Making change, tiling, and Fibonacci's sequence in combinatorics

Van Thuan Romoli

Semester 1, 2022/2023

1 Introduction

In this project we explore how different problems in the real-world can be represented by the same combinatorial problem. In particular, we will look at three problems and how they are related to each other: Making change, tiling, and Fibonacci's sequence.

2 Making Change

In this section, we define functions that output the number of ways of making change, given some specified coins. If the answer is non-zero, then we also give a list of solution tuples (a solution tuple represents a possible way of making change).

The simplest possible option would be writing a function `Pence(n)` that receives an integer n and returns the number of ways to make n pence in change when we only have the 1p coin available. If this value is non-zero, then the second value is a list of solution tuples.

```python
def Pence(n):

    itemsintuple = []

    if n < 0:
        return(0, None)
    else:
        for x in range (0,n):
            itemsintuple.append(1)
        return(1, tuple(itemsintuple))
```

Now, we want to explore how the function would change when we consider the 5p coin. We define the function `FiveP(n)` that receives an integer n and returns the number of ways of ways to make n pence in change when we only have the 5p coin available. If this value is non-zero, then the second value is a list of solution tuples, where the 5's occur at the start of the tuples. We make two remarks: (1) the order of the solutions within the list does not matter; and (2) we can only make n pence in change if n is a multiple of 5 greater than 0. The code below (and, in general, this project) leverages the use of modular arithmetic, floor division, and reminders to determine when n is a multiple of 5.

```python
def FiveP(n):

    itemsintuple = []

    if n % 5 == 0 and n > 0:
        for x in range (0,n//5):
            itemsintuple.append(5)
        return(1, tuple(itemsintuple))
    else:
        return(0, None)
```

We investigate how we can combine the two above answer to make n pence in change when we have both the 5p and 1p coins available. We write a function `ChangeFivePandP(n)` that receives an integer n and returns the number of ways to make change with both 5p and 1p coins available. The function consists of computing the number of ways of making k from 5's and $n-k$ from 1's as $0 \leq k \leq n$.

```python
def ChangeFivePandP(n):
    floordiv = n // 5
    if n < 0:
        return(0)
    else:
        return(floordiv + 1)
```

As done previously, we redefine `ChangeFivePandP(n)` to have a double output: the first value is the answer computed above. If the answer is not zero, then the second value is a list of solution tuples, where the 5's occur at the start of the tuples. The order of the solutions within the list does not matter.

```python
def ChangeFivePandP(n):

    listoftuples = []

    floordiv = n // 5
    reminder = n % 5
    waysofchoosing = floordiv + 1

    if n < 0:
        return(0, listoftuples)

    else:
        counter = 0
        while counter < waysofchoosing:
            temptuple = []

            if counter == 0:
                for x in range(0,n):
                    temptuple.append(1)
                listoftuples.append(tuple(temptuple))
                counter += 1
                #print(f"at counter {counter -1 } we have tuples {listoftuples}")
            else:
                for x in range(0, counter):
                    temptuple.append(5)
                for x in range(n - 5*counter):
                    temptuple.append(1)

                listoftuples.append(tuple(temptuple))
                counter += 1
                #print(f"at counter {counter -1 } we have tuples {listoftuples}")



    return(waysofchoosing,listoftuples)
```

Finally, we wish to define a function `ChangeTenPFivePandP(n)`, which returns the number of ways of making n pence when we have the 10p, 5p, and 1p coins available. The approach we used above using while loops and counters is cumbersome and not as easy to implement when having three coins. Therefore we switch to a different algorithm.

```python
def ChangeTenPFivePandP(n):
    count = 0
    for x in range(n // 10 + 1):  # x = number of 10p coins
        for y in range((n - 10 * x) // 5 + 1):  # y = number of 5p coins
            z = n - 10 * x - 5 * y  # z = number of 1p coins
            if z >= 0:
                count += 1
    return count
```

This code is more clear and concise: the outer loop goes through all possible numbers of 10p coins (`x`). The inner loop goes through all possible numbers of 5p coins (`y`), given the remaining amount after using `x` 10p coins. For each valid combination of `x` and `y`, `z` is determined uniquely and must be non-negative. Every valid (`x`, `y`, `z`) counts as one way.

3 Tiling

In this part, we leverage the “change sequences” output above to solve a tiling question. We will explore how to tile a walkway which is n feet long and two feet tall by 2×1 (ft^2) shaped tiles, laid either horizontally or vertically on the walkway with no overlaps, and with all parts of the the walkway being covered by some tile.

Below are some example tilings of a 5×2 walkway, where we use “`|`” for a vertical tile, and “ `=` ” for two tiles laid horizontally atop each other.

```
1. |||||
2. |||=
3. ||=|
```

Note that in a $n \times 2$ walkway, two horizontal tiles that lie one-above-the-other must be perfectly aligned because of the way the tiles are shaped and the constraints of the tiling. Each tile is 2×1 feet in size, meaning a horizontal tile covers two adjacent columns in a single row. To fully and exactly cover a 2×2 block of the walkway with horizontal tiles, one in the top row and one in the bottom, they must span the same two columns. If they are not aligned (e.g., one covers columns 1–2 and the other covers 2–3), then some parts of the walkway would be uncovered, and others would be covered more than once, violating the tiling rules. Therefore, perfect alignment is required to ensure the walkway is completely and precisely covered without gaps or overlaps.

A solution tuple to making n pence change using 1p and 2p describes a possible layout for the tiling problem of a $n \times 2$ walkway. A 2p coin maps to a horizontal tile pair, and a 1p coin maps to a single vertical tile. Different permutations correspond to distinct valid tilings of the walkway. Thus, by mapping making change sequences to tile placements, we can enumerate all possible tilings of the $n \times 2$ walkway.

Below is the code for the function `ChangeTwoPandP(n)` which returns the number of ways and list of solution tuples to making n pence when we have 2p and 1p coins available.

```python
def ChangeTwoPandP(n):

    listoftuples = []

    floordiv = n // 2
    reminder = n % 2
    waysofchoosing = floordiv + 1

    if n < 0:
        return(0, listoftuples)

    else:
        counter = 0
        while counter < waysofchoosing:
            temptuple = []

            if counter == 0:
                for x in range(0,n):
                    temptuple.append(1)
                listoftuples.append(tuple(temptuple))
                counter += 1
                #print(f"at counter {counter -1 } we have tuples {temptuple}")
            else:
                for x in range(0, counter):
                    temptuple.append(2)
                for x in range(n - 2*counter):
                    temptuple.append(1)

                listoftuples.append(tuple(temptuple))
                counter += 1
                #print(f"at counter {counter -1 } we have tuples {temptuple}")



    return(waysofchoosing,listoftuples)
```

We then use the library function `itertools.combinations` to write a function `TilePermutations(seq)` which takes a change sequence of 2s and 1s as in `ChangeTwoPandP(n)`, and returns a list containing all of the distinct permutations of that sequence in some order. For example, `TilePermutations((2, 1, 1, 1))` might return `[(2,1,1,1), (1,2,1,1), (1,1,2,1), (1,1,1,2)]` (though any other order of these 4-tuples is also fine).

```python
import itertools
def TilePermutations(seq):

    seq_list = list(seq)
    seq_size = len(seq_list)
    n_of_twos = 0

    #the code below is to determine the positions where 2s are located

    pos_for_twos = []

    for i in range(0,seq_size):
        if seq_list[i] == 2:
            n_of_twos += 1

    for i in range(0, seq_size):
        pos_for_twos.append(i)

    permutations = list(itertools.combinations(pos_for_twos,n_of_twos))

    #the code below is to create the required list

    list_of_permutations =[]

    #here we iterate through all the different possible permutations
    for i in range(0, len(permutations)):
        ith_permutation = []
        while_counter = 0
        twos_counter = 0

        #here we create the ith permutation

        #first, we start by creating an array of ones
        for t in range(0, seq_size):
            ith_permutation.append(1)

        #then we determine wich positions the twos must be places and then replace
        #if the position matches the counter
        for x in range(0, n_of_twos):
            position = permutations[i][x]
            ith_permutation[position] = 2
        list_of_permutations.append(tuple(ith_permutation))


    return(list_of_permutations)
```

The next function we write is `TilingString(seq)`, which takes an input sequence of ones and twos and returns the corresponding tiling. For example, `TilingString((1,1,2,1,1))` will return `||=||`.

```python
def TilingString(seq):
    seq_list = list(seq)
    seq_length = len(seq_list)
    string = ""
    for i in range(0,seq_length):
        if seq_list[i] == 1:
            string += '|'
        else:
            string += '='
    return(string)
```

We put everything together by defining two functions. We define `PrintTilingsLength(n)` that uses `ChangeTwoPandP(n)` and `TilingString(seq)` to print all distinct tilings for an $n \times 2$ walkway (in some order); and `TilingsCount(n)` that uses `ChangeTwoPandP(n)` and `TilePermutations(n)` to count all valid tilings for an $n \times 2$ walkway.

```
def PrintTilingsLength(n):
    changetwopandp = ChangeTwoPandP(n)
    for x in changetwopandp[1]:
        print(TilingString(x))

PrintTilingsLength(5)
|||||
=|||
==|
```

```
def TilingsCount(n):
    floordiv = n // 2
    waysofchoosing = floordiv + 1

    changetwopandp = ChangeTwoPandP(n)[1]
    counter = 0
    for i in range(0,waysofchoosing):
        counter += len(TilePermutations(changetwopandp[i]))
    return(counter)

for i in range(8):
    print(TilingsCount(i))

1
1
2
3
5
8
13
21
```

Suspicious of the values above, we verified that we get the correct first 20 Fibonacci numbers as the values of `TilingsCount(n)` for $0 \le n \le 20$.

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

for i in range(0,21):
    if TilingsCount(i) == fib(i):
        print(f"TilingsCount({i})={TilingsCount(i)} and fib({i})={fib(i)} so
    TilingsCount({i})=fib({i})")
    else:
        print(f"TilingsCount({i})={TilingsCount(i)} and fib({i})={fib(i)} so
    TilingsCount({i})=/=fib({i})")
```

4 Fibonacci's sequence

In this section, we wish to explain why the output of `TilingsCount(n)` corresponds to the n-th Fibonacci number, for all $n \in \mathbb{N}$. We define the Fibonacci's sequence by the recurrence relation $F_1 = 1$, $F_2 = 2$, and $F_n = F_{n-1} + F_{n-2}$ for $n > 2$.

We now make the following argument:

1. Consider `TilingsCount(1)`, clearly "`|`" is the only one way of tiling the path, hence we must have `TilingsCount(1)` $= 1$.
2. Consider `TilingsCount(2)`, clearly "`||`" and " `=` " are the only two ways of tiling the path, hence `TilingsCount(2)` $= 2$.
3. When $n > 2$, we note that the walkway either starts with "`|`" or " `=` ".
    - If the walkway starts with "`|`", then we have a $(n-1) \times 2$ path that can be tiled in `TilingsCount(n-1)` ways.
    - If the walkway starts with " `=` ", then we have a $(n-2) \times 2$ path that can be tiled in `TilingsCount(n-2)` ways.

    Hence, the number of ways of tiling a $n \times 2$ path is `TilingsCount(n-1)` $+$ `TilingsCount(n-2)`.

We may find `TilingsCount(n)` by using the following recurrence relation in Python.

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

The primary issue with naive recursion for calculating the Fibonacci sequence is its inefficiency due to redundant calculations. Each function call to `TilingsCount(n)` results in multiple calls to `TilingsCount(n-1)` and `TilingsCount(n-2)`, leading to exponential growth in computation time as the input n increases.

A major benefit of proving that `TilingsCount(n)` corresponds to the n-th Fibonacci number is that we can leverage the use of existing algorithms to calculate Fibonacci's sequence. For example, we may use the following function.

```python
# Fast Tiling Count
def FTC(n):
    a, b = 1, 2
    for i in range(1,n):
        a, b = b, a+b
    return a
```

5 Conclusion

In this project, we showed how the three seemingly different problems of making change, tiling, and Fibonacci's sequence are related. In particular, we presented how all permutations of each tuple in the list of solution tuples to the making change problem can be used to count all valid tilings for an $n \times 2$ walkway. Then, we leveraged Fibonacci's sequence to improve on the recursion by using an optimised algorithm.