Int -> Integer -> Integer -> Integer
    fibHelper 0 x y = x
    fibHelper n x y = fibHelper (n-1) y (x + y)
```

$$
\begin{aligned}
T_{fib}(0) &= 1 \\
T_{fib}(1) &= 1 \\
T_{fib}(n) &= 1 + T_{fib}(n-1)
\end{aligned}
$$

The complexity of this algorithm is $T_{fib}(n) \in O(n)$.

This way every value is calculated only once for each call. However values are not saved between calls.

Memoisation

```haskell
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

fib :: Int -> Integer
fib n = fibs !! n
```

This creates a large list, we must only index on the list to get the value. By using an array we can reduce the time taken to get to the nth element.

Array Based Memoisation

```haskell
-- Array Data Type
import Data.Array ( Ix(range), Array, array )

-- Tabulate function
tabulate :: Ix i => (i,i) -> (i -> e) -> Array i e
tabulate (a,b) f = array (a,b) [(i,f i) | i <- range (a,b)]

-- Ix (class of all indexes)

-- T(!) is in O(1)
(!) :: Ix i => Array i e -> i -> e

-- Range creates a lits of indexes in a range
range :: Ix i => (i,i) -> [i]

-- Array function creates an array from a range of indexes & values
array :: Ix i => (i,i) -> [(i,e)] -> Array i e
```

Hence we can make our algorithm:

```haskell
import Data.Array ( Array, (!) )

fib :: Int -> Integer
fib n = table ! n
  where
    table :: Array Int Integer
    table = tabulate (0,n) memo

    memo :: Int -> Integer
    memo 0 = 0
    memo 1 = 1
    memo n = table ! (n-2) + table ! (n-1)
```

Here we can do constant time lookups for values in the table. If a value is not present, it is lazily evaluated using other elements in the table.

In this way we only calculate each fibonacci number once, and only when we need it. Further it is saved for any subsequent calls to fib.

Edit-Distance

The **Edit-Distance** Problem is concerned with calculating the **Levenshtein** distance between two strings.

Levenshtein Distance

The number of insertions, deletions & updates required to convert one string into another.

$$toil \rightarrow_{+1} oil \rightarrow_{+1} il \rightarrow_{+1} ill$$

```haskell
dist :: String -> String -> Int
dist xs [] = length xs
dist [] ys = length ys
dist xxs@(x:xs) yys@(y:ys)
  = minimum
    [dist xxs ys + 1,
     dist xs yys + 1,
     dist xs ys + if x == y then 0 else 1]
```

This problem becomes of order $O(3^n)$ as it recurs 3 ways for each call.

We can reuse results for two substrings through memoisation, first we make a new recursive version that uses the index we are checking in each string:

```haskell
dist :: String -> String -> Int
dist xs ys = dist' xs ys (length xs) (length ys)

dist' :: String -> String -> Int -> Int -> Int
dist' xs ys i 0 = i
dist' xs ys 0 j = j
dist' xs ys i j
  = minimum [dist' xs ys i (j-1) + 1,
             dist' xs ys (i-1) j + 1,
             dist' xs ys (i-1) (j-1) + if x == y then 0 else 1]
  where
    m = length xs
    n = length ys
    x = xs !! (m-i)
    y = ys !! (n-j)
```

We can then use **tabulate** to create a memoised version.

```haskell
import Data.Array ( Array, (!) )

dist :: String -> String -> Int
dist xs ys = table ! (m,n)
  where
    table = tabulate ((0,0),(m,n)) (uncurry memo)

    memo :: Int -> Int -> Int
    memo i 0 = i
    memo 0 j = j
    memo i j
      = minimum [table ! (i, j - 1) + 1,
                 table ! (i-1,j) + 1,
                 table ! (i - 1,j - 1) + if x == y then 0 else 1]
      where
        x = ays ! (m - i)
        y = ays ! (n - j)

    m = length xs
    n = length ys
```

```haskell
    axs,ays :: Array Int Char
    axs = fromList xs
    ays = fromList ys
```

As there are at most $m \times n$ entires in the table, and each are calculated at most once, and the lookup time is constant (using arrays), the complexity is $O(mn)$.