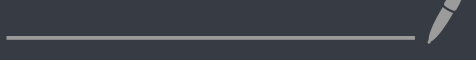


TUT 1

CSC236 is mainly talking about inductive thinking

What is

How it works

What are its forms

Inductive thinking ┬ proof
                   └ recursion (helps with proof and how it works)

EX 1: 普通递归

```
def A(n):
    if n ≤ 1: return 2
    else:
        r = A(n-1)
        s = A(n-2)
        return 2 * r + s
```

#trace A(4)

A(4) = A(3)*2 + A(2)

= (A(2)*2 + A(1))*2 + (A(1)*2 + A(0))

= ((A(1)*2 + A(0))*2 + A(1))*2 + (A(1)*2 + A(0))

= 28 + 6

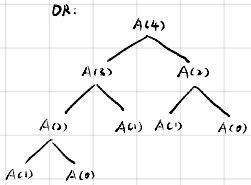
= 34

structure method

Try:

A(4) = 34
- A(3) = 14
  - A(2) = 6
    - A(1) = 2
    - A(0) = 2
  - A(1) = 2
- A(2) = 6
  - A(1) = 2
  - A(0) = 2

OR:



EX 2: 普通递归 2

```
def B(n):
    if n == 1: return 1
    else  return B(n//2) - B((n+1)//2)
```

#trace B(5)

- B(5) = -1
  - B(2) = 0
    - B(1) = 1
    - B(1) = 1
  - B(3) = 1
    - B(1) = 1
    - B(2) = 0
      - B(1) = 1
      - B(1) = 1

EX 3: 元参数，可能有多输出递归

```
4  def E():
5      if maybe(): return "1"
6      elif maybe(): return "2"
7      elif maybe(): return "-" + E()
8      else :        return E() + "^" + E()
```

show a potential trace of E() that contains exactly two recursive calls

① case 1: maybe() = true
return "1"

② case 2: maybe() = false
recursion error ✗

E() 8≅ '2^2'
- E() 6≅ '2'
- E() 6≅ '2'

E() 7≅ '--1'
- E() 7≅ '-1'
  - E() 5≅ 1

For each code fragment below, trace each call using the format presented in class. (To simplify your trace, you may *omit* the values of parameters that remain unchanged through *every* recursive call. You may also abbreviate the name of the algorithm by its first letter.)

Then, trace each algorithm on more examples, to explore the range of behaviours it exhibits. Do any of the algorithms "break" (trigger an error or fall into an infinite recursion)? Can you find argument values to produce specific output values? Etc.

1.

```
0 # Pre(n): n ∈ ℕ and n ⩾ 2.
1 # Return b such that Post(b,n): b ∈ ℕ and (... omitted ...)
2 def algo(n):
3     if n == 2: return 6
4     if n == 3: return 9
5     if n == 4: return 12
6     a = algo(n − 2)
7     return a + 6
8
9 # trace algo(7)
```

algo(7) = 21
algo(5) = 15
algo(3) = 9

2.

```
0 # Pre(n): n ∈ ℕ and n ⩾ 2.
1 # Return b such that Post(b,n): b ∈ ℕ and (... omitted ...)
2 def algo(n):
3     if n == 2: return 2 * 2
4     r = algo(n − 1)
5     s = algo(n − 2)
6     return r + s
7
8 # trace algo(4)
```

algo(4) =
algo(3)
algo(2) = 4
algo(1) = ... } never stop recursion
algo(2) = 4

3.

```
0  # Pre(x,L,b,e): x is a comparable value, and L is a non-empty list of
1  #   comparable values, and b,e ∈ ℕ, and 0 ⩽ b < e ⩽ len(L).
2  # Return c s.t. Post(c,x,L,b,e): c ∈ ℕ and (... omitted ...)
3  def count(x, L, b, e):
4      if e == b + 1:
5          if L[b] < x: return 1
6          return 0
7      c = (2 * b + e + 2) // 3
8      return count(x, L, b, c) + count(x, L, c, e)
9
10 # trace count(3, [3,1,4,1,5,9], 0, 5)
```

c(0,5) = 2
c(0,2) = 1
c(0,1) = 0
c(1,2) = 1
c(2,5) = 1
c(2,3) = 0
c(3,5) = 1
c(3,4) = 1
c(4,5) = 0

4.

```
0 # Pre(x,L,b,e): L is a sorted list and b,e ∈ ℕ and 0 ⩽ b ⩽ e ⩽ len(L)
1 # Return r s.t. Post(r,x,L,b,e): r is boolean and (... omitted ...)
2 def binsearch(x, L, b, e):
3     if b == e: return False
4     m = (b + e) // 2
5     if x < L[m]: return binsearch(x, L, b, m)
```

```
def binsearch(x, L, b, e):
    if b == e: return False
    m = (b + e) // 2
    if x < L[m]: return binsearch(x, L, b, m)
    if x > L[m]: return binsearch(x, L, m, e)
    return True

# trace binsearch(41, [31,41,65,92], 0, 4)
```

b(0,4) = True
b(0,2) = True

5.

```
# Pre(x,L,b,e): L is a sorted list and b,e ∈ ℕ and 0 ⩽ b ⩽ e ⩽ len(L)
# Return r s.t. Post(r,x,L,b,e): r is boolean and (... omitted ...)
def binsearch(x, L, b, e):
    if b == e: return False
    m = (b + e) // 2
    if x < L[m]: return binsearch(x, L, b, m)
    if x > L[m]: return binsearch(x, L, m + 1, e)
    return True

# trace binsearch(314, [123,156,234,289,300,320,456,567,678,789], 0, 10)
```

b(0,10) = False
b(0,5) = False
b(3,5) = False
b(5,5) = False

6.

```
# Pre(n,m): n, m ∈ ℕ and n < 2**m.
# Return b s.t. Post(b,n,m): b is a tuple of length m, and n = Σ_{i=0}^{m-1} b[i]·2^i.
def r(n, m):
    if m == 0: return ()
    return (n % 2,) + r(n // 2, m - 1)

# trace r(11, 5)
```

r(11,5) = (1,1,0,1,0)
r(5,4) = (1,0,1,0)
r(2,3) = (0,1,0)
r(1,2) = (1,0)
r(0,1) = (0,)
r(0,0) = ()

7.

```
# Return b s.t. Post(b): b is a boolean.
def maybe():  # ... implementation omitted ...

# If r is returned, then Post(r): r∈ℕ and (... omitted ...)
def V():
    if   maybe(): return 3
    elif maybe(): return V() ** 2
    else:         return V() // 3
```

① V() = 3
② V() = 9

V = 3

8.

```
# Return b s.t. Post(b): b is a boolean.
def maybe():  # ... implementation omitted ...

# If r is returned, then Post(r): r is a string or a tuple.
def T():
    if   maybe(): return '1'
    elif maybe(): return '2'
    elif maybe(): return ('-', T())
    else:         return (T(), '*', T())
```

① T() = ('1', '*', '-', '2')
T() = '1'
T() = ('-', '2')
T() = '2'

A Change-Making Algorithm

Consider a country that has only 3 and 7 "dollar" bills. Which amounts of money can be made from those bills? E.g., we could take two 3 dollar bills and five 7 dollar bills, to make $2 \cdot 3 + 5 \cdot 7 = 41$ dollars.

Mathematically: for which natural numbers n are there (natural number) a and b for which $a \cdot 3 + b \cdot 7 = n$? Symbolically: for which $n \in \mathbb{N}$ is $\exists\, a, b \in \mathbb{N}$, $n = a \cdot 3 + b \cdot 7$ true?

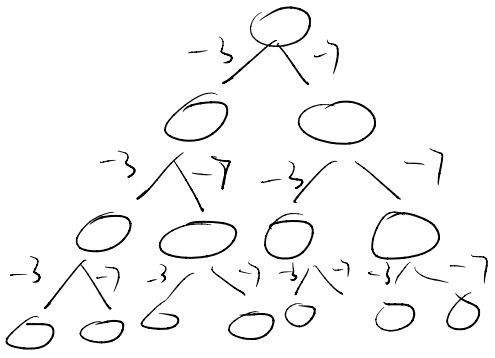
Let's try to solve this problem recursively, including an algorithm to produce a pair a and b given such an n. We'll start "bottom-up", solving it for $n = 0, 1, 2, \ldots$ in sequence, looking for instances of the problem that are solvable with the help of smaller instances.

$n \in \mathbb{N}$	solvable?	a solution	thoughts
0	Yes	$0 \cdot 3 + 0 \cdot 7$	no change is made
1	No		less than smallest bill
2	No		
3	Yes	$1 \cdot 3 + 0 \cdot 7$	multiple change
4	No		
5	No		
6	Yes	$2 \cdot 3 + 0 \cdot 7$	multiple of 3, use one more 3 than before
7	Yes	$0 \cdot 3 + 1 \cdot 7$	
8	No		
9	Yes	$3 \cdot 3 + 0 \cdot 7$	multiple of 3, one more 3 than before
10	Yes	$1 \cdot 3 + 1 \cdot 7$	thinking in multiples of 3: one more 3 than for 7
11	No		
12	Yes	$4 \cdot 3 + 0 \cdot 7$	multiple of 3, one more 3 than before
13	Yes	$2 \cdot 3 + 1 \cdot 7$	one more 3 than "before", i.e., for 10
14	Yes	$0 \cdot 3 + 2 \cdot 7$	maybe think in multiples of 7 as well?
15	Yes	$5 \cdot 3 + 0 \cdot 7$	one more 3 than before, i.e., for $15 - 3$
16	Yes	$3 \cdot 3 + 1 \cdot 7$	one more 3 than before, i.e., for $16 - 3$
17	Yes	$1 \cdot 3 + 2 \cdot 7$	one more 3 than before, i.e., for $17 - 3$
18	Yes	$6 \cdot 3 + 0 \cdot 7$	one more 3 than before, i.e., for $18 - 3$
⋮	⋮		⋮

Once we made three in a row, i.e., 12, 13, 14, we could solve the next three by adding one more 3, then we could make the next three, the next three, and so on.

Let's focus on solving the problem for just $n \geq 12$ (the rest can be a small number of cases handled specially). Then this sort of thinking might be familiar from your previous experience with Simple Induction, starting with a base case of 12 (or maybe three base cases 12, 13, 14?).

But notice that to make, e.g., 16, we don't have to solve each of 12, 13, 14, 15. Switching to recursive thinking: we would start with 16, then ask for a solution to 13. For, e.g., 20, we would ask for a solution to $20 - 3 = 17$, which would ask for a solution to $17 - 3 = 14$. For, e.g., 236, we would ask for a solution to 233, which would ask for a solution to 230, which would ..., and since we're subtracting 3 each time we expect to hit one of 12, 13, 14 eventually. When we get back an a and b for which $233 = a \cdot 3 + b \cdot 7$ we would solve 236 with $(a + 1) \cdot 3 + b \cdot 7$.



A Python function implementing that algorithm is:

```
# Pre(n): n ∈ ℕ with n ≥ 12.
# Return a, b (in a tuple) such that Post(a, b, n): a,b ∈ ℕ with n = 3·a + 7·b.
def change(n):
    if n == 12: return (4, 0)
    elif n == 13: return (2, 1)
    elif n == 14: return (0, 2)
    else:
        (a, b) = change(n - 3)
        return (a + 1, b)
```

claim:
$(a_n, b_n) = (a_{n-3}+1, b_{n-3})$
for $\forall\, n \in \mathbb{N}$, s.t. $n \geq 15$

We'll often return multiple values by bundling them in a single tuple, but still talk about them as multiple values. For the time being our recursive functions will have explicit returns and no observable side-effects (mutation of argument data, or I/O). So their specifications will be a precondition predicate on the arguments along with a postcondition predicate relating the return values to those arguments.

Starting from the specification of change we can read it as saying that the function constructs witness pairs for the claim that

$$\forall\, n,\ Pre(n) \Rightarrow \exists\, a, b,\ Post(a, b, n)\text{, i.e.,}$$

$$\forall\, n \in \mathbb{N},\ n \geq 12 \ \Rightarrow\ \exists\, a, b \in \mathbb{N},\ n = 3 \cdot a + 7 \cdot b.$$

Thinking of the pair of returned values for n as a_n and b_n, and using mathematical notation, we can express the implementation as defining a pair of sequences $a_{12}, a_{13}, \ldots$ and $b_{12}, b_{13}, \ldots$ by

$$(a_{12}, b_{12}) = (4, 0)\text{,}\ (a_{13}, b_{13}) = (2, 1)\text{,}\ (a_{14}, b_{14}) = (0, 2)\text{, and}$$

$$(a_n, b_n) = (a_{n-3} + 1, b_{n-3})\ \text{ for } n \in \mathbb{N} \text{ with } n \geq 15.$$

A proof of correctness of the function can be interpreted as a *constructive* proof of the claim, i.e., that those constructed sequences produce witnesses for the existentials, i.e.,

$$\forall\, n \in \mathbb{N},\ n \geq 12 \ \Rightarrow\ a_n, b_n \in \mathbb{N} \ \wedge\ n = 3 \cdot a_n + 7 \cdot b_n.$$

Trace change with enough arguments so you can picture the call trees and calculation of the return value in general.

CORRECTNESS

To prove the correctness of a function we need to prove that if someone makes a valid call of it, i.e., with valid arguments, i.e., with arguments satisfying the precondition, then it:

- Terminates normally, i.e., returns, rather than
  - stopping due to an error
  - executing forever.
- And the then-returned values are correct, i.e., together with the arguments the values satisfy the postcondition.

no error / loop forever
correct of recursive structure
postcondition is satisfied

We assume in this course that memory is infinite, in particular that there are no out-of-memory / stack-overflow errors.

Invalid Python calls (or operations) will usually stop with an error (raise an exception) rather than executing forever or returning some possibly-incorrect value. This is not necessarily true for the functions we write, including change: when it's called with, e.g., 11, it executes forever.

So to prove change is correct we need to justify that if it's called with a natural $n \geq 12$ then:

- Its Python calls (`n == 12`, `n == 13`, `n == 14`, `n - 3`, `a + 1`, tuple assignment in `(a, b) = change(n-3)`) are valid (so not the source of any kind of incorrectness).
- It doesn't execute forever due to the recursive calls (the only other potential source of infinite execution since there are no loops).
- The then-returned values a, b are natural and $n = 3 \cdot a + 7 \cdot b$.

Let's concentrate on justifying the latter two points.

VALID RECURSIVE STRUCTURE

For a recursive function we'll refer to the *recursive* calls in the code, including their locations (which determines under what conditions they're called) as the *recursive structure*. We'll say that a recursive structure is valid iff its (recursive) calls are always valid and the call tree is always finite: then the structure is neither a source of error nor infinite execution.

There is a *standard form* of justification that a recursive structure is valid. It's indirect, and may seem odd initially for something with the simple structure of change, but it's minimal and generalizes to much more complicated structures. Consider, e.g., calling change with 236: the root argument of the call tree is 236, the child argument is $236 - 3 = 233$, which is smaller than 236, then its child argument is smaller, and so on. If the calls are valid then these are also all naturals. A decreasing sequence of naturals must end.

In previous courses you've recursed on arguments such as lists and trees, making sure to recurse on only "smaller" ones. So the arguments might not be natural numbers, so we always specify some natural-number size measure for the (valid) arguments, and then justify that all recursive calls are on valid smaller (according to that measure) arguments. Then the sequence of sizes of the arguments in any branch is finite. Note we *do not* argue that every branch reaches certain specific base cases — although (assuming non-recursive calls are valid so they don't interfere) we can *deduce this must happen* after our standard justification that the call tree is always finite.

We'll put the justification that the recursive structure for change is valid into the code itself:

```
# Pre(n): n ∈ ℕ with n ≥ 12.
# Return a, b such that Post(a, b, n): a,b ∈ ℕ with n = 3·a + 7·b.
def change(n):

    # Let size(n) = n.
    # valid [to use size as a] measure: n ∈ ℕ (by Pre).

    if n == 12: return (4, 0)
    elif n == 13: return (2, 1)
    elif n == 14: return (0, 2)

    else:
        # (C) n ≥ 15
        # ∵ n ∈ ℕ and n ≥ 12 (by Pre), and n ≠ 12, 13, 14.

        # valid [arguments in] call [change(n - 3)]
        #
        # I.e., Pre(n - 3) ≡ n - 3 ∈ ℕ and n - 3 ≥ 12:
        #
        # n - 3 ≥ 15 - 3 (by C) = 12,
        # and n ∈ ℕ (by Pre) so n - 3 ∈ ℤ so n - 3 ∈ ℕ ∵ n - 3 ≥ 12.
        #
        # [This justified that n - 3 ∈ ℕ by a "closure" property of ℤ
        # (subtracting integers produces an integer) and then a bound.
        # We could also use a closure property for ℕ: subtracting a natural
        # from a natural that is at least as large produces a natural:
        # n - 3 ∈ ℕ since n,3 ∈ ℕ and n ≥ 3 (by C)]
        #
        # [and] valid [to use] recursively [in current change(n) call]
        #
        # size(n - 3) = n - 3 < n = size(n).
        #
        (a, b) = change(n - 3)

        return (a + 1, b)
```

Notice that, along with the annotations required to justify the validity of the recursive structure, we added an annotation making the condition of the else-branch more explicit (but it doesn't completely contain the information that n is natural) and labeled it, for use in the reasoning.

Here is our general template for annotating a recursive function's recursive structure to justify that the structure is valid:

```
# Pre(v̄): ⋯
# Return ⋯
def f(v̄):

    # Let size(v̄) = ⋯.
    # valid [to use size as a] measure:
    # size(v̄) ∈ ℕ ∵ ⋯

    ⋯

    # valid [arguments in] call [f(ā)]:
    # Pre(ā) ∵ ⋯
    # [and] valid [to use] recursively [in current f(v̄) call]:
    # size(ā) < size(v̄) ∵ ⋯
    ⋯ f(ā) ⋯

    ⋯
```

Here we're using $\overline{v}$ to mean some sequence of parameter variables, and $\overline{a}$ some sequence of argument expressions.

We can also include other annotation to support the reasoning (e.g., a more explicit condition).

We'll usually leave out the [bracketed] parts of the labels from now on.

For the time being, our recursive functions will only recurse on themselves directly, i.e., not call functions that call them back, and if they also contain loops those loops will have explicit bounds. So if the recursive structure is valid, and non-recursive calls are valid, the function returns.

Overall correctness:

① Valid Recursive Structure

② Postcondition

Goal of ① is to justify each valid call terminates normally.
(no error and no infinite call)

How to check ①?

A. Define a size function (aka. "measure") over the argument
   - size is a "valid measure" $\equiv$ size $\in \mathbb{N}$ (for each valid input)

B. Each call (recursive or not) is valid

C. Each recursive call is "valid recursively" meaning size of argument to recursive call is strictly less than size of current argument