Programming in Haskell: Lecture 15

S P Suresh

September 30, 2019

Fibonacci numbers

- Naive recursion

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Fibonacci numbers

- Naive recursion

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

- `fib n` calls `fib (n-2)` twice

Fibonacci numbers

- Naive recursion

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

- `fib n` calls `fib (n-2)` twice
  - Once directly and once from `fib (n-1)`

Fibonacci numbers

- Naive recursion

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

- `fib n` calls `fib (n-2)` twice
    - Once directly and once from `fib (n-1)`
- Similarly `fib (n-2)` calls `fib (n-4)` twice

Fibonacci numbers

- Naive recursion

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

- `fib n` calls `fib (n-2)` twice
  - Once directly and once from `fib (n-1)`
- Similarly `fib (n-2)` calls `fib (n-4)` twice
- So at least four calls to `fib (n-4)`, eight calls to `fib (n-6)`, &c.

Fibonacci numbers

- Naive recursion

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

- `fib n` calls `fib (n-2)` twice
  - Once directly and once from `fib (n-1)`
- Similarly `fib (n-2)` calls `fib (n-4)` twice
- So at least four calls to `fib (n-4)`, eight calls to `fib (n-6)`, &c.
- At least 2^k calls to `fib (n-2*k)`

Fibonacci numbers

- Naive recursion

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

- `fib n` calls `fib (n-2)` twice
  - Once directly and once from `fib (n-1)`
- Similarly `fib (n-2)` calls `fib (n-4)` twice
- So at least four calls to `fib (n-4)`, eight calls to `fib (n-6)`, &c.
- At least 2^k calls to `fib (n-2*k)`
- It appears that `fib n` takes time exponential in `n`

Fibonacci numbers: analysis

- $F(n)$: value of `fib n`

Fibonacci numbers: analysis

- $F(n)$: value of `fib n`
- $G(n)$: number of recursive calls to `fib 1` while computing `fib n`

Fibonacci numbers: analysis

- $F(n)$: value of `fib n`
- $G(n)$: number of recursive calls to `fib 1` while computing `fib n`
    - $G(0) = 0$ – no call to `fib 1`

Fibonacci numbers: analysis

- $F(n)$: value of `fib n`
- $G(n)$: number of recursive calls to `fib 1` while computing `fib n`
  - $G(0) = 0$ – no call to `fib 1`
  - $G(1) = 1$ – one call to `fib 1`

Fibonacci numbers: analysis

- $F(n)$: value of `fib n`
- $G(n)$: number of recursive calls to `fib 1` while computing `fib n`
  - $G(0) = 0$ – no call to `fib 1`
  - $G(1) = 1$ – one call to `fib 1`
  - $G(2) = 1$ – one call to `fib 1`

Fibonacci numbers: analysis

- $F(n)$: value of `fib n`
- $G(n)$: number of recursive calls to `fib 1` while computing `fib n`
  - $G(0) = 0$ – no call to `fib 1`
  - $G(1) = 1$ – one call to `fib 1`
  - $G(2) = 1$ – one call to `fib 1`
- **Claim**: For all $n \geq 0$, $G(n) = F(n)$

Fibonacci numbers: analysis

- $F(n)$: value of `fib n`
- $G(n)$: number of recursive calls to `fib 1` while computing `fib n`
    - $G(0) = 0$ – no call to `fib 1`
    - $G(1) = 1$ – one call to `fib 1`
    - $G(2) = 1$ – one call to `fib 1`
- **Claim**: For all $n \geq 0$, $G(n) = F(n)$
- True for $n = 0, 1$.

Fibonacci numbers: analysis

- $F(n)$: value of `fib n`
- $G(n)$: number of recursive calls to `fib 1` while computing `fib n`
  - $G(0) = 0$ – no call to `fib 1`
  - $G(1) = 1$ – one call to `fib 1`
  - $G(2) = 1$ – one call to `fib 1`
- **Claim**: For all $n \geq 0$, $G(n) = F(n)$
- True for $n = 0, 1$.
- For $n > 2$, there is one call to `fib (n-1)` and one to `fib (n-2)`.

Fibonacci numbers: analysis

- $F(n)$: value of `fib n`
- $G(n)$: number of recursive calls to `fib 1` while computing `fib n`
    - $G(0) = 0$ – no call to `fib 1`
    - $G(1) = 1$ – one call to `fib 1`
    - $G(2) = 1$ – one call to `fib 1`
- **Claim**: For all $n \geq 0$, $G(n) = F(n)$
- True for $n = 0, 1$.
- For $n > 2$, there is one call to `fib (n-1)` and one to `fib (n-2)`.
- So $G(n) = G(n-1) + G(n-2) = F(n-1) + F(n-2) = F(n)$.

Fibonacci numbers: analysis

- $F(n)$: value of `fib n`
- $G(n)$: number of recursive calls to `fib 1` while computing `fib n`
  - $G(0) = 0$ – no call to `fib 1`
  - $G(1) = 1$ – one call to `fib 1`
  - $G(2) = 1$ – one call to `fib 1`
- **Claim**: For all $n \geq 0$, $G(n) = F(n)$
- True for $n = 0, 1$.
- For $n > 2$, there is one call to `fib (n-1)` and one to `fib (n-2)`.
- So $G(n) = G(n-1) + G(n-2) = F(n-1) + F(n-2) = F(n)$.
- Effectively computing $F(n)$ by adding up so many 1s

Fibonacci numbers: analysis

- **Recall**: $F(n) = \dfrac{\varphi^n - \psi^n}{\sqrt{5}}$

Fibonacci numbers: analysis

- **Recall**: $F(n) = \dfrac{\varphi^n - \psi^n}{\sqrt{5}}$
- $\varphi = \dfrac{1+\sqrt{5}}{2} \approx 1.6180339887$

Fibonacci numbers: analysis

- **Recall**: $F(n) = \frac{\varphi^n - \psi^n}{\sqrt{5}}$
- $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.6180339887$
- $\psi = \frac{1-\sqrt{5}}{2} \approx -0.6180339887$

Fibonacci numbers: analysis

- **Recall**: $F(n) = \frac{\varphi^n - \psi^n}{\sqrt{5}}$
- $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.6180339887$
- $\psi = \frac{1-\sqrt{5}}{2} \approx -0.6180339887$
- Thus $G(n) = F(n)$ is exponential in n

Fibonacci numbers

- What is the problem?

Fibonacci numbers

- What is the problem?
- Multiple recursive calls with the same argument

Fibonacci numbers

- What is the problem?
- Multiple recursive calls with the same argument
- Wasteful recomputation!

Fibonacci numbers

- What is the problem?
- Multiple recursive calls with the same argument
- Wasteful recomputation!
- Suffices to keep track of two values:

```
fib = go (0,1)
    where
        go (a,b) 0 = a
        go (a,b) n = go (b, a+b) (n-1)
```

Fibonacci numbers

- A fancier solution:

```
fib = (!!) fibs
fibs = 0:1:zipWith (+) fibs (tail fibs)
```

Fibonacci numbers

- A fancier solution:

```
fib = (!!) fibs
fibs = 0:1:zipWith (+) fibs (tail fibs)
```

- Let `z = zipWith (+) fibs (tail fibs)`

Fibonacci numbers

- A fancier solution:

```
fib = (!!) fibs
fibs = 0:1:zipWith (+) fibs (tail fibs)
```

- Let `z = zipWith (+) fibs (tail fibs)`
- Then `fibs = 0:1:z`

Fibonacci numbers

- A fancier solution:

```
fib = (!!) fibs
fibs = 0:1:zipWith (+) fibs (tail fibs)
```

- Let `z = zipWith (+) fibs (tail fibs)`
- Then `fibs = 0:1:z`
- Substituting, we can define `z` without referring to `fibs`

Fibonacci numbers

- A fancier solution:

```
fib = (!!) fibs

fibs = 0:1:zipWith (+) fibs (tail fibs)
```

- Let `z = zipWith (+) fibs (tail fibs)`
- Then `fibs = 0:1:z`
- Substituting, we can define `z` without referring to `fibs`
- `z = zipWith (+) (0:1:z) (1:z)`

Fibonacci numbers

- A fancier solution:

```
fib = (!!) fibs
fibs = 0:1:zipWith (+) fibs (tail fibs)
```

- Let `z = zipWith (+) fibs (tail fibs)`
- Then `fibs = 0:1:z`
- Substituting, we can define `z` without referring to `fibs`
- `z = zipWith (+) (0:1:z) (1:z)`
- Thus `z = 1:zipWith (+) (1:z) z`

Fibonacci numbers

- A fancier solution:

```
fibs = 0:1:z
    where
        z = 1:zipWith (+) (1:z) z
```

Fibonacci numbers

- A fancier solution:

```
fibs = 0:1:z
    where
        z = 1:zipWith (+) (1:z) z
```

- Let `go = zipWith (+)` and remember the list is infinite (hence nonempty)

Fibonacci numbers

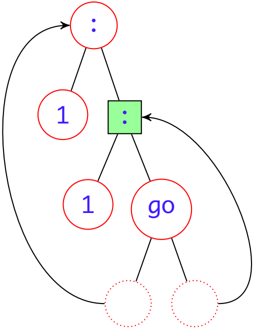
- A fancier solution:

```
fibs = 0:1:z
    where
        z = 1:zipWith (+) (1:z) z
```

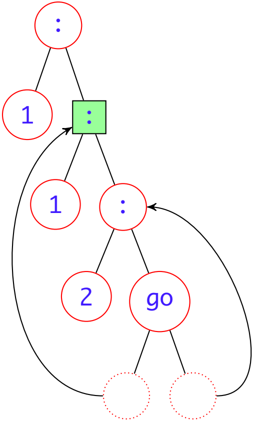
- Let `go = zipWith (+)` and remember the list is infinite (hence nonempty)
- Final code:

```
fib = (!!) fibs
fibs = 0:1:z
    where z = 1:go (1:z) z
          go (x:xs) (y:ys) = x+y: go xs ys
```

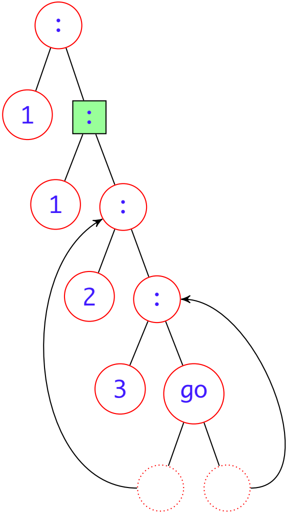
Computing `fibs`



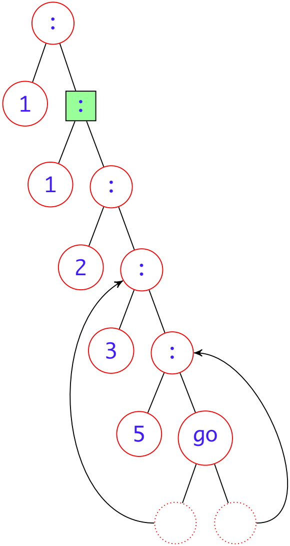
Computing `fibs`



Computing `fibs`



Computing `fibs`



Computing `fibs`

- There is always one unevaluated `go`

Computing `fibs`

- There is always one unevaluated `go`
- Pointers to two nodes on the tree

Computing `fibs`

- There is always one unevaluated `go`
- Pointers to two nodes on the tree
- The pointers move down as `go` is evaluated more and more

Computing `fibs`

- There is always one unevaluated `go`
- Pointers to two nodes on the tree
- The pointers move down as `go` is evaluated more and more
- To compute `fib n` we expand the tree to `n` levels

Dynamic programming

- **Dynamic programming** – technique to make recursive programs efficient

Dynamic programming

- **Dynamic programming** – technique to make recursive programs efficient
- Key idea is **memoization** – Keeping track of already computed values to avoid recomputation

Dynamic programming

- **Dynamic programming** – technique to make recursive programs efficient
- Key idea is **memoization** – Keeping track of already computed values to avoid recomputation
- Achieved (in the case of `fibs`) using a list defined in terms of itself

Dynamic programming

- **Dynamic programming** – technique to make recursive programs efficient
- Key idea is **memoization** – Keeping track of already computed values to avoid recomputation
- Achieved (in the case of `fibs`) using a list defined in terms of itself
- Another example next

Longest common subsequence

- Given two strings `as` and `bs`, find the **length** of the **longest common subsequence** of `as` and `bs`

Longest common subsequence

- Given two strings `as` and `bs`, find the length of the longest common subsequence of `as` and `bs`
- Haskell function `lcs`:

```
    lcs "agcat" "gact" = 3 -- subsequence "gat"
    lcs "abracadabra" "bacarrat" = 6
                                -- subsequence "bacara"
```

Longest common subsequence

- Given two strings `as` and `bs`, find the **length** of the **longest common subsequence** of `as` and `bs`
- Haskell function `lcs`:

```
    lcs "agcat" "gact" = 3 -- subsequence "gat"
    lcs "abracadabra" "bacarrat" = 6
                                    -- subsequence "bacara"
```

- **Strategy**

Longest common subsequence

- Given two strings `as` and `bs`, find the **length** of the **longest common subsequence** of `as` and `bs`
- Haskell function `lcs`:

```
    lcs "agcat" "gact" = 3 -- subsequence "gat"
    lcs "abracadabra" "bacarrat" = 6
                               -- subsequence "bacara"
```

- **Strategy**
  - If first letter is same in both strings, that letter is always in the longest common subsequence

Longest common subsequence

- Given two strings `as` and `bs`, find the length of the longest common subsequence of `as` and `bs`
- Haskell function `lcs`:

```
lcs "agcat" "gact" = 3 -- subsequence "gat"
lcs "abracadabra" "bacarrat" = 6
                        -- subsequence "bacara"
```

- Strategy
  - If first letter is same in both strings, that letter is always in the longest common subsequence
  - Else we need to skip the first letter in `as` or `bs` or both

Longest common subsequence

- Given two strings `as` and `bs`, find the length of the longest common subsequence of `as` and `bs`
- Haskell function `lcs`:

```
    lcs "agcat" "gact" = 3 -- subsequence "gat"
    lcs "abracadabra" "bacarrat" = 6
                                 -- subsequence "bacara"
```

- Strategy
    - If first letter is same in both strings, that letter is always in the longest common subsequence
    - Else we need to skip the first letter in `as` or `bs` or both
    - … and compute recursively

Longest common subsequence

- Haskell function `lcs`:

```
lcs "" _  = 0
lcs _  "" = 0
lcs as bs = if a == b then 1 + lcs as' bs'
            else max (lcs as' bs) (lcs as bs')
    where (a, as') = (head as, tail as)
          (b, bs') = (head bs, tail bs)
```

Longest common subsequence

- Haskell function `lcs`:

```
    lcs "" _  = 0
    lcs _  "" = 0
    lcs as bs = if a == b then 1 + lcs as' bs'
                else max (lcs as' bs) (lcs as bs')
        where (a, as') = (head as, tail as)
              (b, bs') = (head bs, tail bs)
```

- This takes time exponential in n

Longest common subsequence

- Haskell function `lcs`:

```
lcs "" _  = 0
lcs _  "" = 0
lcs as bs = if a == b then 1 + lcs as' bs'
            else max (lcs as' bs) (lcs as bs')
    where (a, as') = (head as, tail as)
          (b, bs') = (head bs, tail bs)
```

- This takes time exponential in n
- Same problem as with `fibs`

Longest common subsequence

- Haskell function `lcs`:

```
lcs "" _  = 0
lcs _  "" = 0
lcs as bs = if a == b then 1 + lcs as' bs'
            else max (lcs as' bs) (lcs as bs')
    where (a, as') = (head as, tail as)
          (b, bs') = (head bs, tail bs)
```

- This takes time exponential in n
- Same problem as with `fibs`
- Many recursive calls repeated with same arguments

Towards a smarter `lcs`

- Rather than present the program and explain, we shall derive it in a series of small steps

Towards a smarter `lcs`

- Rather than present the program and explain, we shall derive it in a series of small steps
- Important exercise in reasoning about programs

Towards a smarter `lcs`

- Rather than present the program and explain, we shall derive it in a series of small steps
- Important exercise in reasoning about programs
- First step: express the recursion in terms of prefixes

```
lcs "" _  = 0
lcs _  "" = 0
lcs as bs = if a == b then 1 + lcs as' bs'
                else max (lcs as' bs) (lcs as bs')
    where (as', a) = (init as, last as)
          (bs', b) = (init bs, last bs)
```

Towards a smarter `lcs`

- Let `length as = m` and `length bs = n`

Towards a smarter `lcs`

- Let `length as = m` and `length bs = n`
- For `i <- [0..m]` and `j <- [0..n]`, let

```
    f i j = lcs (take i as) (take j bs)
```

Towards a smarter `lcs`

- Let `length as = m` and `length bs = n`
- For `i <- [0..m]` and `j <- [0..n]`, let

```
    f i j = lcs (take i as) (take j bs)
```

- Then we can define `f` directly as follows:

```
    f 0 _ = 0
    f _ 0 = 0
    f i j = g (as!!(i-1)) (bs!!(j-1))
              (f (i-1) (j-1), f (i-1) j, f i (j-1))
        where g a b (d,u,l) =
              if a == b then 1+d else max l u
```

Towards a smarter `lcs`

- For `i <- [0..m]`, let

```
    l i = [f i j | j <- [0..n]]
```

Towards a smarter `lcs`

- For `i <- [0..m]`, let

```
    l i = [f i j | j <- [0..n]]
```

- `l 0 = replicate (n+1) 0`

Towards a smarter `lcs`

- For `i <- [0..m]`, let

```
l i = [f i j | j <- [0..n]]
```

- `l 0 = replicate (n+1) 0`
- For `i > 0`,

```
l i = 0: [g (as!!(i-1)) (bs!!(j-1))
            (f (i-1) (j-1), f (i-1) j, f i (j-1))
               | j <- [1..n]]
```

Towards a smarter `lcs`

- We can define `l i` directly in terms of itself and `l (i-1)`

Towards a smarter `lcs`

- We can define `l i` directly in terms of itself and `l (i-1)`
- Observe that:

```
    zip3 (l (i-1)) (tail (l (i-1))) (l i) =
        [(f (i-1) (j-1), f (i-1) j, f i (j-1)) | j <- [1..n]]
```

Towards a smarter `lcs`

- We can define `l i` directly in terms of itself and `l (i-1)`
- Observe that:

```
zip3 (l (i-1)) (tail (l (i-1))) (l i) =
    [(f (i-1) (j-1), f (i-1) j, f i (j-1)) | j <- [1..n]]
```

- So

```
l i = 0 : zipWith (g (as!!(i-1))) bs
             (zip3 (l (i-1)) (tail (l (i-1))) (l i))
```

Towards a smarter `lcs`

- We have:

```
l i = 0 : zipWith (g (as!!(i-1))) bs
          (zip3 (l (i-1)) (tail (l (i-1))) (l i))
```

Towards a smarter `lcs`

- We have:

```
l i = 0 : zipWith (g (as!!(i-1))) bs
          (zip3 (l (i-1)) (tail (l (i-1))) (l i))
```

- Can clean it further:

```
l i = nextList (as!!(i-1)) (l (i-1))
    where nextList a l = 0 : zipWith (g a) bs
          (zip3 l (tail l) (nextList a l))
```

Towards a smarter `lcs`

- We have:

```
    l i = nextList (as!!(i-1)) (l (i-1))
        where nextList a l = 0 : zipWith (g a) bs
              (zip3 l (tail l) (nextList a l))
```

Towards a smarter `lcs`

- We have:

```
l i = nextList (as!!(i-1)) (l (i-1))
    where nextList a l = 0 : zipWith (g a) bs
              (zip3 l (tail l) (nextList a l))
```

- Let `lcsTab = [l i | i <- [1..m]]`

Towards a smarter `lcs`

- We have:

```
l i = nextList (as!!(i-1)) (l (i-1))
    where nextList a l = 0 : zipWith (g a) bs
            (zip3 l (tail l) (nextList a l))
```

- Let `lcsTab = [l i | i <- [1..m]]`
- Then `l i = lcsTab!!i`

Towards a smarter `lcs`

- We have:

```
l i = nextList (as!!(i-1)) (l (i-1))
    where nextList a l = 0 : zipWith (g a) bs
           (zip3 l (tail l) (nextList a l))
```

- Let `lcsTab = [l i | i <- [1..m]]`
- Then `l i = lcsTab!!i`
- So we have

```
lcsTab = l 0 : [nextList (as!!(i-1)) (lcsTab!!(i-1))
                           | i <- [1..m]]
```

Smarter `lcs`: we are there!

- We have:

```
lcsTab = l 0 : [nextList (as!!(i-1)) (lcsTab!!(i-1))
                                 | i <- [1..m]]
```

Smarter `lcs`: we are there!

- We have:

```
lcsTab = l 0 : [nextList (as!!(i-1)) (lcsTab!!(i-1))
                        | i <- [1..m]]
```

- Final simplification:

```
lcsTab = l 0 : zipWith nextList as lcsTab
```

Smarter `lcs`: we are there!

- We have:

```
lcsTab = l 0 : [nextList (as!!(i-1)) (lcsTab!!(i-1))
                          | i <- [1..m]]
```

- Final simplification:

```
lcsTab = l 0 : zipWith nextList as lcsTab
```

- Recall that `l 0` is just a list of `0`s

Smarter `lcs`: we are there!

- We have:

```
lcsTab = l 0 : [nextList (as!!(i-1)) (lcsTab!!(i-1))
                              | i <- [1..m]]
```

- Final simplification:

```
lcsTab = l 0 : zipWith nextList as lcsTab
```

- Recall that `l 0` is just a list of `0`s
- The final answer we want is `f m n = last (last lcsTab)`

Putting it all together

```haskell
lcs :: String -> String -> Int
lcs as bs = last (last lcsTab)
    where
    lcsTab        = firstList : zipWith nextList as lcsTab
    firstList     = replicate (length bs + 1) 0
    nextList a l  = 0: zipWith (g a) bs
                            (zip3 l (tail l) (nextList a l))
    g a b (d,u,l) = if a == b then 1 + d else (max u l)
```

Complexity of `lcs`

- Laziness ensures that `lcsTab` is expanded as needed

Complexity of `lcs`

- Laziness ensures that `lcsTab` is expanded as needed
- An analysis similar to `fib` can be performed

Complexity of `lcs`

- Laziness ensures that `lcsTab` is expanded as needed
- An analysis similar to `fib` can be performed
- `lcsTab` is computed completely in $O(m \cdot n)$ time

Complexity of `lcs`

- Laziness ensures that `lcsTab` is expanded as needed
- An analysis similar to `fib` can be performed
- `lcsTab` is computed completely in $O(m \cdot n)$ time
- Sample runs on the strings `"ababababababababababababababababababab"` and `"bbbbbbbbbbbbbbbbbbbbbbbbb"`

Complexity of `lcs`

- Laziness ensures that `lcsTab` is expanded as needed
- An analysis similar to `fib` can be performed
- `lcsTab` is computed completely in $O(m \cdot n)$ time
- Sample runs on the strings `"ababababababababababababababababababab"` and `"bbbbbbbbbbbbbbbbbbbbbbbbb"`
    - Answer is `19`

Complexity of `lcs`

- Laziness ensures that `lcsTab` is expanded as needed
- An analysis similar to `fib` can be performed
- `lcsTab` is computed completely in $O(m \cdot n)$ time
- Sample runs on the strings `"abababababababababababababababababababab"` and `"bbbbbbbbbbbbbbbbbbbbbbbbb"`
  - Answer is `19`
  - Naive recursion: `(32.35 secs, 19,422,476,336 bytes)`

Complexity of `lcs`

- Laziness ensures that `lcsTab` is expanded as needed
- An analysis similar to `fib` can be performed
- `lcsTab` is computed completely in $O(m \cdot n)$ time
- Sample runs on the strings `"abababababababababababababababababababab"` and `"bbbbbbbbbbbbbbbbbbbbbbbbb"`
    - Answer is `19`
    - Naive recursion: `(32.35 secs, 19,422,476,336 bytes)`
    - DP version: `(0.01 secs, 3,504,504 bytes)`

Computing the subsequence itself

```
lcs :: String -> String -> (Int, String)
lcs as bs = last (last lcsTab)
    where
    lcsTab          = firstList : zipWith nextList as lcsTab
    firstList       = replicate (length bs + 1) (0, "")
    nextList a l    = (0, ""): zipWith (g a) bs
                           (zip3 l (tail l) (nextList a l))
    g a b (d,u,l)   = if a == b
                      then (1 + fst d, snd d ++ [b])
                      else
                          if fst u > fst l then u else l
```