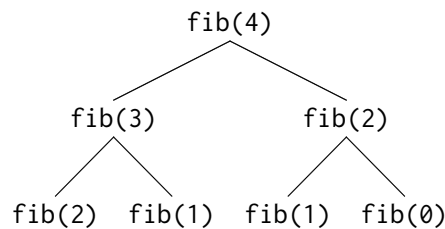# 1 Tree Recursion

Consider a function that requires more than one recursive call. A simple example is the recursive `fibonacci` function:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

This type of recursion is called tree recursion, because it makes more than one recursive call in its recursive case. If we draw out the recursive calls, we see the recursive calls in the shape of an upside-down tree:

```
                    fib(4)

          fib(3)              fib(2)

      fib(2)  fib(1)      fib(1)   fib(0)
```

We could, in theory, use loops to write the same procedure. However, problems that are naturally solved using tree recursive procedures are generally difficult to write iteratively. It is sometimes the case that a tree recursive problem also involves iteration: for example, you might use a while loop to add together multiple recursive calls.

As a general rule of thumb, whenever you need to try multiple possibilities at the same time, you should consider using tree recursion.

Questions

1.1   You want to go up a flight of stairs that has n steps. You can either take 1 or 2 steps each time. How many different ways can you go up this flight of stairs? Write a function `count_stair_ways` that solves this problem. Assume n is positive.

```
When there is only 1 step, there is only one way.
When there are 2 steps, there are two ways to go up the stairs:
take 1 step two times, or take 2 steps in one go.
```

Before we start, what's the base case for this question? What is the simplest input?

```
1. n == 1
2. n == 2
```

What do `count_stair_ways(n - 1)` and `count_stair_ways(n - 2)` represent?

```
The number of different ways to get to the stair (n - 1) which is
right below the current one.
The number of different ways to get to the stair (n - 2).
```

Fill in the code for count_stair_ways:

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

1.2 **Tutorial:** Consider a special version of the `count_stairways` problem, where instead of taking 1 or 2 steps, we are able to take **up to and including k** steps at a time.

Write a function `count_k` that figures out the number of paths for this scenario. Assume n and k are positive.

```python
def count_k(n, k):
    """
    >>> count_k(3, 3) # 3, 2 + 1, 1 + 2, 1 + 1 + 1
    4
    >>> count_k(4, 4)
    8
    >>> count_k(10, 3)
    274
    >>> count_k(300, 1) # Only one step at a time
    1
    """
    if k == 1:
        return 1
    if n == 1 or n == 0:
        return 1
    if n == 2:
        return 2

    i = 1
    total = 0
    while i <= k and n - i >= 0:
        total += count_k(n - i, k)
        i += 1
    return total
```

2   Lists

A *sequence* is an ordered collection of values. It has two fundamental properties: length and element selection. In this discussion, we'll explore one of Python's data types, the *list*, which implements this abstraction.

In Python, we can have lists of whatever values we want, be it numbers, strings, functions, or even other lists! Furthermore, the types of the list's contents need not be the same. In other words, the list need not be homogenous.

Lists can be created using square braces. Their elements can be accessed (or *indexed*) with square braces. Lists are zero-indexed: to access the first element, we must index at 0; to access the ith element, we must index at $i - 1$.

We can also index with negative numbers. These begin indexing at the end of the list, so the index -1 is equivalent to the index `len(list) - 1` and index -2 is the same as `len(list) - 2`.

Let's try out some indexing:

```
>>> fantasy_team = ['aaron rodgers', 'desean jackson']
>>> print(fantasy_team)
['aaron rodgers', 'desean jackson']
>>> fantasy_team[0]
'aaron rodgers'
>>> fantasy_team[len(fantasy_team) - 1]
'desean jackson'
>>> fantasy_team[-1]
'desean jackson'
```

List Slicing

If we want to access more than one element of a list at a time, we can use a *slice*. Slicing a sequence is very similar to indexing. We specify a starting index and an ending index, separated by a colon. Python creates a new list with the elements from the starting index up to (but not including) the ending index.

We can also specify a step size, which tells Python how to collect values for us. For example, if we set step size to 2, the returned list will include every **other** value, from the starting index until the ending index. A negative step size indicates that we are stepping backwards through a list when collecting values.

You can also choose not to specify any/all of the slice arguments. Python will perform some default behaviour if this is the case:

- If the step size is left out, the default step size is 1.

- If the start index is left out, the default start index is the beginning of the list.

- If the end index is left out, the default end index is the end of the list.

- If the step size is negative, the default start index becomes the end of the list, and the default end index becomes the beginning of the list.

Thus, `lst[:]` creates a list that is identical to `lst` (a copy of `lst`). `lst[::-1]` creates a list that has the same elements of `lst`, but reversed. Those rules still apply if more than just the step size is specified e.g. `lst[3::-1]`.

```
>>> directors = ['jenkins', 'spielberg', 'bigelow', 'kubrick']
>>> directors[:2]
['jenkins', 'spielberg']
>>> directors[1:3]
['spielberg', 'bigelow']
>>> directors[1:]
['spielberg', 'bigelow', 'kubrick']
>>> directors[0:4:2]
['jenkins', 'bigelow']
>>> directors[::-1]
['kubrick', 'bigelow', 'spielberg', 'jenkins']
```

List Comprehensions

A list comprehension is a compact way to create a list whose elements are the results of applying a fixed expression to elements in another sequence.

```
[<map exp> for <name> in <iter exp> if <filter exp>]
```

It might be helpful to note that we can rewrite a list comprehension as an equivalent for statement. See the example to the right.

Let's break down an example:

```
[x * x - 3 for x in [1, 2, 3, 4, 5] if x % 2 == 1]
```

In this list comprehension, we are creating a new list after performing a series of operations to our initial sequence `[1, 2, 3, 4, 5]`. We only keep the elements that satisfy the filter expression `x % 2 == 1` (1, 3, and 5). For each retained element, we apply the map expression `x*x - 3` before adding it to the new list that we are creating, resulting in the output `[-2, 6, 22]`.

Note: The `if` clause in a list comprehension is optional.

Questions

2.1   What would Python display?

```
>>> a = [1, 5, 4, [2, 3], 3]
>>> print(a[0], a[-1])
    1 3

>>> len(a)
    5

>>> 2 in a
    False

>>> 4 in a
    True

>>> a[3][0]
    2
```

2.2   **Tutorial:**   Write a function that takes a list s and returns a new list that keeps only the even-indexed elements of s and multiplies them by their corresponding index.

```
def even_weighted(s):
    """
    >>> x = [1, 2, 3, 4, 5, 6]
    >>> even_weighted(x)
    [0, 6, 20]
    """

    return [_i * s[i] for i in range(len(s)) if i % 2 == 0_]
```

2.3 Write a function that takes in a list and returns the maximum product that can be formed using nonconsecutive elements of the list. The input list will contain only numbers greater than or equal to 1.

```python
def max_product(s):
    """Return the maximum product that can be formed using non-consecutive
    elements of s.

    >>> max_product([10,3,1,9,2]) # 10 * 9
    90
    >>> max_product([5,10,5,10,5]) # 5 * 5 * 5
    125
    >>> max_product([])
    1
    """
    if len(s) == 0:
        return 1
    if len(s) == 1:
        return s[0]
    return max(max_product(s[1:]), s[0] * max_product(s[2:]))
```