# Lecture 13, 08 November 2021

## Induction on "structures"

- A list consists of the first element and the rest
- Base case is usually the empty list `[]`
- May occasionally also have a base case for a singleton list

```
1 def mylength(l):
2   if l == []:
3     return(0)
4   else:
5     return(1 + mylength(l[1:]))
```

```
1 def mysum(l):
2     if l == []:
3         return(0)
4     else:
5         return(l[0] + mysum(l[1:]))
```



- Alternate between ascending and descending
- Two possibilities
  - up-down-up-down..., `[1,3,2,7,1,5]`
  - down-up-down-up..., `[8,2,18,-5,7,2,8]`

### Up-down

- If `len(l)` is 0 or 1, nothing to do
- Up-down unit repeats after two elements
- 2 element list, check up-down-up
- Recursively check that `l[2:]` is also up-down

### Down-up is symmetric

### Combine to get zigzag

```
 1 def updown(l):
 2   if len(l) <= 1:
 3     return(True)
 4   elif len(l) == 2:
 5     return(l[0] < l[1])
 6   else:
 7     return(l[0] < l[1] and l[1] > l[2] and updown(l[2:]))
 8
 9 def downup(l):
10   if len(l) <= 1:
11     return(True)
12   elif len(l) == 2:
13     return(l[0] > l[1])
14   else:
15     return(l[0] > l[1] and l[1] < l[2] and downup(l[2:]))
16
17 def zigzag(l):
18   return(updown(l) or downup(l))
```

## Mutual recursion

- Can define `updown` and `downup` in terms of each other
- **Mutual recursion**

```
def zigzag(l):
    return(updown(l) or downup(l))

def updown(l):
    if len(l) < 2:
        return(True)
    else:
        return(l[0] < l[1] and downup(l[1:]))

def downup(l):
    if len(l) < 2:
        return(True)
    else:
        return(l[0] > l[1] and updown(l[1:]))
```

- Function must be defined before it can be called
- ifferent from executing it
- e error is flagged only when the function is executed



```
def fnwitherror():
  return(thisisanewname)

```

- The function definition above does not generate an error, though `thisisanewname` is undefined
- The function call below generates the error

```
fnwitherror()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-6-69ca489906bb> in <module>()
----> 1 fnwitherror()

<ipython-input-5-9c2565e3b99b> in fnwitherror()
      1 def fnwitherror():
----> 2   return(thisisanewname)
      3

NameError: name 'thisisanewname' is not defined
```

SEARCH STACK OVERFLOW

- Similarly, if we try to execute `updown` before we define `downup` we get an `NameError`

```
# Remove the earlier definitions and redefine
del(zigzag)
del(updown)
del(downup)

def zigzag(l):
    return(updown(l) or downup(l))

def updown(l):
    if len(l) < 2:
```

```
10     if len(l) < 2:
11         return(True)
12     else:
13         return(l[0] < l[1] and downup(l[1:]))
14 
15 updown([1,3,1])
16 
17 def downup(l):
18     if len(l) < 2:
19         return(True)
20     else:
21         return(l[0] > l[1] and updown(l[1:]))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-8-47cc435a8dd7> in <module>()
      2 del(zigzag)
      3 del(updown)
----> 4 del(downup)
      5 
      6 def zigzag(l):

NameError: name 'downup' is not defined
```

SEARCH STACK OVERFLOW

## More recursive functions on lists

Saved successfully!

### find(l,v)

Check `v` is a member of `l` -- like built-in `v in l`

- Base case, if `l == []` then `v` is not found
- If `l[0] == v` then `v` is found
- Otherwise, inductively search for `v` in `l[1:]`

```
1 def find(l,v):
2 
3   if l == []:
4     return(False)
5   if l[0] == v:
6     return(True)
7   else:
8     return(find(l[1:],v))
```

## Short cut evaluation of boolean expressions

- If we write A or B, we evaluate A and B and then check if at least one is true
- In what order are A and B evaluated?
- Python (and other languages) **always** evaluate left to right
- And stop when then answer is known
- Here is a version of `find` in which the two cases of the inductive step are combined using `or`

```
1 def find2(l,v):
2   if l == []:
3     return(False)
4   else:
5     return((l[0] == v) or find2(l[1:],v))
6     # Unwinds as l[0] == v or l[1] == v or l[2] == v or ... or l[len(l)-1] == v
```

```
l1 = list(range(0,100,3))
```

```
l2 = [j for j in range(0,100,5) if find(l1,j)]
print(l2)
```

```
[0, 15, 30, 45, 60, 75, 90]
```

```
l2 = [j for j in range(0,100,5) if find2(l1,j)]
print(l2)
```

```
[0, 15, 30, 45, 60, 75, 90]
```

## `insert(l,v)

Insert `v` in `l`, assume `l` is sorted in ascending order

- If `l == []` return singleton list `[v]`
- If `v < l[0]` return `[v] + l`
- If `l[0] <= v`, inductively insert `v` in `l[1:]` and stick `l[0]` before this list

```
def insert(l,v): # Assume l sorted in ascending order
  if l == []:
    return([v])
  if v < l[0]:
```

Saved successfully!

```
    return(l[:1] + insert(l[1:],v))
    # same as
    # return([l[0]] + insert(l[1:],v))
```

```
l3 = insert(l1,1000)
print(l3)
```

```
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75, 78, 8
```

## delete(l,v)

Delete first occurrence `v` from `l`, if `v` exists

- Similar structure to `insert(l,v)`
- If `l == []` nothing to be done
- If `l[0] == v`, return `l[1:]`
- Otherwise, inductively delete `v` from `l[1:]` and stick `l[0]` before this list

```
def delete(l,v):
  if l == []:
    return(l)
  if l[0] == v:
    return(l[1:])
  else:
    return(l[:1] + delete(l[1:],v))
```

```
l3 = delete(insert(l1,15),13)
print(l3)
```

```
[0, 3, 6, 9, 12, 15, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75, 7
```

## findpos(l,v)

- Like `find(l,v)` but report position of first `v` in `l`
- If `v` is not found in `l` return `-1`
- If `l == []`, return `-1`
- If `l[0] == v`, return `0`
- Otherwise, inductively find the first position of `v` in `l[1:]` and add `1` to account for `l[0]`
- Unless `v` is not found in `l[1:]` in which case the recursive call returns `-1` and this should be passed on untouched

```
def findpos(l,v):  # Returns -1 if v not in l
  if l == []:
    return(-1)
  if l[0] == v:
    return(0)
  else:
    z = findpos(l[1:],v)
    if z >= 0:
      return(1+z)
    else:
      return(z)
```

## Alternative findpos(l,v)

- Return `len(l) + 1` if `v` is not found in `l`
- Saved successfully!
- pler: just add `1` to the recursive call, which works whether or not `v` is found in `l[1:]`

```
def findpos2(l,v):  # Returns len(l)+1 if v not in l
  if l == []:
    return(1)
  if l[0] == v:
    return(0)
  else:
    z = findpos2(l[1:],v)
    return(1+z)
```

```
findpos(l1,17),findpos2(l1,17),len(l1)
```

```
(-1, 35, 34)
```