

Linked Lists

Announcements

Tree Recursion with Lists

=

```
def longest(s: list[int], n: int) -> list[int] | None:
```

A list of integers or None

```
    """Return the longest sublist of s that sums to n, or None if none exists.
```

```
>>> longest([4, 1, 3, -1, 2, -1], 5)
```

```
[4, 1, -1, 2, -1]
```

```
>>> longest([3, 1, 4], 6)
```

```
None
```

```
"""
```

```
if len(s) == 0:
```

```
    if n == 0:
```

```
        return []
```

```
    else:
```

```
        return None
```

```
minus_first = longest(s[1:], _____)
```

```
without_first = longest(s[1:], n)
```

```
if _____:
```

```
    with_first = _____ minus_first
```

```
    if isinstance(without_first, list):
```

```
        return _____
```

```
    else:
```

```
        return with_first
```

```
else:
```

```
    return without_first
```

Concepts that are useful for this problem:

- A list can be viewed as a first item `s[0]` & the rest `s[1:]`
- Tree recursion is good for searching & counting variants
- To find a sublist, try including/excluding the first item
- Use `isinstance(s, list)` to test whether `s` is a list
- The `max` function can pick the longer of two lists

`[1, -1, 2, -1]` is the longest way in `s[1:]` to make 1

`[1, 3, -1, 2]` is the longest way in `s[1:]` to make 5

`[4, 1, 3, -1, 2]` is a way to make 5

Viewing a Sequence as the First Item and the Rest

```
def longest(s: list[int], n: int) -> list[int] | None:
    """Return the longest sublist of s that sums to n, or None if none exists.
```

A list of integers or None

```
>>> longest([4, 1, 3, -1, 2, -1], 5)
[4, 1, -1, 2, -1]
>>> longest([3, 1, 4], 6)
None
"""
```

```
if len(s) == 0:
    if n == 0:
        return []
    else:
        return None
```

```
minus_first = longest(s[1:], n - s[0])
without_first = longest(s[1:], n)
```

```
if isinstance(minus_first, list):
```

```
    with_first = [s[0]] + minus_first
    if isinstance(without_first, list):
        return max(with_first, without_first, key=len)
    else:
        return with_first
```

```
else:
    return without_first
```

Concepts that are useful for this problem:

- A list can be viewed as a first item `s[0]` & the rest `s[1:]`
- Tree recursion is good for searching & counting variants
- To find a sublist, try including/excluding the first item
- Use `isinstance(s, list)` to test whether `s` is a list
- The `max` function can pick the longer of two lists

`[1, -1, 2, -1]` is the longest way in `s[1:]` to make 1

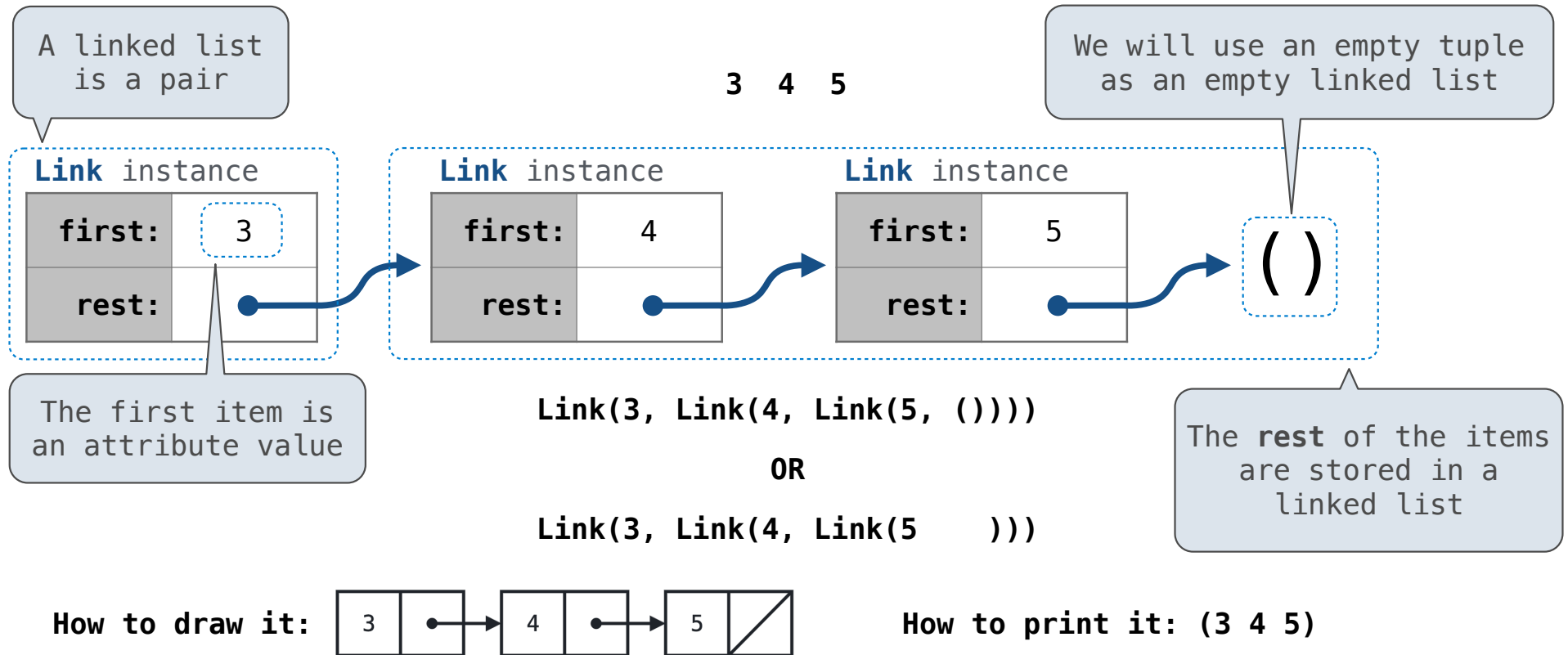
`[1, 3, -1, 2]` is the longest way in `s[1:]` to make 5

`[4, 1, 3, -1, 2]` is a way to make 5

Linked Lists

Linked List Structure

A linked list is either empty **or** a first value and the rest of the linked list



Linked List Data Class

The **Link** data class describes a non-empty **LinkedList**: one link in the chain

```
type LinkedList[T] = Link[T] | tuple[()]
```

The type of an empty tuple

```
@dataclass
```

```
class Link[T]:
```

```
    first: T
```

```
    rest: LinkedList[T] = () # rest defaults to ()
```

A Link object is never empty: it has first and rest

The type variable T describes
the type of the items
contained in the linked list

This is a **Link[int]**
or a **LinkedList[int]**
(or a **Link** or a
LinkedList)

```
Link(3, Link(4, Link(5)))
```

(Demo)

Linked List Processing

How to Work with Linked Lists

For a LinkedList `s`, check if it's a `Link` before accessing its **first** and/or **rest** attributes

```
if isinstance(s, Link):
```

```
    ...
```

In here you can access
`s.first` and `s.rest`

```
if not isinstance(s, Link):
```

```
    else:
```

```
        ...
```

Also in here

Each time you call `Link` on 2 arguments, the second operand must evaluate to a linked list!

- `Link(4, ())` or `Link(4, Link(...))`
- `Link(4, s)` as long as `s` is a name for a linked list
- `Link(4, f(...))` as long as `f` returns a linked list

Construct the rest of the linked list, then add to the front:



```
>>> s = ()
>>> s = Link(5, s)
>>> s = Link(4, s)
>>> s = Link(3, s)
>>> print(s)
(3 4 5)
```

```
Link(3, Link(4, Link(5)))
```

Iteration and Recursion for Linked List Processing

Two approaches to `range_link(3, 6) == Link(3, Link(4, Link(5)))`

```
def range_link(start, end):
    s = ()
    k = end - 1
    while start <= k:
        s = Link(k, s)
        k = k - 1
    return s

def range_link_recursive(start, end):
    if start >= end:
        return ()
    else:
        return Link(start, range_link_recursive(start+1, end))
```

Can recursion start from `k = end - 1` and work backward using `k = k - 1` ?

```
def range_link(start: int, end: int) -> LinkedList[int]:
    def f(s, k):
        if _____:
            return s
        else:
            return _____
    return _____
```

Iteration and Recursion for Linked List Processing

Two approaches to `range_link(3, 6) == Link(3, Link(4, Link(5)))`

```
def range_link(start, end):
    s = ()
    k = end - 1
    while start <= k:
        s = Link(k, s)
        k = k - 1
    return s

def range_link_recursive(start, end):
    if start >= end:
        return ()
    else:
        return Link(start, range_link_recursive(start+1, end))
```

Can recursion start from `k = end - 1` and work backward using `k = k - 1` ?

```
def range_link(start: int, end: int) -> LinkedList[int]:
    def f(s, k):
        if k < start:
            return s
        else:
            return f(Link(k, s), k-1)
    return f(), end-1
```

Example: Partitions

Partitions as Linked Lists

partitions(n,m): list ways to make n by summing positive pieces up to m in increasing order
Summing to n using pieces up to size m involves either:

- (using an m, then) summing to n-m using pieces up to size m, or
- (not using m, then) summing to n using pieces up to size m-1

```
def partitions(n: int, m: int) -> list[LinkedList[int]]:
    """Return a list of partitions of n using parts of up to m.
    Each partition is represented by a linked list.
    """
    if n == 0:
        return [()] # A list containing just the empty partition
    elif n < 0 or m == 0:
        return [] # An empty list with no partitions at all
    else:
        with_m = _____
        without_m = partitions(n, m-1)
        return with_m + without_m # Adding lists creates a longer list of all items in both
```

Useful concepts

- **partitions** returns a list
- List comprehensions create lists from lists
- Each partition is a LinkedList of integers
- Add to a partition p with Link(___, p), putting an integer in the blank

Partitions as Linked Lists

partitions(n,m): list ways to make n by summing positive pieces up to m in increasing order

Summing to n using pieces up to size m involves either:

- (using an m, then) summing to n-m using pieces up to size m, or
- (not using m, then) summing to n using pieces up to size m-1

```
def partitions(n: int, m: int) -> list[LinkedList[int]]:
    """Return a list of partitions of n using parts of up to m.
    Each partition is represented by a linked list.
    """
    if n == 0:
        return [()] # A list containing just the empty partition
    elif n < 0 or m == 0:
        return [] # An empty list with no partitions at all
    else:
        sums_to_n = [Link(m, p) for p in partitions(n-m, m)]
        without_m = partitions(n, m-1)
        return sums_to_n + without_m # Adding lists creates a longer list of all items in both
```

Useful concepts

- **partitions** returns a list
- List comprehensions create lists from lists
- Each partition is a LinkedList of integers
- Add to a partition p with Link(___, p), putting an integer in the blank