LECTURE 7: Dynamic Programming

"A man who dares waste one hour of time has not discovered the value of life"

— Charles Darwin, 1836

Dynamic programming is a technique to improve the complexity of an algorithm:

"trade space for speed".

There are two parts:

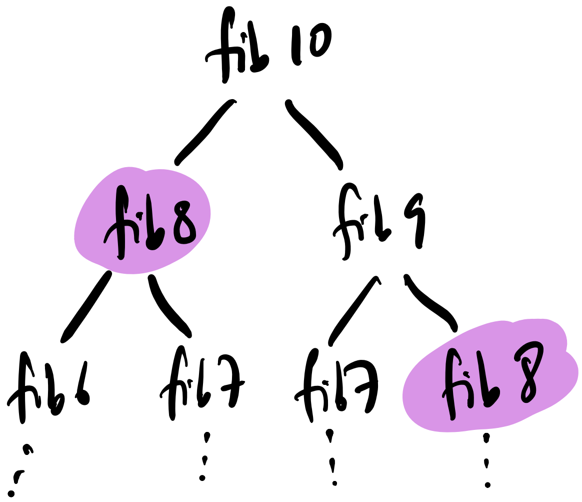
1. Describe an inefficient recursive algorithm
2. Use sharing to improve complexity.

For example:

Fibonacci sequence:

n	0	1	2	3	4	5	6	...
$fib(n)$	0	1	1	2	3	5	8	...

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



$O(2^n)$

[0, 1, 1, 2, 3, 5

We will apply dynamic programming and <u>memoise</u> the results:

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

key/index

value

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

This uses two built in functions:

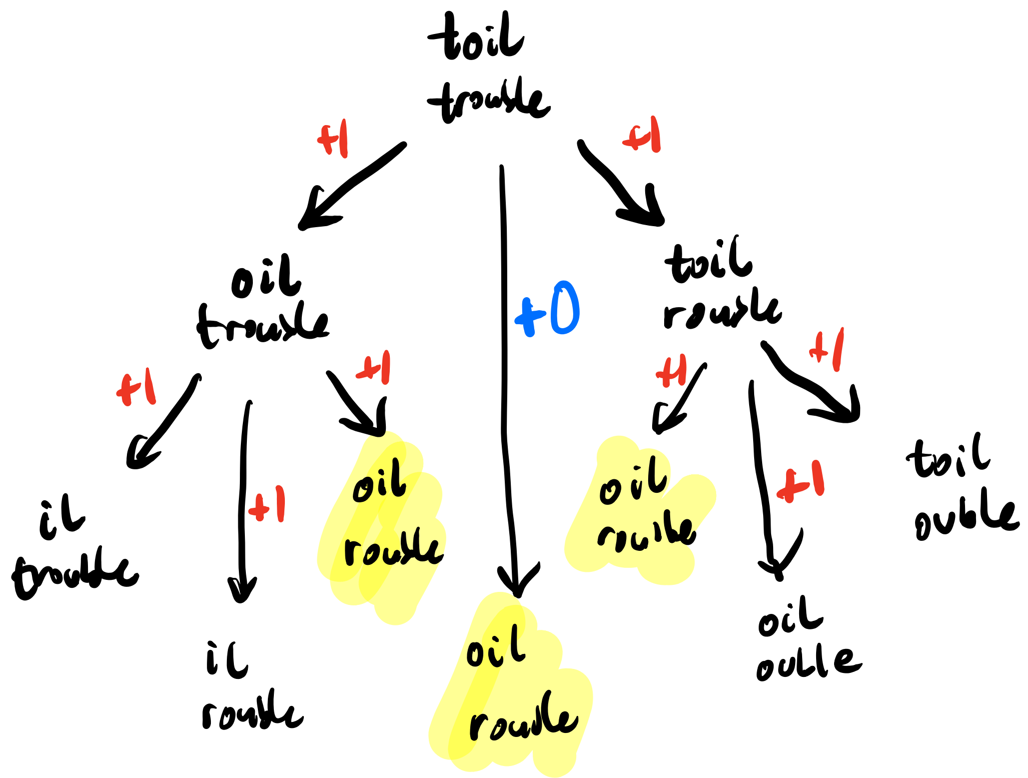
```
> tabulate :: Ix i => (i,i) -> (i -> e) -> Array i e
> tabulate (a,b) f =
>     array (a,b) [ (i, f i) | i <- range (a,b) ]
```

```
-- T(!) or i ∈ O(1)
(!) :: Ix i => Array i e -> i -> e
-- T(array) (a,b) xs ∈ O(n) where n = length xs.
array :: Ix i => (i,i) -> [(i,e)]
```

Edit-distance.

The edit-distance problem finds the Levenstein distance between two strings: the number of inserts, deletes, updates to turn one string into the other.

t o i l ~ t r o u b l e




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

We cannot index on String so we must have a different strategy. Notice that the strings are always suffixes of the original String. We can therefore index using the lengths of the string.

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

                where x = xs!! i
                      y = ys!! j


> dist'' :: String -> String -> Int
> dist'' xs ys  =  table ! (m,n)
>     where
>      table = tabulate ((0,0),(m,n)) memo
>
>      memo  i 0  = i
>      memo  0 j  = j
>      memo  i j  =
>         minimum [table ! (i-1, j) +1
```

>
>
>
>
>
>

$$
\begin{aligned}
&, \ table ! \ (i, j-1) + 1 \\
&, \ table ! \ (i-1, j-1) \\
&\quad + \text{if } x == y \text{ then } 0 \text{ else } 1 \\
&]
\end{aligned}
$$

$x = xs \, !! \, i$
$y = ys \, !! \, j$

where m = length xs
n = length ys

we can improve this with

axs ! i
ays ! j

where
axs = array (0, m) (zip [0..m] xs)
ays = array (0, n) (zip [0..n] ys)