DISCUSSION 04

Tree Recursion, Python Lists

Mingxiao Wei
mingxiaowei@berkeley.edu

Sep 22, 2022

LOGISTICS 

- Homework 03 due today Thu 09/22
  - There's an optional contest in the homework! - Not worth any credit, but if you want to have fun with higher order functions, go for it 😼
- CATS is released 😽😽😽
  - Try out the game here: cats.cs61a.org (not now lol)
  - Checkpoint 1 due next Tue 09/27
  - The whole project due next Fri 09/30
  - Submit everything by next Thu 09/29 for one extra credit

TREE RECURSION 

TREE RECURSION

- A tree recursive function is a recursive function that makes more than one call to itself, resulting in a tree-like series of calls.
- For example, let's say we want to recursively calculate the n^{th} Fibonacci number, defined as:

```
def fib(n):
    if n == 0 or n == 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

- Now, what happens when we call `fib(4)`?
  - Each `fib(i)` node represents a recursive call to `fib`.
  - For `i >= 2`, each recursive call `fib(i)` makes another two recursive calls, which are to `fib(i - 1)` and `fib(i - 2)`.
  - Whenever we reach a `fib(0)` or `fib(1)` node, we can directly return 0 or 1, since these are our base cases.

COUNT PARTITIONS REVISIT

Given two positive integers `n` and `m`, return the number of ways in which `n` can be expressed as the sum of positive integer parts up to `m` in increasing order.

- Recursive case: Since each integer part is up to `m`, at each step, where each step generates one number in the partition, we have two choices:
    1. Use `m` to partition `n`, so that at the next step, `n` becomes `n - m`, and the largest possible part is still `m`
    2. Don't use `m`. So at the next step, `n` remains unchanged, but `m` becomes `m - 1` (we choose not to use the largest possible part, `m`, so the next largest possible one is `m - 1`)
- The two choices will result in two distinct sets of results, since in the first one we use `m` to partition, while in the second one we use at most `m - 1`
- Therefore, the total # of partitions = # partitions from choice 1 + # partitions from choice 2

COUNT PARTITIONS REVISIT

Given two positive integers `n` and `m`, return the number of ways in which `n` can be expressed as the sum of positive integer parts up to `m` in increasing order.

- Base case:
  - `n == 0` - note that since `n` and `m` are positive integers according to the problem description, when `n` is 0, it could only be the case where `n - m` results in 0 from the previous recursive call. In other words, when `n` is 0, it menas that we've successfully partitioned `n` so that there's nothing left to partition. In this case, return 1, since we found one valid parition.
  - `n < 0` - similarly, since the original input to the function must be positive integers, a negative `n` can only result from `n - m` from the last step. In this case, `m` was greater than `n` from the last step, indicating that the partition was not succeessful.
  - `m == 0` - also similarly, a negative `m` can only result from `m - 1` from the last step. Since the question requires that all parts of a partition are positive integers, such an partition is invalid.

COUNT PARTITIONS REVISIT

Given two positive integers `n` and `m`, return the number of ways in which `n` can be expressed as the sum of positive integer parts up to `m` in increasing order.

```
def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0 or m == 0:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```

WORKSHEET Q1, 2

PYTHON LISTS

[ , ]

PYTHON LISTS - INTRO

- In Python, a list is a data structure that can store multiple elements in a defined order.
- Each element can be of any type, even a list itself.
- Lists are created by wrapping square brackets around comma-separated expressions
- `len(<seq>)` is a built-in function that takes in a sequence and returns the length (i.e., the number of elements in the sequence)

```
>>> a = ['a', 'b']
>>> b = [1, 2]
>>> len(a)
2
>>> c = [a, b, [3, 4], 5]
>>> c
[['a', 'b'], [1, 2], [3, 4], 5]
>>> len(c)
4
```

PYTHON LISTS - INDEXING

- Each element has its corresponding index, starting from 0 (i.e., "zero-indexed").
  - The index of the last element is `len(lst)-1`
- When index is negative, it means starting from the end of the list
  - `-i` is equivalent to `len(lst)-i`
  - `lst[-1]` is the last element, `lst[-2]` is the second to last element, etc.

```
>>> a = ['a', 'b']
>>> b = [1, 2]
>>> a[-1]
'b'
>>> c = [a, b, [3, 4], 5]
>>> c
[['a', 'b'], [1, 2], [3, 4], 5]
>>> c[1][1]
1
```

PYTHON LISTS - OTHER OPERATIONS

- Concatenation - use `+` to concatenate multiple lists together

```
>>> a = ['say']
>>> b = ['cheese', '!']
>>> a + b # same as add(a, b)
['say', 'cheese', '!']
```

- Repetition - use `*` to duplicate a list certain times

```
>>> a = [6, 1]
>>> a * 3 # same as a + a + a, or mul(a, 3)
[6, 1, 6, 1, 6, 1]
>>> add(mul(a, 2), mul(['a'], 2)) # a * 2 + ['a'] * 2
[6, 1, 6, 1, 'a', 'a']
```

PYTHON LISTS - OTHER OPERATIONS

- Checking element existence - use `in` to check whether or not some value is contained in the list
- To check for non-existence, use `elem not in lst` or `not (elem in lst)`

```
>>> a = ['o', 'p']
>>> 'p' in a
True
>>> 'oops' in a
False
>>> 'op' not in a
True
>>> not ('op' in a)
True
```

LIST SLICING

- List slicing creates a copy of part or all of the list.

```
lst[ <start index> : <end index> : <step size> ]
```

- `start index`
  - index to start at, *inclusive*, default to 0*
- `end index`
  - index to end by, *exclusive*, default to `len(lst)`*
  - when negative, counts from the end of the list, similar to negative indexing
- `step size`
  - the difference between indices of elements to include , default to 1
  - negative steps means stepping backwards

* when step is positive (when step is negative, start index defaults to the end of the list and end index defaults to the start of the list)

LIST SLICING

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst[1: ]
[2, 3, 4, 5]
>>> lst[:-2]
[1, 2, 3]
>>> lst[1::2]
[2, 4]
>>> lst[::-1] # reverse the list
[5, 4, 3, 2, 1]
>>> lst[5:9] # list slicing won't cause an index error
[]
```

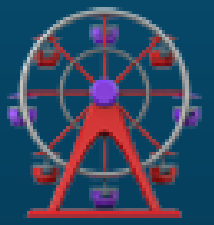
Takeaway: list slicing picks elements at indices `start`, `start + step`, `start + 2 * step`, ... and stops before `end`, and makes those selected elements into a new list

LIST COMPREHENSION

- List comprehensions are a compact and powerful way of creating new lists out of sequences.

```
[<expr> for <var> in <seq> if <cond>]
```

- In English, this translates to:
  1. For each element in the sequence `<seq>`, bind it to the variable name `<var>`.
  2. If the element satisfy the condition `<cond>` (or skip this check if there's no condition), evaluate the expression `<expr>`, and add the value from `<expr>` to the resulting list.
- Note:
  - `if <cond>` is optional.
  - `<expr>` and `<cond>` may refer to `<var>`, which is essentially every element in the sequence

RANGES AND FOR LOOPS 

RANGE

```
range(start, end, step)
```

- represents a sequence of integers
- can be converted to a list by calling the list constructor `list()`
- `start`: optional, inclusive, default to 0
- `end`: required, exclusive
- `step`: optional, default to 1

```
>>> list(range(4))
[0, 1, 2, 3]
>>> list(range(1, 6))
[1, 2, 3, 4, 5]
>>> list(range(2, 9, 2))
[2, 4, 6, 8]
```

FOR LOOPS

- For loops allow us to iterate over some sequences conveniently

```
for <name> in <sequence>:
    <suite>
```

- To iterate over the elements in a seqeunce, we can either iterate over the sequence directly or iterate through its indices and index into the sequence each time.

```
# example: print all elements in lst
for elem in lst:
    print(elem)
# this is equivalent to
for i in range(len(lst)):
    print(lst[i])
```

WORKSHEET Q3,4,5

DICTIONARIES 

DICTIONARIES

- Dictionaries maps keys to their corresponding values
- Create a dictionary
  - `{key1: val1, key2: val2, key3: val3}`
- Select from a dictionary
  - `dict[key]`
  - If `key` does not exist in `dict`, this will error
- Modify a dictionary:
  - `dict[key] = val`
  - If `key` does not exist in `dict`, this will create a new entry. Otherwise it updates the value corresponding to `key` to be `val`
- `len(dict)` returns the number of entries (key-value pairs) in the dictionary
- The key of a dictionary must be immutable (numbers, strings, tuples, but NOT lists)

WORKSHEET Q6

ATTENDANCE! 

go.cs61a.org/mingxiao-att

- The attendance form and slides are both linked on our section website!
- Once again, please do remember to fill out the form by midnight today!!

CLARIFICATION ON Q1

Solution #1:

```
def count_stair_ways(n):
    if n == 1:
        return 1
    elif n == 2:
        return 2
    return count_stair_ways(n-1) + count_stair_ways(n-2)
```

Solution #2:

```
def count_stair_ways(n):
    if n < 3:
        return n
    return count_stair_ways(n-1) + count_stair_ways(n-2)
```

CLARIFICATION ON Q1

Solution #3:

```
def count_stair_ways(n):
    if n == 0:
        return 1
    elif n < 0:
        return 1
    return count_stair_ways(n-1) + count_stair_ways(n-2)
```

- In solution #1, we consider the case when there's 1 or 2 stairs left to go, and the result is 1 and 2, respectively. When there's one stair left, we can only take one step to get to the top. Whereas when there's two stairs left, we can either take one step twice or two stpes once.
- Solution #2 is a shorter version of solution #1, with the same ideas.
- In solution #3, we consider the case where we've finished (`n == 0`) or found a unsuccessful combination of steps (`n < 0`).

CLARIFICATION ON Q1

INCORRECT ANSWERS

```
def count_stair_ways(n):
    if n == 1:
        return 1
    elif n < 0:
        return 0
    return count_stair_ways(n-1) + count_stair_ways(n-2)
```

- In this case, we miss the base case where `n == 0`

To sum up, once `n` goes past 0, we'll need a base case where `n == 0`, since `n` could reach 0 directly without reaching 1.