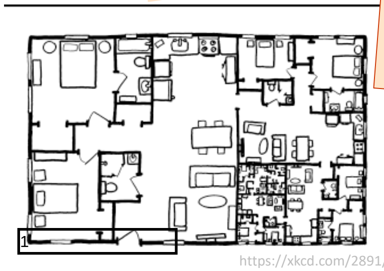
Today in CS5:

chr(9829)

The ♥ of CS (and CSers...)

Algorithms!


https://xkcd.com/2891/

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

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

Use it!

it

'the rest'

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

keep L[0]
+ remove e from the rest

`[5,7,8]`

it

'the rest'

remAll('q','~~q~~aqqlqqiqqiiqeqqnqs')

drop L[0]
+ remove e from the rest

`'aliiiens'`

Lose it!

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

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(8, [7,8,9,8]) ➡ [7,9]

0 1 2 3

sharpening our model for <u>where</u> + <u>how</u> 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?

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
                              it
    elif s[0]
```

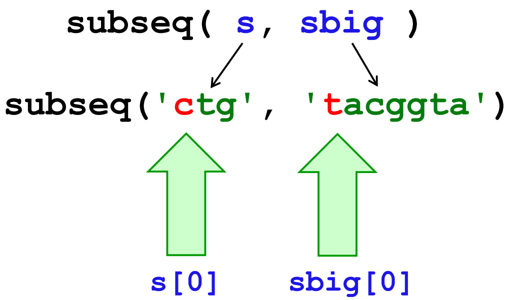
Base case(s)

but first, *algorithms!*

Recursive step(s)

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

Subseq ~ *thinking* it out...



Use it!

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

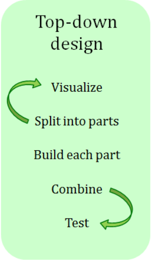
How would we describe it in terms of the inputs?

- or -

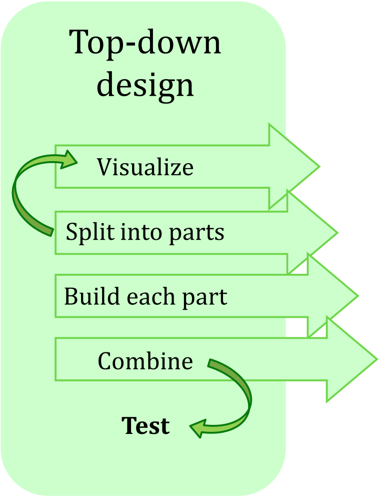
Lose it!

What is left after handling this piece?

Are there other functions we will need?



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



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

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 **gen**eral list **sort**ing — blsort(L), gensort(L)

exact_change making — exact_change(t, L)

hw3pr2: *use it or lose it*

Longest Common Subsequence

LCS(S, T)

'BONOBO'

'CHIMPANZEE'

'CGCTGAGCTAGGCA...'

'ATCCTAGGTAACTG...'

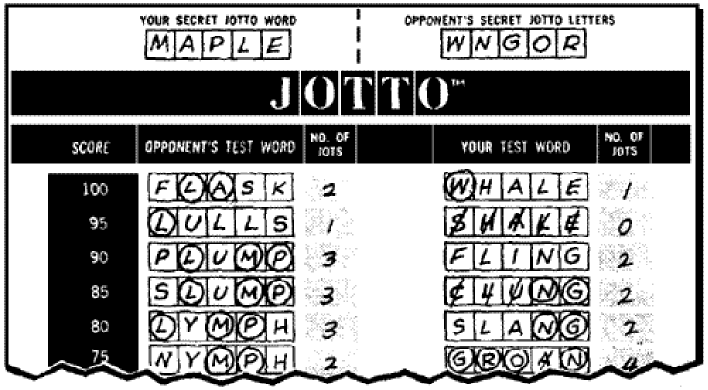
$+10^9$ more

Eye oneder if this haz other aplications?

also in hw3pr2: *Jotto* !

a word-guessing game...

jscore(S, T)



YOUR SECRET JOTTO WORD: MAPLE

OPPONENT'S SECRET JOTTO LETTERS: WNGOR

JOTTO™

SCORE	OPPONENT'S TEST WORD	NO. OF JOTS	YOUR TEST WORD	NO. OF JOTS
100	FLASK	2	WHALE	1
95	LULLS	1	SHAKE	0
90	PLUMP	3	FLING	2
85	SLUMP	3	CLUNG	2
80	LYMPH	3	SLANG	2
75	NYMPH	2	GROAN	4

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?

don't write any code for these...

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

min

remOne

how are min and remOne used?

do answer examples + *brainstorm*

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') →

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

(blank)

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
```