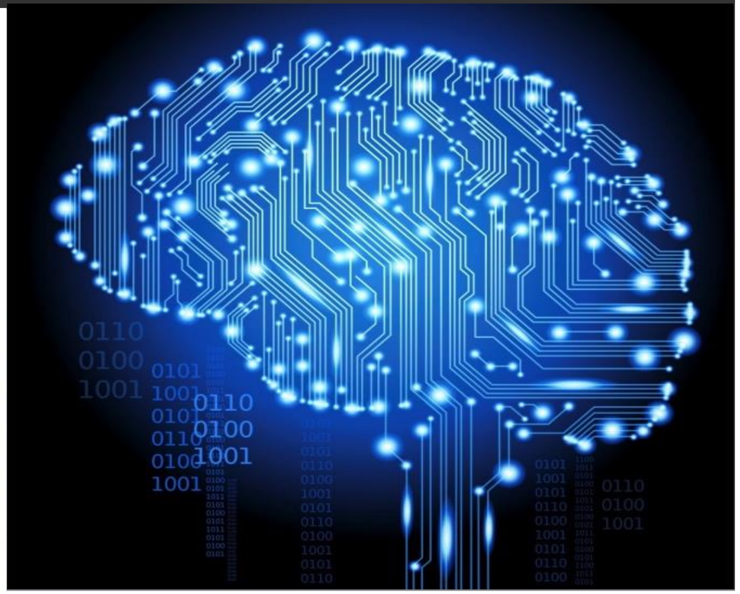
MONASH University

Information Technology

Recursion I

Prepared by Maria Garcia de la Banda
Updated by Brendon Taylor



Objectives for this lecture

- To re-visit the concept of recursive algorithm
- To understand how to implement recursive algorithms
- To be able to reason about their Big O complexity
- To start exploring the relationship between iteration and recursion

Motivation behind Recursion

Revision: recursive algorithms

- **Solve a large problem by reducing it to one or more sub-problems that are:**
  - Of the same kind as the original
  - Simpler to solve
- **Each of the sub-problems is itself solved using the same algorithm …**

… until the sub problems are so "simple" that they can be solved without further reductions (base cases)

Examples

- **To find a route from A to B:**
  - if they are “very close” (e.g., one step), easy to find (the one step);
  - else …
    - find a place C “between” A and B;
    - find a smaller route from A to C;
    - find a smaller route from C to B;
    - put the two routes together
- **To wash up a pile of dirty dishes:**
  - if there are no dishes in the pile, easy to do (stop);
  - else …
    - take one dish, wash it up, and then …
    - wash up the remaining pile of dirty dishes

Feels like iteration…
Hold that thought…

Candidate problems for recursion

1. Must be possible to decompose them into simpler similar problems
2. At some point, the problems must become so simple that can be solved without further decomposition
3. Once all subproblems are solved, the solution to the original problem can be computed by combining these solutions

General recursive structure

- **That of a function that calls itself (directly or via others):**

```
def solve(problem):
    if problem is simple:           } Base case(s)
        Solve problem directly
    else:
        Decompose problem into subproblems p1, p2,…
        solve(p1)
        solve(p2)   } Recursive calls
        solve(p3)
        …
        Combine the subsolutions to solve problem
```

Thinking about recursion

- **As a programmer all you need to know is how to:**
  - Detect and solve the base cases
  - Decompose the problem into simpler subproblems
    - That converge to the base cases
  - Combine the sub-solutions

Recursion example
Factorials

Example: factorials

- Determine the number of permutations of a given number of distinct elements
- For example: consider the letters

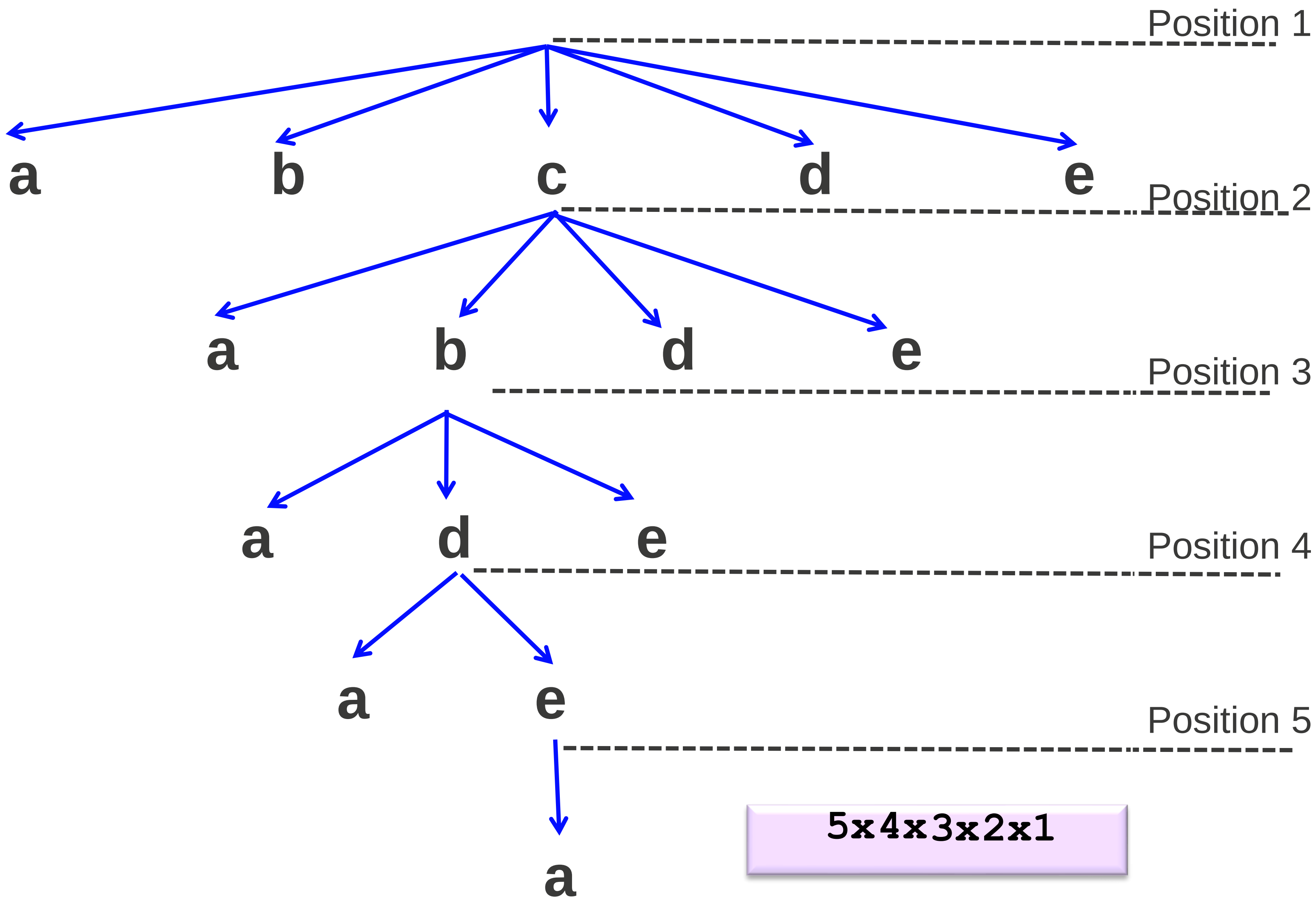
  a b c d e

- How many permutations of these 5 letters can we make?

  $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

  since there are 5 choices for the first letter, 4 for the second, 3 for the third, etc.

How many permutations?



Factorials: how do we program it?

- **We assume n ≥ 0 and factorial of 0 = 1**
- **We start by looking at examples:**

```
0! = 1
1! = 1
2! = 1*2
3! = 1*2*3
4! = 1*2*3*4
…
n! = 1*2*3*4*…*(n-1)*n
```

Easy to implement using iteration (a loop)

Factorial: an iterative approach

```
def factorial(n: int) -> int:
    result = 1
    for i in range(1,n+1):
       result = result * i
    return result
```

`n! = 1*2*3*4*…*(n-1)*n`

What happens if n = 0?

And if n < 0?

- **What is the value of `result` before and after each iteration if n=5?**

```
1*1;
1*2;
2*3;
6*4;
24*5;
```

result	1	1	2	6	24	120
i	before	1	2	3	4	5

thus, it would correctly return `result=120`

Complexity? O(n)

Factorials: what about recursively?

- **We start by looking at examples:**

```
0! = 1
1! = 1
2! = 1*2
3! = 1*2*3
4! = 1*2*3*4
…
n! = 1*2*3*4*…*(n-1)*n
```

$$n! = \underbrace{1*2*3*4*\ldots*(n-1)}_{(n-1)!}*n$$

```
n! = (n-1)! * n
```

We can easily code this by using a recursive method

How does it converge?

n-1

So the recursive call needs to have n-1 as argument

How does it combine solutions?

*

So the result of the recursive call needs to be multiplied

Base case?

0? 1? Both?

We need to answer 0. The result of 0! is 1 which can be combined. So no need to stop at 1.

Factorial: recursive approach

```
def factorial(n: int) -> int:
    if n == 0:  # base case
        return 1
    else:
        return n*factorial(n-1) # recursive call
```

convergence

Important: same type!

combination

What would the execution be like? Cascading calls

5 * factorial(4)

4 * factorial(3)

3 * factorial(2)

2 * factorial(1)

1 * factorial(0)

1

Each call would push its stack frame in MIPS

Factorial: recursive approach

```
def factorial(n: int) -> int:
    if n == 0:  # base case
        return 1
    else:
        return n*factorial(n-1) # recursive call
```

convergence

Important: same type!

combination

What would the execution be like? Cascading calls

5 * factorial(4)

4 * factorial(3)

3 * factorial(2)

2 * factorial(1)

1 * 1

Factorial: recursive approach

```
def factorial(n: int) -> int:
    if n == 0:  # base case
        return 1
    else:
        return n*factorial(n-1) # recursive call
```

convergence

Important: same type!

combination

What would the execution be like? Cascading calls

$$5 * \underbrace{factorial(4)}_{4 * \underbrace{factorial(3)}_{3 * \underbrace{factorial(2)}_{2 * 1}}}$$

Factorial: recursive approach

```
def factorial(n: int) -> int:
    if n == 0:  # base case
        return 1
    else:
        return n*factorial(n-1) # recursive call
```

convergence

Important: same type!

combination

What would the execution be like? Cascading calls

5 * factorial(4)

4 * factorial(3)

3 * 2

Factorial: recursive approach

```
def factorial(n: int) -> int:
    if n == 0:  # base case
        return 1
    else:
        return n*factorial(n-1) # recursive call
```

convergence

Important: same type!

combination

What would the execution be like? Cascading calls

$5 * \underbrace{factorial(4)}_{4 * 6}$

Factorial: recursive approach

```
def factorial(n: int) -> int:
    if n == 0:  # base case
        return 1
    else:
        return n*factorial(n-1) # recursive call
```

convergence

Important: same type!

combination

What would the execution be like? Cascading calls

5 * 24

Factorial: recursive approach

```
def factorial(n: int) -> int:
    if n == 0:   # base case
        return 1
    else:
        return n*factorial(n-1) # recursive call
```

convergence

Important: same type!

combination

What would the execution be like? Cascading calls

120

All stack frames except original factorial(5) are finished

Complexity?

The same: O(n)

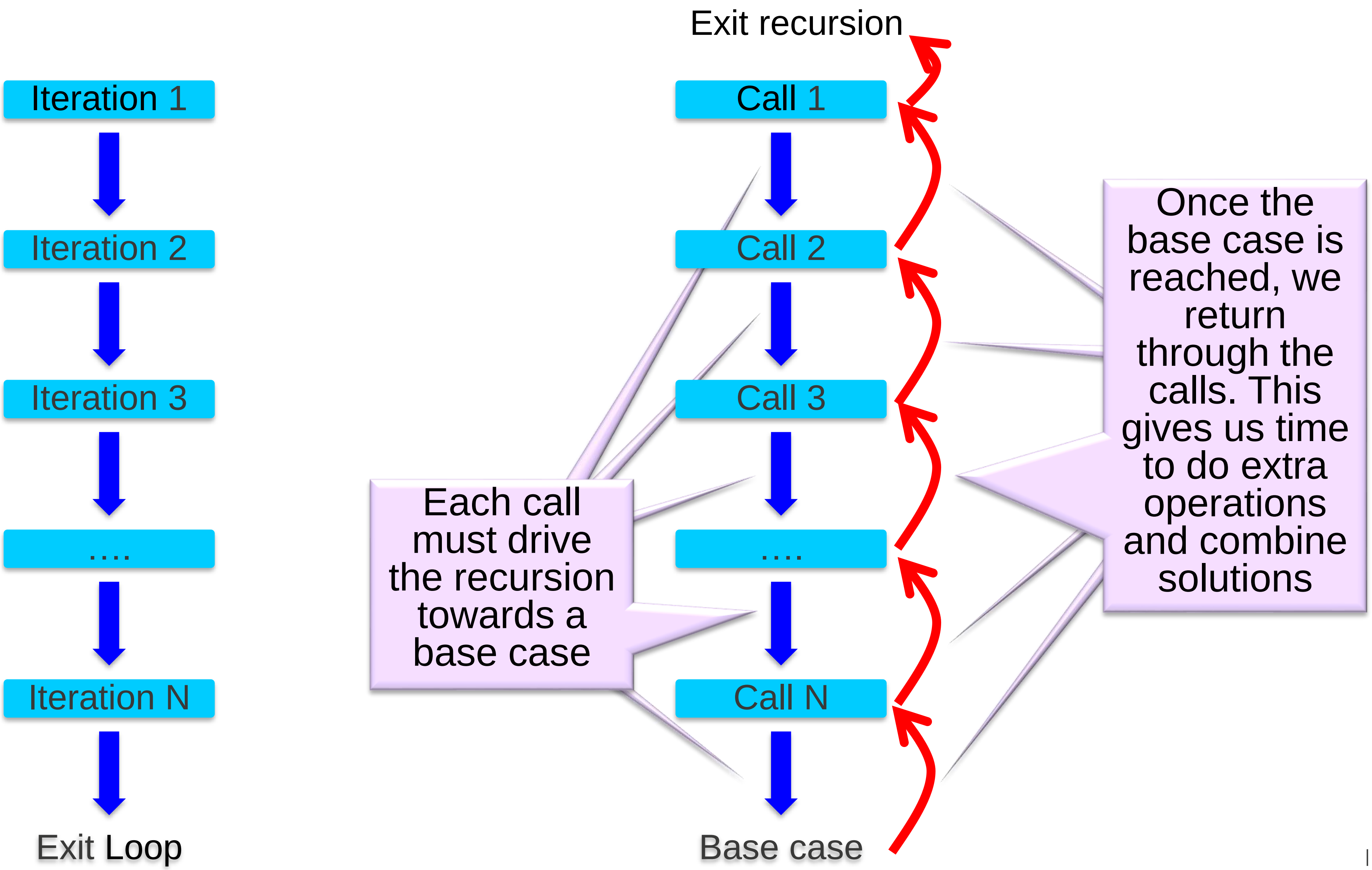
Recursive procedure/method

- **Must have the following components:**
  1. At least one base case
  2. At least one recursive call whose result is combined
  3. Convergence to base case (must be “simpler”)
- **In factorial:**
  1. `if n==0:`
  2. `factorial(n-1)` and `*`
  3. `(n-1)`
- **What happens if**
  - no base case?
  - no convergence? (e.g., we code `n*factorial(n)`)

```
def factorial(n: int) -> int:
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
```

Iteration versus Recursion

Iteration versus (linear) Recursion



Recursion versus iteration

To iterate is human, to recurse divine – Anonymous

- **Can every iterative function be implemented using recursion?**
  - Yes, it is straightforward
    - Iterations are replaced by function calls
    - The base case is the (negated) condition of the loop
  - Often needs an auxiliary function to prepare the converging arguments (see later)
- **Can every recursive function be implemented using iteration?**
  - Yes, BUT you might also need to store past results in either
    - Accumulators, or
    - A stack (recall how the run-time stack is also used to implement recursive functions)

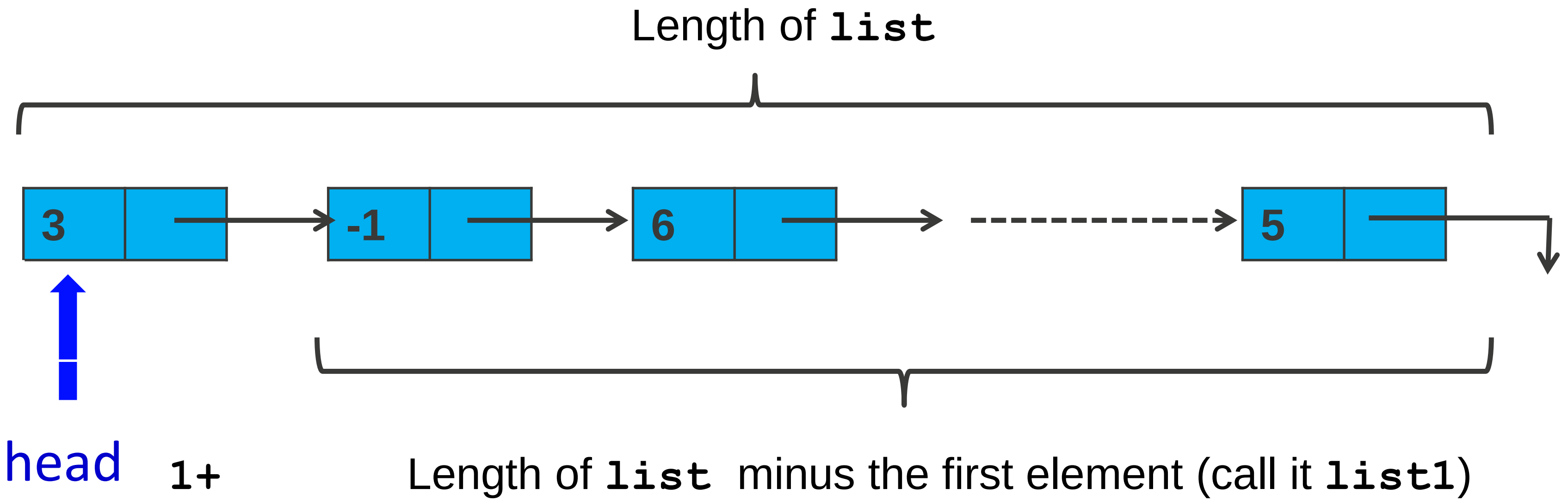
Example: from iteration to recursion

- **Consider an iterative method in `LinkList` to compute the length, if the class did not have `self.length`:**

```
def __len__(self) -> int:
    current = self.head
    count = 0
    while current is not None:
        current = current.link
        count += 1
    return count
```

Complexity? | O(n) where n is the length of the list

- **Let's think how to implement it recursively**



How does it converge?

`list1`

So the recursive call should have **`list1`** as its argument

How? Move `head`? Not allowed. Use **`current`** (starts as a pointer to **`head`**)

Different type! Need an auxiliary method to converge through **`current`**

How does it combine solutions?

+

So the result of the recursive call needs to be added

Base case?

Empty?
One element?
Both?

We need to answer when the list is empty. The result when **`list`** is empty is 0, which can be combined through +. So no need to stop at the last element.

Example: from iteration to recursion

Auxiliary method: sets up the initial parameters (in tis case, the current node). Often required in practice.

```
def __len__(self) -> int:
    return self.len_aux(self.head)


def len_aux(self, current: Node) -> int:
    if current is None: # base case
        return 0
    else:
        return 1 + self.len_aux(current.link)
```

combination

Convergence: pass a pointer to the next node (seen as first node of the remaining list)

Complexity?

Identical: O(n)

Recursion example
Length

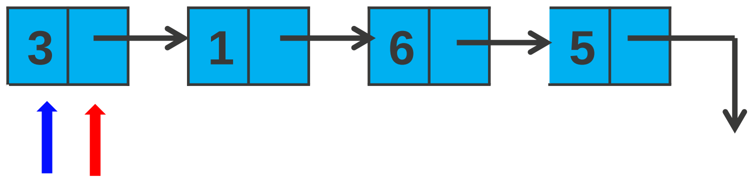
Length: recursive approach

```
def __len__(self) -> int:
    return self.len_aux(self.head)

def len_aux(self, current: Node) -> int:
    if current is None: # base case
        return 0
    else:
        return 1 + self.len_aux(current.link)
```

Execution? Cascading calls

```
len(a_list)
 len_aux(a_list.head)
```



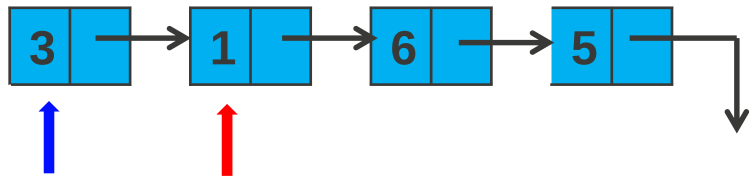
Length: recursive approach

```
def __len__(self) -> int:
    return self.len_aux(self.head)

def len_aux(self, current: Node) -> int:
    if current is None: # base case
        return 0
    else:
        return 1 + self.len_aux(current.link)
```

Execution? Cascading calls

```
len(a_list)
 len_aux(a_list.head)
  1 + len_aux(current.link)
```



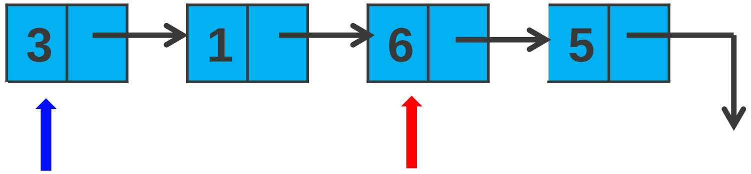
Length: recursive approach

```
def __len__(self) -> int:
    return self.len_aux(self.head)

def len_aux(self, current: Node) -> int:
    if current is None: # base case
        return 0
    else:
        return 1 + self.len_aux(current.link)
```

Execution? Cascading calls

```
len(a_list)
 len_aux(a_list.head)
  1 + len_aux(current.link)
        1 + len_aux(current.link)
```



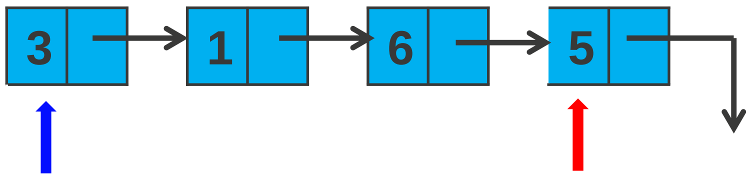
Length: recursive approach

```
def __len__(self) -> int:
    return self.len_aux(self.head)

def len_aux(self, current: Node) -> int:
    if current is None: # base case
        return 0
    else:
        return 1 + self.len_aux(current.link)
```

Execution? Cascading calls

```
len(a_list)
 len_aux(a_list.head)
  1 + len_aux(current.link)
       1 + len_aux(current.link)
            1 + len_aux(current.link)
```



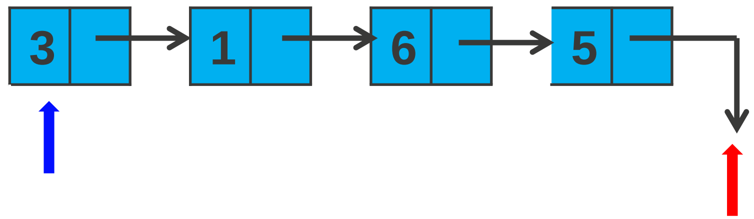
Length: recursive approach

```
def __len__(self) -> int:
    return self.len_aux(self.head)

def len_aux(self, current: Node) -> int:
    if current is None: # base case
        return 0
    else:
        return 1 + self.len_aux(current.link)
```

Execution? Cascading calls

```
len(a_list)
 len_aux(a_list.head)
  1 + len_aux(current.link)
        1 + len_aux(current.link)
              1 + len_aux(current.link)
                    1 + len_aux(current.link)
                                   0
```



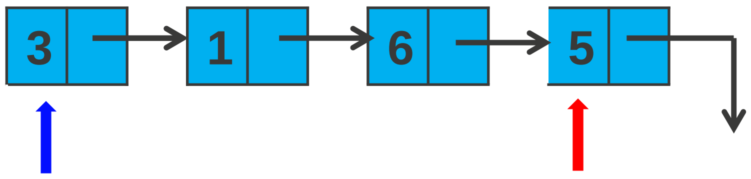
Length: recursive approach

```
def __len__(self) -> int:
    return self.len_aux(self.head)

def len_aux(self, current: Node) -> int:
    if current is None: # base case
        return 0
    else:
        return 1 + self.len_aux(current.link)
```

Execution? Cascading calls

```
len(a_list)
 len_aux(a_list.head)
  1 + len_aux(current.link)
       1 + len_aux(current.link)
            1 + len_aux(current.link)
                 1 +         0
```



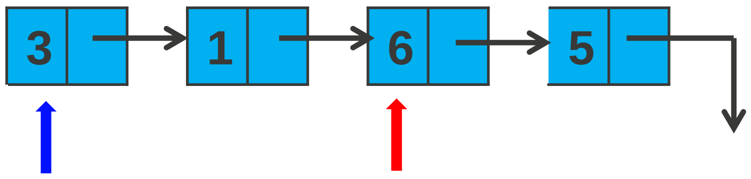
Length: recursive approach

```
def __len__(self) -> int:
    return self.len_aux(self.head)

def len_aux(self, current: Node) -> int:
    if current is None: # base case
        return 0
    else:
        return 1 + self.len_aux(current.link)
```

Execution? Cascading calls

```
len(a_list)
 len_aux(a_list.head)
  1 + len_aux(current.link)
        1 + len_aux(current.link)
              1 +         1
```



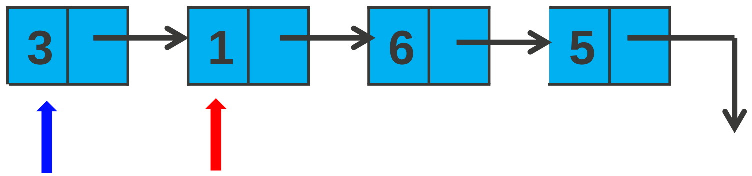
Length: recursive approach

```
def __len__(self) -> int:
    return self.len_aux(self.head)

def len_aux(self, current: Node) -> int:
    if current is None: # base case
        return 0
    else:
        return 1 + self.len_aux(current.link)
```

Execution? Cascading calls

```
len(a_list)
 len_aux(a_list.head)
  1 + len_aux(current.link)
      1 +         2
```



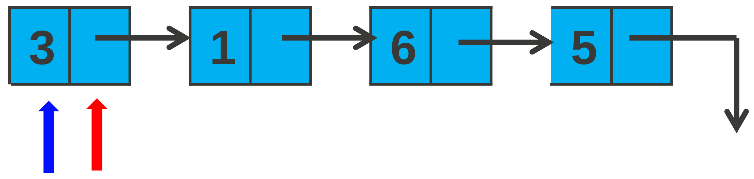
Length: recursive approach

```
def __len__(self) -> int:
    return self.len_aux(self.head)

def len_aux(self, current: Node) -> int:
    if current is None: # base case
        return 0
    else:
        return 1 + self.len_aux(current.link)
```

Execution? Cascading calls

```
len(a_list)
 len_aux(a_list.head)
  1 + 3
```



Length: recursive approach

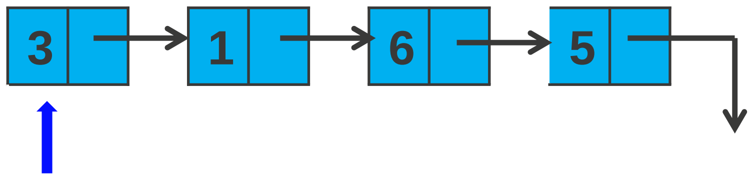
```
def __len__(self) -> int:
    return self.len_aux(self.head)

def len_aux(self, current: Node) -> int:
    if current is None: # base case
        return 0
    else:
        return 1 + self.len_aux(current.link)
```

Execution? Cascading calls

`len(a_list)`

4



Recursion example
Contains

Another example: iteration to recursion

- **Consider an iterative method in `LinkList` for checking if an item is in the linked list or not, again assuming we do not have the length:**
- **Iterative version:**

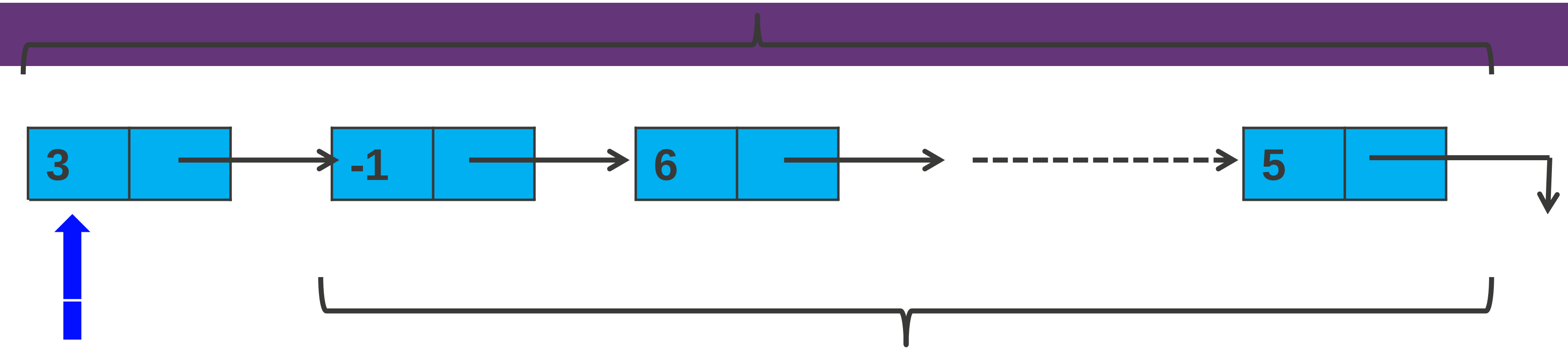
```
def __contains__(self, item: T) -> bool:
    current = self.head
    while current is not None:
        if current.item == item:
            return True
        current = current.link
    return False
```

Complexity?

Another example: iteration to recursion

- **Complexity?**
  - Best case when found first: **O(1)*CompEq** where
    - CompEq is the complexity of == (or __eq__)
    - Often this is O(1)*O(m), so O(m) where m is the maximum size for an item
  - Worst case when not found: **O(n)*CompEq** where
    - n is the length of the list
    - Often this is O(n)*O(m), so O(n*m)

- **Let's think how to implement it recursively** `__contains__(6) in list`



head `6 == head.item or __contains__(6) in list1`

How does it converge?	`list1`	Again we need an auxiliary method that converges through **`current`**
How does it combine solutions?	or	So the result of the recursive call needs to be **`or`**
Base case?	Empty Element found	We need to answer when the list is empty (**`False`**). We can also stop when we found the element (**`True`**)

Another example: iteration to recursion

Again: need an auxiliary to recur through the nodes rather than through the list

```
def __contains__(self, item: T) -> bool:
    return self.contains_aux(self.head, item)

def contains_aux(self, current: Node, item -> T) -> bool:
    if current is None: # base case
        return False
    else:
        return current.item == item or
            self.contains_aux(current.link, item)
```

combination

convergence

Complexity?

Identical

Alternative coding for the same method

```
def __contains__(self, item: T) -> bool:
    return self.contains_aux(self.head, item)

def contains_aux(self, current: Node, item: T) -> bool:
    if current is None: # base case
        return False
    elif current.item == item
        return True
    else:
        return self.contains_aux(current.link, item)
```

`if A True else B` is the same as saying `A or B`

So, the only difference is that **you** are **explicitly** doing the “OR” through the `elif`

Example: add the elements of a queue

Write as a user a recursive method that empties a queue returning the sum of its items. All you need is: `is_empty()` `serve()`

```
def sum_queue(a_queue: Queue[int]) -> int:
    if a_queue.is_empty(): //base case
        return 0
    else:
        return a_queue.serve() + sum_queue(a_queue)
```

convergence

No need for auxiliary function: same type

combination

No need for while/for loops! The recursive calls create the loop!

Another possibility:

```
def sum_queue(a_queue: Queue) -> int:
    result = 0
    if not a_queue.is_empty():
        result = a_queue.serve() + sum_queue(a_queue)
    return result
```

Summary

- **Recursive algorithms are characterised by:**
  1. Existence of base cases
  2. Decomposition into simpler sub-problems
  3. Combination of solutions to sub-problems
- **Recursive methods require:**
  1. One or more base cases
  2. One or more recursive calls
  3. Convergence in the recursive calls
  4. Combination of sub-solutions