Today in CS5:

chr(9829)

The ♥ of CS (and CSers...)

Algorithms!

https://xkcd.com/2891/

I *feel at home* with recursion now!

Caesar Cipher: `encipher`

`encipher(s,n)` should return the string `s` with each ***alphabetic*** character shifted/wrapped by `n` places in the alphabet

```
encipher( 'I <3 Latin' , 0 )   returns   'I <3 Latin'
encipher( 'I <3 Latin' , 1 )   returns   'J <3 Mbujo'
encipher( 'I <3 Latin' , 2 )   returns   'K <3 Ncvkp'
encipher( 'I <3 Latin' , 3 )   returns   'L <3 Odwlq'
encipher( 'I <3 Latin' , 4 )   returns   'M <3 Pexmr'
encipher( 'I <3 Latin' , 5 )   returns   'N <3 Qfyns'
                               :
encipher( 'I <3 Latin' , 25 )  returns   'H <3 Kzshm'
```

Algorithm 0

Caesar Cipher: `encipher`

`encipher(s,n)` should return the string `s` with each ***alphabetic*** character shifted/wrapped by `n` places in the alphabet

```
encipher( 'I <3 Latin' , 0 )  returns  'I <3 Latin'
encipher( 'I <3 Latin' , 1 )  returns  'J <3 Mbujo'
encipher( 'I <3 Latin' , 2 )  returns  'K <3 Ncvkp'
```

"…si qua occultius perferenda erant, per notas scripsit, id est sic structo litterarum ordine, ut nullum verbum effici posset; quae si qui investigare et persequi velit, quartam elementorum litteram, **id est D pro A** et perinde reliquas commutet…"

- Suetonius, *De Vitae Caesar*

"…if any were to be conveyed more secretly, he wrote in notes, that is, in such a structured order of letters that no word could be made; that is, he **exchanges D for A** and exchanges the rest in the same manner…"

- Suetonius, *The Life of Caesar*

Design…

*design of **what**?*

The ♥ of CS

(and CSers…)

Algorithms!

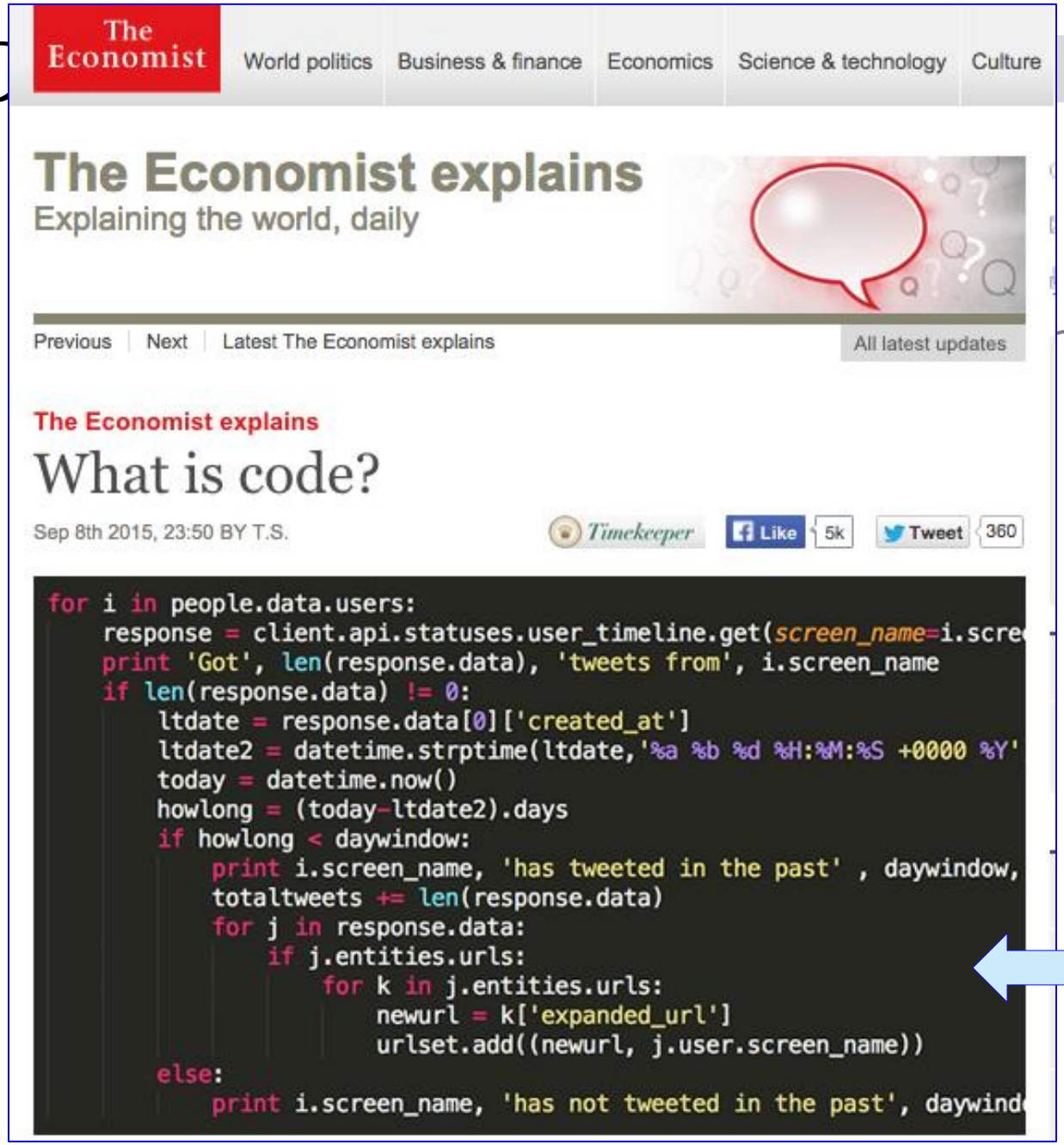
Design…

*design of **what**?*

~~*Code?*~~

syntax

*design of **what**?*



The Economist | World politics | Business & finance | Economics | Science & technology | Culture

The Economist explains

Explaining the world, daily

Previous | Next | Latest The Economist explains | All latest updates

The Economist explains

What is code?

Sep 8th 2015, 23:50 BY T.S.

Timekeeper | Like 5k | Tweet 360

```
for i in people.data.users:
    response = client.api.statuses.user_timeline.get(screen_name=i.scre
    print 'Got', len(response.data), 'tweets from', i.screen_name
    if len(response.data) != 0:
        ltdate = response.data[0]['created_at']
        ltdate2 = datetime.strptime(ltdate,'%a %b %d %H:%M:%S +0000 %Y'
        today = datetime.now()
        howlong = (today-ltdate2).days
        if howlong < daywindow:
            print i.screen_name, 'has tweeted in the past' , daywindow,
            totaltweets += len(response.data)
            for j in response.data:
                if j.entities.urls:
                    for k in j.entities.urls:
                        newurl = k['expanded_url']
                        urlset.add((newurl, j.user.screen_name))
        else:
            print i.screen_name, 'has not tweeted in the past', daywind
```

Python!

*design of **what**?*

The Economist | World politics | Business & finance | Economics | Science & technology | Culture

The Economist explains

Explaining the world, daily

Previous | Next | Latest The Economist explains

The Economist explains

What is code?

Sep 8th 2015, 23:50 BY T.S.

FROM lifts to cars to airliners to smartphones, modern civilisation is powered by software, the digital instructions that allow computers, and the devices they control, to perform calculations and respond to their surroundings. How did that software get there? Someone had to write it. But code, the sequences of symbols painstakingly created by programmers, is not quite the same as software, the sequences of instructions that computers execute. So what exactly is it?

syntax

Coding, or programming, is a way of writing instructions for computers that bridges the gap between how humans like to express themselves and how computers actually work. Programming languages, of which there are hundreds, cannot generally be executed by computers directly. Instead, programs written in a particular "high level" language such as C++, Python or Java are translated by a special piece of software (a compiler or an interpreter) into low-level instructions which a computer can actually run. In some cases programmers write software in low-level instructions directly, but this is fiddly. It is usually much easier to use a high-level programming language, because such languages make it

```
for i in people.data.users:
    response = client.api.statuse
    print 'Got', len(response.dat
    if len(response.data) != 0:
        ltdate = response.data[0]
        ltdate2 = datetime.strptime(ltdate, ...
        today = datetime.now()
        howlong = (today-ltdate2).days
        if howlong < daywindow:
            print i.screen_name, 'has tweeted in the past' , daywindow,
            totaltweets += len(response.data)
            for j in response.data:
                if j.entities.urls:
                    for k in j.entities.urls:
                        newurl = k['expanded_url']
                        urlset.add((newurl, j.user.screen_name))
        else:
            print i.screen_name, 'has not tweeted in the past', daywind
```

Python!

Design…

*design of **what**?*

~~*Code?*~~

syntax

Algorithms!

ideas!

Algorithm Design...

`remAll(e,L)`

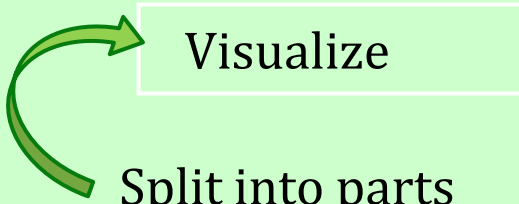
remove all e's from L

Design…

`remAll(e,L)`

remove all e's from L

Top-down design



Visualize

Split into parts

Build each part

Combine

Test

L1

```
remAll(42,[5,7,42,8,42])
```



```
[5,7,8]
```

L2

```
remAll('q','qaqqlqqiqqiiqeqqnqs')
```



```
'aliiiens'
```

Design...

`remAll(e,L)`

remove all e's from L

Top-down design

Visualize

Split into parts

Build each part

Combine

Test

'it'

L[0] and L[1:]

'the rest'

Use it!

it

```
remAll(42,[5,7,42,8,42])
```

'the rest'

```
[5,7,8]
```

it

```
remAll('q','qaqqlqqiqqiiqeqqnqs')
```

'the rest'

```
'aliiiens'
```

Lose it!

Design...

Top-down design

Visualize

Split into parts

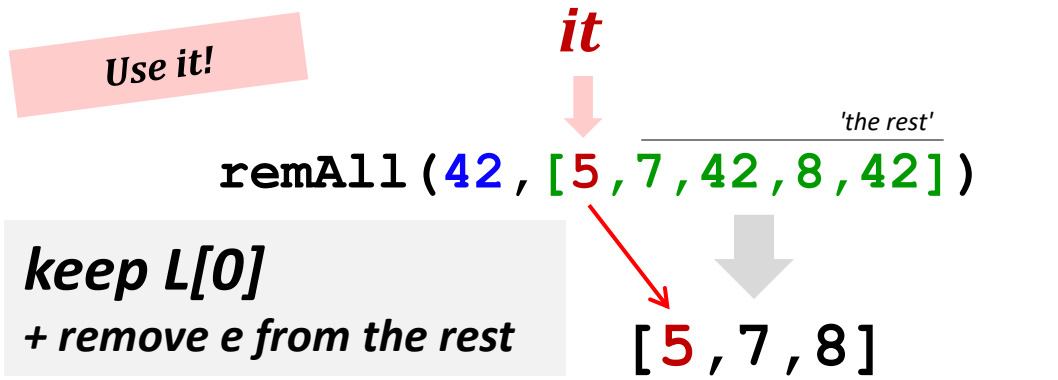
Build each part

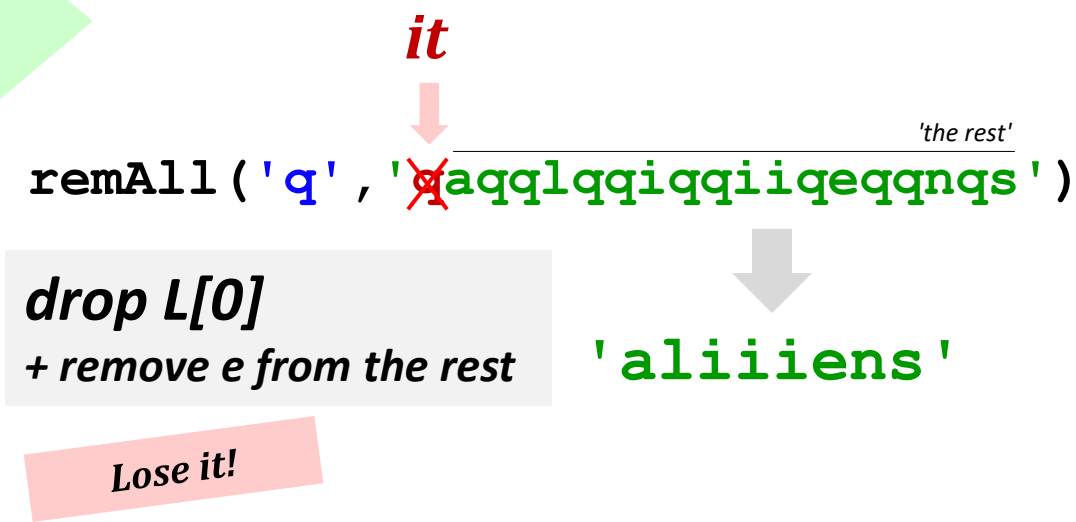
Combine

Test

`remAll(e,L)`

remove all e's from L





Design…

Top-down design

Visualize

Split into parts

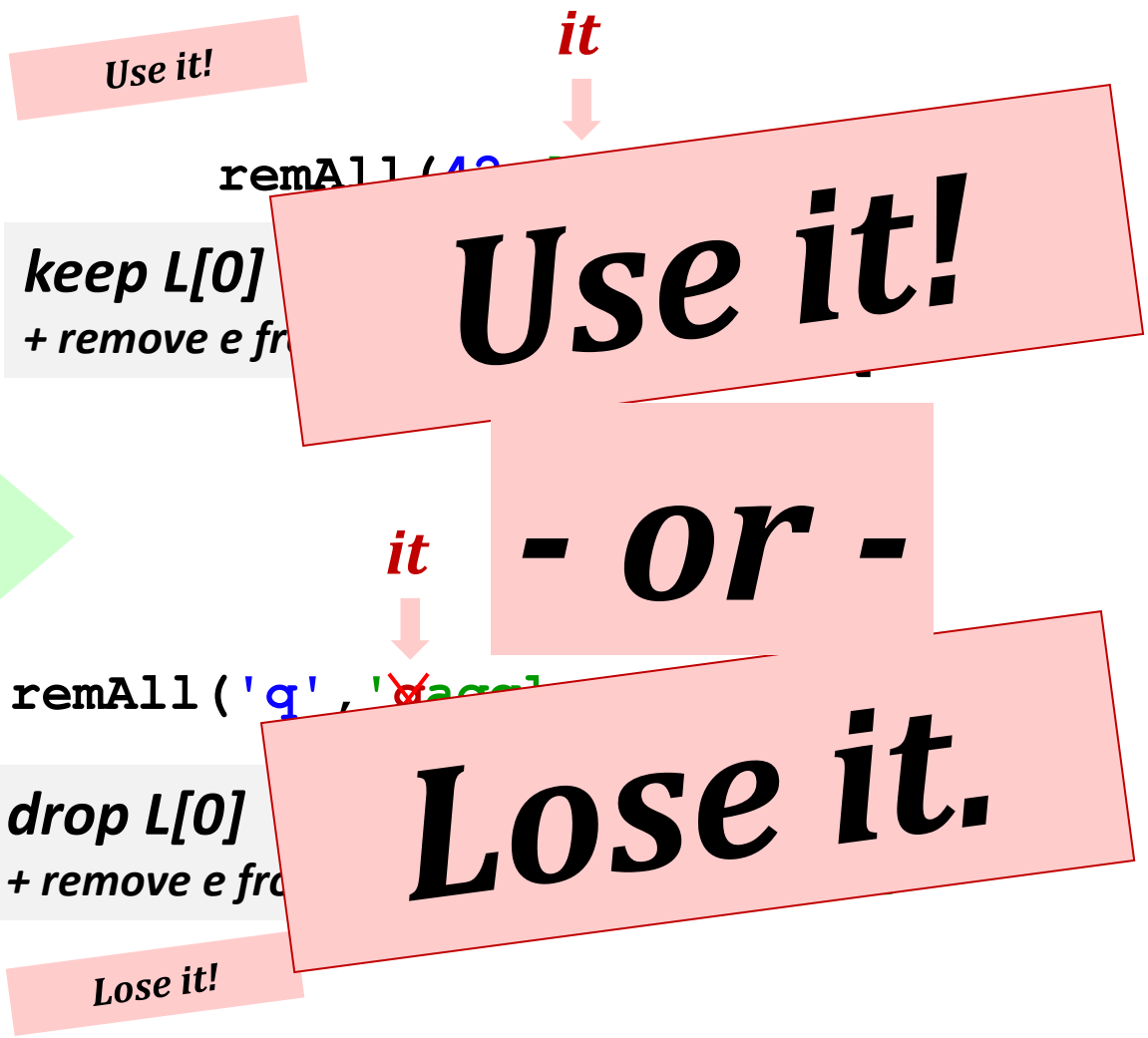
Build each part

Combine

Test

`remAll(e,L)`

remove all e's from L



Design ~ *code*

`remAll(e,L)`

remove all e's from L

Top-down design

Re-Visualize *in syntax!?*

If there are no elements or characters in L, we're done – return L itself!

```
def remAll( e, L ):
    """ removes all       from L """
    if len(L) == 0:
        return L
    elif L[0] != e:
        return L[0:1] + remAll(e,L[1:])
    else:
        return            remAll(e,L[1:])
```

Design ~ *code*

5 [7,5,42]

remAll(e,L)

remove all e's from L

Top-down design

Re-Visualize *in syntax!?*

```
def remAll( e, L ):
    """ removes all                    L """
    if len(L) == 0:
        return L
    elif L[0] != e:
        return L[0:1] + remAll(e,L[1:])
    else:
        return          remAll(e,L[1:])
```

If **it** is not e,

USE it (keep it in the return value)

AND remove all the e's from the rest of L!

Design ~ *code*

7 [7,5,42]

remAll(e,L)

remove all e's from L

Top-down design

Re-Visualize *in syntax!?*

```
def remAll( e, L ):
    """ removes all e's from L """
    if len(L) == 0:
        return L
    elif L[0] != e:
        return L[0:    + remAll(e,L
    else:
        return                    remAll(e,L[1:])
```

If it *is* e,

LOSE it (don't keep it in the return value)

AND still remove all of the e's from the rest of L!

Design ~ *code*

`remAll(e,L)`

That's it. ***Algorithmic expression ~ it's what CSers*** do.
(think we)

Re-visualize *in syntax*!?

```
def remAll( e, L ):
    """ removes all e's from L """
    if len(L) == 0:
        return L
    elif L[0] != e:
        return L[0:1] + remAll(e,L[1:])
    else:
        return remAll(e,L[1:])
```

`remAll` insight

```
def remAll( e, L ):
    """ removes all e's from L """
    if len(L) == 0:
        return L
    elif L[0] != e:
        return L[0:1] + remAll(e,L[1:])
    else:
        return remAll(e,L[1:])
```

syntax

`remAll(8, [7,8,9,8])` ➡ `[7,9]`

0 1 2 3

sharpening our model for where + how actions happen...

other **rem** examples...

```
remAll(8, [7,8,9,8]) ➡ [7,9]        remAll
remAll('d', 'coded') ➡ 'coe'        remAll

remOne(8, [7,8,9,8]) ➡ [7,9,8]      remOne
remOne('d', 'coded') ➡ 'coed'       remOne

remUpto(8,[7,8,9,8]) ➡ [9,8]        remUpto
remUpto('d','coded') ➡ 'ed'         remUpto
```

Subsequences!

in order, but not necessarily **adjacent**...

```
def subseq( s, sbig )
```
→ True or False?

s is the subsequence to find (or not)

sbig is the bigger string in which we are looking for **s**

```
subseq('', 'cataga') → True
subseq('ctg', 'cataga') → True
subseq('ctg', 'tacggta') →
subseq('aliens', 'always frighten dragons')→
subseq('trogdor', 'that dragon is gone for good')
→
```

T or **F**?

Here there be NO dragons!

Why Are these True? or False?

Try it... Algorithm design

Quiz

```
def remAll( e, L ):
    """ removes all e's from L """
    if len(L) == 0:
        return L
    elif L[0] != e:
        return L[0:1]
    else:
        return    remAll(e,L[1:])
```

Hint: remove one thing for **remOne**!

Hint: remove one ***more*** thing for **remUpto**!

1

Change **remAll** so that it removes only ***one*** **e** from **L**. (We could call it **remOne**.)

remOne(8,[7,8,9,8]) ➡ **[7,9,8]**

what's needed

is mostly crossing stuff out!

What stuff?

2

Make *more* changes to **remAll** so that it removes all of the elements ***up to and including the first*** **e** in **L**. (We could call it **remUpto**.)

remUpto('d','coded') ➡ **'ed'**

If e is not in L, remUpto should remove *everything*...

```
def subseq( s, sbig ):
    """ returns True if s is a subseq. of sbig,
        False otherwise. Both are strings.
    """
    if s == '':
        return True
    elif
```

Challenge...

3 Write the other cases needed for **subseq**...

Hint: you'll need 3-4 cases total for **subseq**.

```
subseq('alg','magical')
lse

subseq('alg','twasbrillig')
True
```

Try it... Algorithm design

Names: _____

```
def remAll( e, L ):
    """ removes all e's from L """
    if len(L) == 0:
        return L
    elif L[0] != e:
        return L[0:1] + remAll(e,L[1:])
    else:
        return   remAll(e,L[1:])
```

1

Change **remAll** so that it removes only ***one*** **e** from **L**. (We could call it **remOne**.)

remOne(8,[7,8,9,8]) ➡ **[7,9,8]**

Hint: In both 1 + 2, what's needed is *mostly crossing stuff out! What stuff?*

2

Make *more* changes to **remAll** so that it removes all of the elements ***up to and including the first*** **e** in **L**. (We could call it **remUpto**.)

remUpto('d','coded') ➡ **'ed'**

If e is not in L, remUpto should remove *everything*...

```
def subseq( s, sbig ):
    """ returns True if s is a subseq. of sbig,
        False otherwise. Both are strings.
    """
    if s == '':
        return True
    elif
```

Challenge...

3 Write the other cases needed for **subseq**...

```
subseq('alg','magical')
False

subseq('alg','twasbrillig')
True
```

from **remAll** to **remOne**

Hint: remove one thing for **remOne**!

```
def remAll( e, L ):
    """ returns seq. L with all e's rmovd
    """
    if  len(L) == 0:
        return L

    elif  L[0] != e:
        return  L[0:1] + remAll( e, L[1:] )

    else:
        return  remAll( e, L[1:] )
```

Is remAll really "an algorithm"?

`remOne(8,[7,8,9,8])` ➡ `[7,9,8]`

`remOne('d','coded')` ➡ `'coed'`

from **remAll** to **remOne**

Hint: remove one thing for **remOne**!

```
def remOneAll( e, L ):
    """ returns seq. L with all e's rmovd
    """
    if  len(L) == 0:
        return L

    elif  L[0] != e:
        return  L[0:1] + remOne( e, L[1:] )

    else:
        return  remAll( e, L[1:] )
```

Wait! I see **One** more error!

`remOne(8,[7,8,9,8])` ➡ `[7,9,8]`

`remOne('d','coded')` ➡ `'coed'`

from **remOne** to **remUpto**

Hint: remove one ***more*** thing for **remUpto**!

```
        Upto
def remOne( e, L ):
    """ returns seq. L with one e rmoved
    """
    if  len(L) == 0:
        return L

    elif  L[0] != e:
        return  L[0:1] + remUpto( e, L[1:] )

    else:
        return  L[1:]
```

I <3 remSleep!

`remUpto(8,[7,8,9,8])` ➡ `[9,8]`

`remUpto('d','coded')` ➡ `'ed'`

from **remOne** to **remUpto**

Hint: remove one ***more*** thing for **remUpto**!

```
        Upto
def remOne( e, L ):
    """ returns seq. L with one e rmoved
    """
    if  len(L) == 0:
        return L

    elif  L[0] != e:
        return  L[0:1] + remUpto( e, L[1:] )

    else:
        return  L[1:]
```

I <3 remSleep!

`remUpto(8,[7,8,9,8])` ➡ `[9,8]`

`remUpto('d','coded')` ➡ `'ed'`

Subseq ~ *coding* it out...

```
def subseq( s, sbig ):
    """ returns True if s is a subseq. of sbig;
        False otherwise. Both are strings.
    """
    if s == '':
        return True

    elif s[0]
```

it

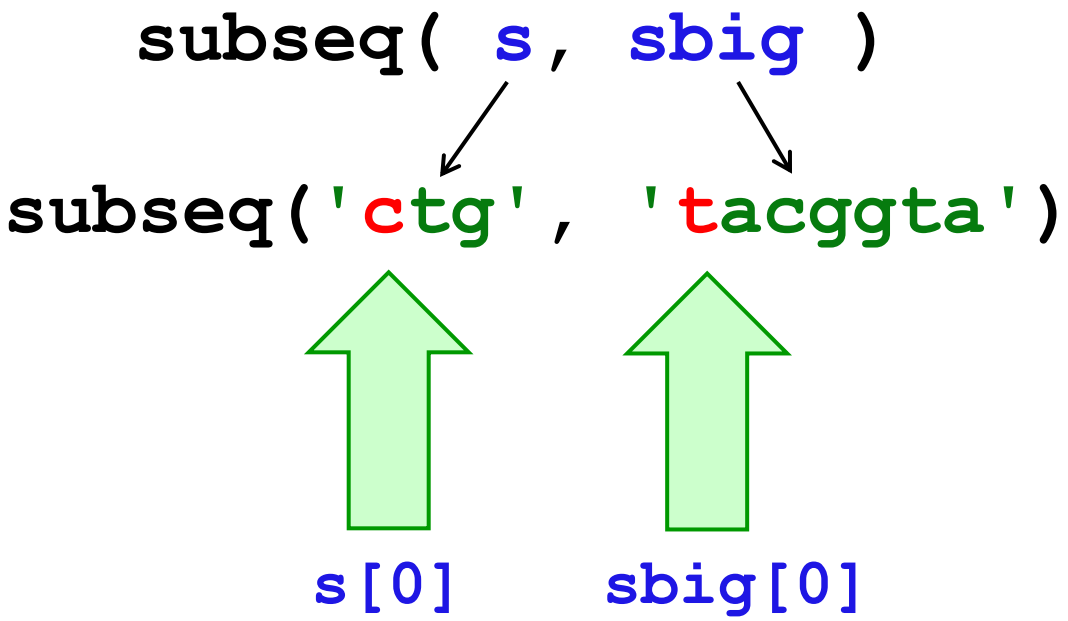
Base case(s)

but first, *algorithms*!

Recursive step(s)

Where are the *useit* and *loseit* here?

Subseq ~ *thinking* it out...



What is a small (initial) piece of the problem?

How would we describe it in terms of the inputs?

Use it!

- or -

Lose it!

What is left after handling this piece?

Are there other functions we will need?

Top-down design

Visualize

Split into parts

Build each part

Combine

Test

Subseq ~ *coding* it out...

```
def subseq( s, sbig ):
    """ returns True if s is a subseq. of sbig;
        False otherwise. Both are strings.
    """
    if s == '':
        return True

    elif s[0]
```

it

Base case(s)

Recursive step(s)

Where are the *useit* and *loseit* here?

Subseq ~ *coding* it out...

```
def subseq( s, sbig ):
    """ returns True if s is a subseq. of sbig;
        False otherwise. Both are strings.
    """
    if s == '':
        return True

    elif s[0] not in sbig:
        return False
    elif s[0] == sbig[0]:
        return subseq( s[1:], sbig[1:] )
    else:
        return subseq( s[0:], sbig[1:] )
```

it

Base case(s)

rest of `s`

rest of `sbig`

Recursive step(s)

Where are the *useit* and *loseit* here?

Subseq ~ *coding* it out...

```
def subseq( s, sbig ):
    """ returns True if s is a subseq. of sbig;
        False otherwise. Both are strings.
    """
    if s == '':
        return True

    elif s[0] not in sbig:
        return False

    else:
        return subseq(s[1:],remUpto(s[0],sbig))
```

it

Base case(s)

rest of `s`

rest of `sbig` after `s[0]`

Recursive step(s)

Where are the *useit* and *loseit* here?

Subseq ~ *coding* it out...

```
def subseq( s, sbig ):
    """ returns True if s is a subseq. of sbig;
        False otherwise. Both are strings.
    """
    if s == '':
        return True

    elif s[0] not in sbig:
        return False

    else:
        return subseq(s[1:],remUpto(s[0],sbig))
```

it

Base case(s)

"Use it or lose it"

Recursive step(s)

Where are the *useit* and *loseit* here?

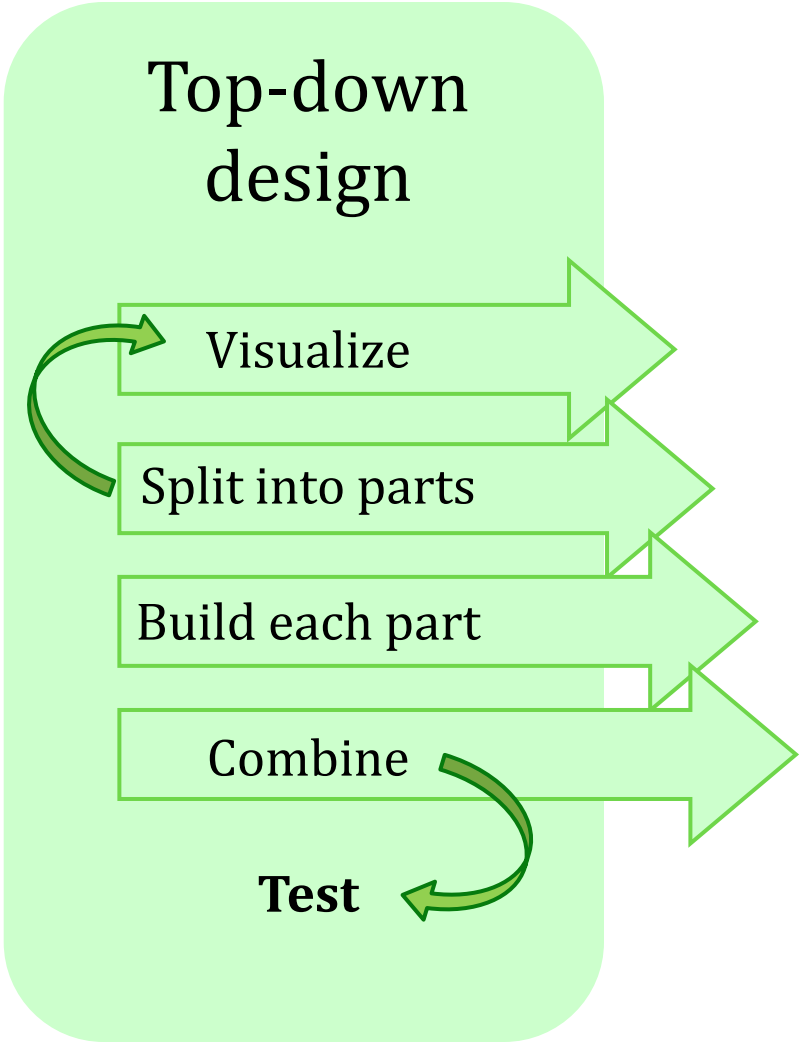
Design ~ *(code)*

That's it. ***Algorithmic expression ~ it's what CSers think they do.***

... at this moment in a prior CS5 ...

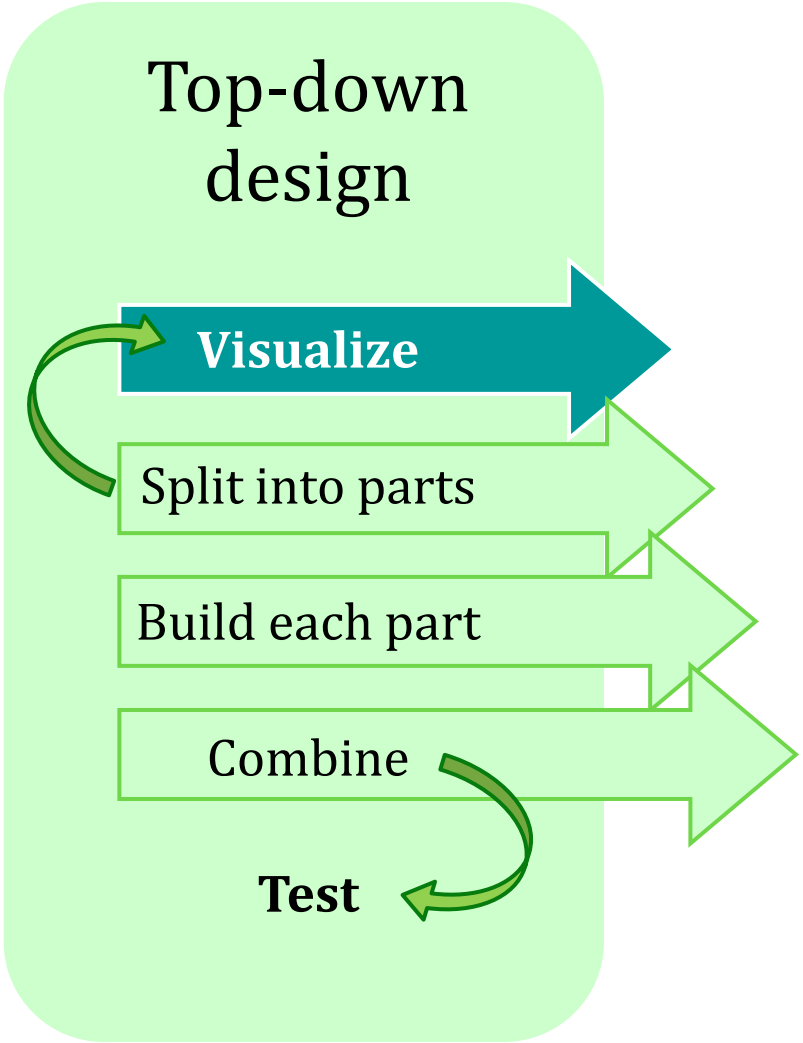
it can take some *"getting used to"* ... ?

*What's the **problem**?!*



*Which **one** of these steps is the most important?*

What's the problem?!



understanding *what the problem demands!!*

*I want some **examples**!*

hw3pr2: *use-it-or-lose-it algorithm design*

Longest **C**ommon **S**ubsequence — LCS(S, T)

Jotto **Score** counting — jscore(s1, s2)

binary **l**ist and
general list **sort**ing — blsort(L), gensort(L)

exact_change making — exact_change(t, L)

hw3pr2: *use it or lose it*

Longest **C**ommon **S**ubsequence

LCS(S, T)

'BONOBO'

'CHIMPANZEE'

'CGCTGAGCTAGGCA...'

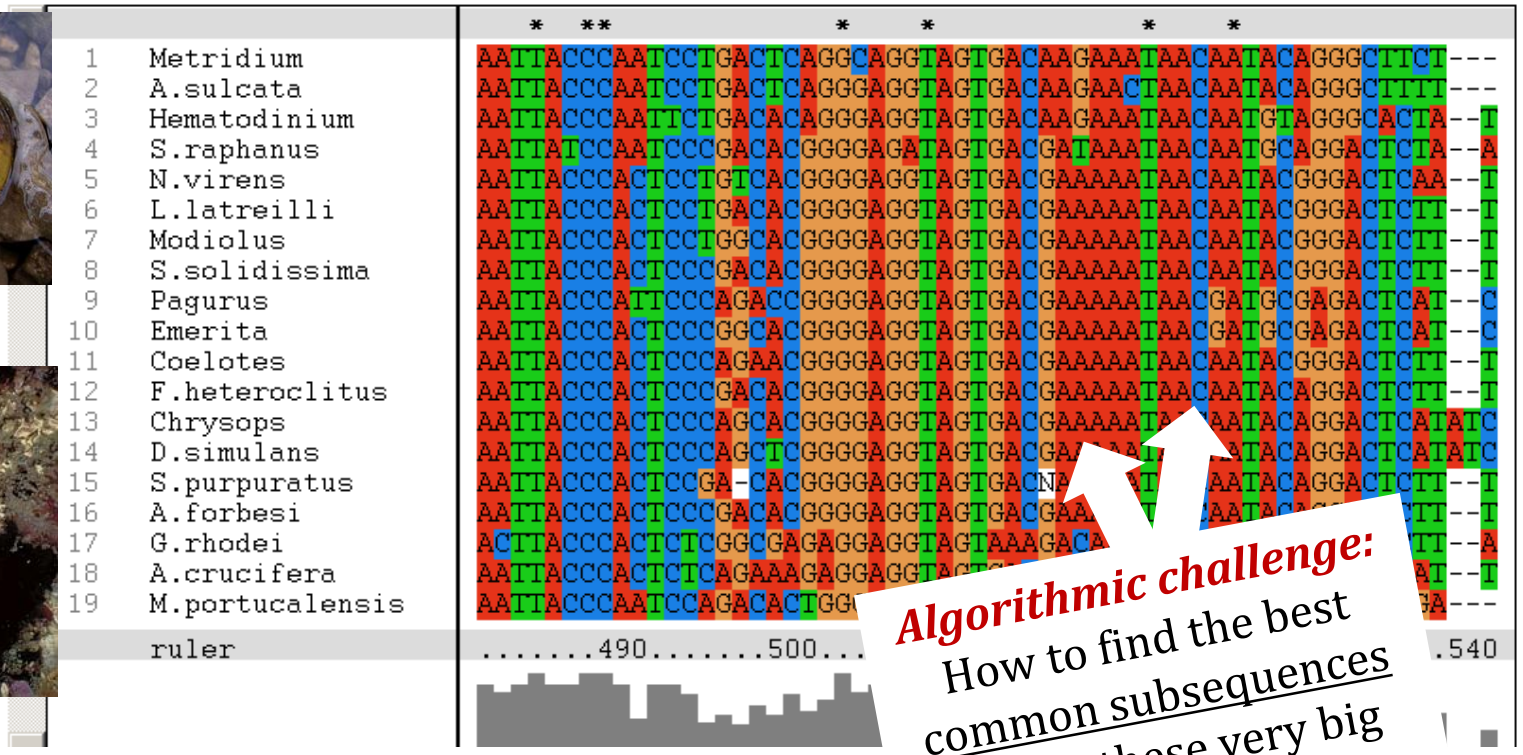
'ATCCTAGGTAACTG...'

+10^9more

Eye oneder if this haz other aplications?

Why LCS?

Screenshot from the ClustalX multiple subsequence alignment tool...

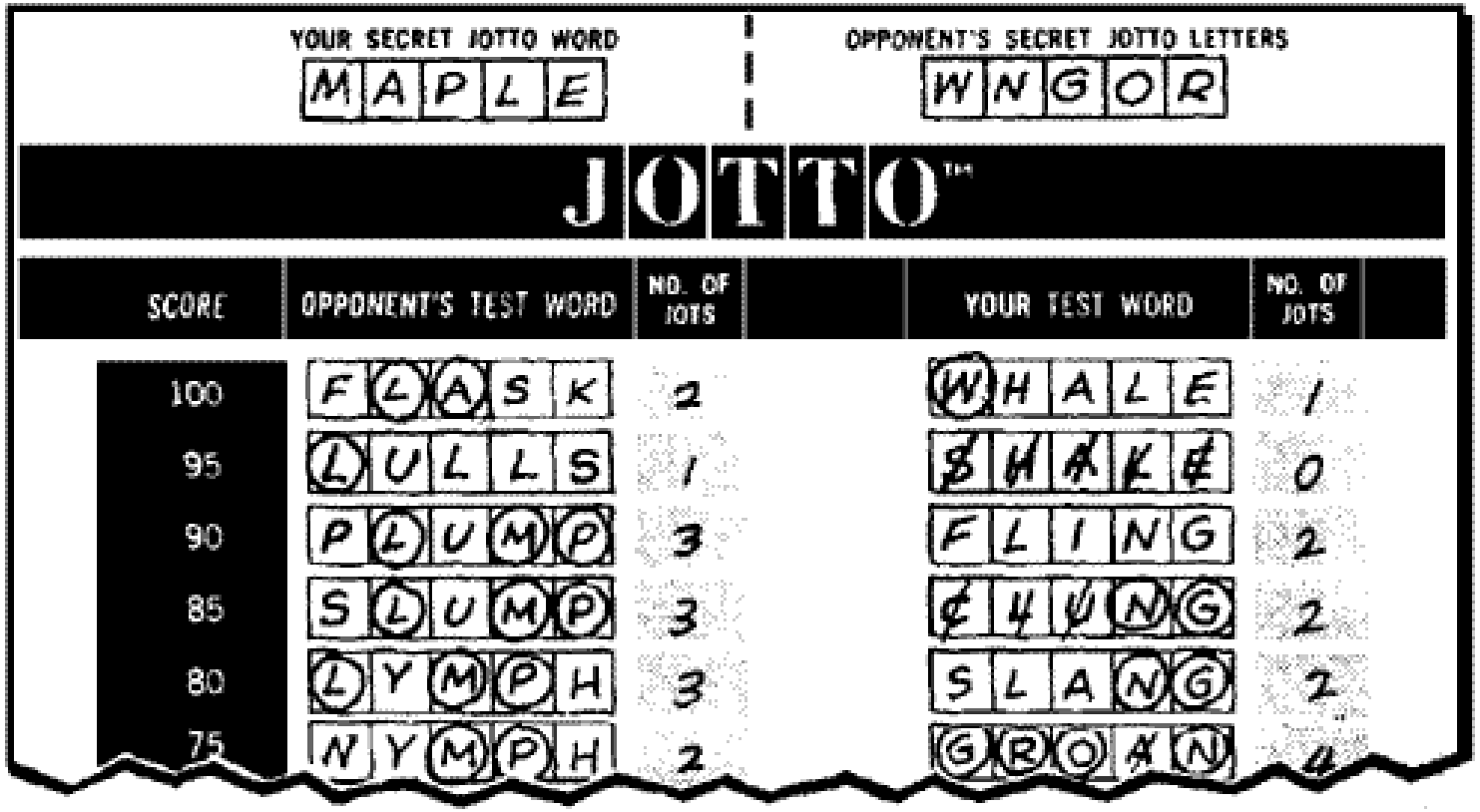


Algorithmic challenge:
How to find the best common subsequences among these very big genome strings ?!?

also in hw3pr2: *Jotto* !

a word-guessing game…

jscore(S, T)



jscore

"Jotto scoring"

These are two cute

'robot'

'otter'

```
jscore( 'robot', 'otter' ) ⟶
```

```
jscore( S, T )
```

Let's try it!

also in hw3pr2: **`sort` + `exact_change`**

sort([42,5,7] **)** ⟶ [5,7,42]

sort([42,7] **)** ⟶ [7,42]

sort([42] **)** ⟶ [42]

returns an ascending list

---

exact_change(42, [25,30,2,5] **)** ⟶ `False`

exact_change(42, [25,30,2,15] **)** ⟶ `True`

returns **`True`** or **`False`**

should return the jotto score for any strings **S** and **T**

jscore(S,T)

jscore('robot', 'otter') → **3**

jscore('geese', 'seems') → **3**

jscore('fluff', 'lulls') → **2**

jscore('pears', 'diner') →

jscore('xylyl', 'slyly') →

Extra! Which of these 10 is the *cruellest* hidden jotto word?

Use it!

Lose it!

remOne

how is remOne used?

should return a new list that is the sorted version of the input **L**

sort(L)

sort([42,5,7]) → **[5,7,42]**

sort([42,7]) → **[7,42]**

sort([42]) → **[42]**

sort([ ]) →

blsort([1,0,1]) →

binary-list sort: same as sort, but all of the #s are 0 or 1

don't write any code for these...

min

remOne

how are min and remOne used?

should return the Longest Common Subsequence of strings **S** and **T**

LCS(S,T)

LCS('ctga', 'tagca') → 'tga'

LCS('tga', 'taacg') → 'ta' (or 'tg')

LCS('tga', 'a') →

LCS('gattaca', 'ctctgcgat') →

do answer examples + *brainstorm*

Use it!

Lose it!

Lose it!

only recursion here...

this is eerily like svTree

Brainstorm algorithms for these problems. What **helper functions???** might help for each...

returns True if ***any*** subset of elements in L add up to t; returns False otherwise

exact_change(t,L)

exact_change(42, [25,30,2,5]) → **False**

exact_change(42, [22,16,3,2,17]) →

exact_change(42, [18,21,22]) →

exact_change(42, [40,17,1,7]) →

exact_change(20, [16,3,2,17]) →

Use it!

Lose it!

... and here

should return the jotto score for any strings **s1** and **s2**

jscore(s1,s2)

jscore('robot', 'otter') → 3
jscore('geese', 'seems') → 3
jscore('fluff', 'lulls') → 2
jscore('pears', 'diner') → 2
jscore('xylyl', 'slyly') → 4

Extra! Which of these 10 is the *cruellest* hidden jotto word?

Use it!
Lose it!

remOne

how is remOne used?

should return a new list that is the sorted version of the input **L**

sort(L)

sort([42,5,7]) → [5,7,42]
sort([42,7]) → [7,42]
sort([42]) → [42]
sort([ ]) → [ ]
blsort([1,0,1]) → [0,1,1]

binary-list sort: same as sort, but all of the #s are 0 or 1

don't write any code for these...

min
remOne

how are min and remOne used?

should return the Longest Common Subsequence of strings **S** and **T**

LCS(S,T)

LCS('ctga', 'tagca') → 'tga'
LCS('tga', 'taacg') → 'ta' (or 'tg')
LCS('tga', 'a') → 'a'
LCS('gattaca', 'ctctgcgat') → 'ttca'

4 chars

do answer examples + *brainstorm*

Use it!
Lose it!
Lose it!

only recursion here...

this is eerily like svTree

Brainstorm algorithms for these problems. What **helper functions???** might help for each...

returns True if ***any*** subset of elements in L add up to t; returns False otherwise

exact_change(t,L)

exact_change(42, [25,30,2,5]) → False
exact_change(42, [22,16,3,2,17]) → True
exact_change(42, [18,21,22]) → False
exact_change(42, [40,17,1,7]) → False
exact_change(20, [16,3,2,17]) → True

Use it!
Lose it!
... and here

decipher('Weet bksa ed xecumeha 3!')

kxn rkfo k qbokd goouoxn ...

decipher('Weet bksa ed xecumeha 3!')

Good luck on homework 3!

kxn rkfo k qbokd goouoxn ...

and have a great weekend ...