COL100 Assignment 4

Arpit Saxena
2018MT10742
Group 29

1. Consider the ML function for computing the number of ways a rook can travel from the bottom left corner to the top left corner in a $p \times q$ chess board:

```
fun rook(p,q) =
    if ((p = 1) orelse (q = 1)) then 1
    else rook(p−1,q) + rook(p,q−1);

fun C(m,n) =
    if ((m = n) orelse (n = 0)) then 1
    else C(m−1,n−1) + C(m−1,n);

fun rook1(p,q) = C(p+q−2,p−1);
```

(a) Analyse the correctness of the function.

Solution: Considering the rook function, let $N = p + q - 2$ be the induction variable.

- **To Prove:** $S \equiv$ rook(p, q) computes the number of ways a rook can travel from the bottom left corner to the top left corner in a $p \times q$ chess board; $p \geq 0, q \geq 0$
- **Base case:** $N = 0$
  Since $p \geq 1, q \geq 1; N = 0 \implies p = 1, q = 1$ In a 1×1 board, there's only one way for rook to go from start to end i.e. to stay at the same place.
- **Induction Hypothesis:** $S = S(N)$ is true for all p, q such that $p + q = N$
- **Induction Step:** Consider any p, q such that $p + q = N + 1$
  1. If $q = 1$ or $p = 1$, then rook has only one way to go from start to end. Also, rook(p, q) = 1 if $p = 1$ or $q = 1$
  2. If $p \neq 1$ and $q \neq 1$: Backtracking from rook's final position, (p, q), it could have got there from $(p, q - 1)$ or $(p - 1, q)$ only. By the fundamental principle of counting, the ways to get from (1,1) to (p, q) equal the sum of the ways to get from (1,1) to (p - 1, q) and ways to get from (1, 1) to (p, q - 1). By the induction hypothesis, rook(p, q - 1) and rook(p - 1, q) equal the number of ways for rook to get from (1, 1) to (p, q - 1) and (p - 1, q) respectively, rook(p, q) = rook(p, q - 1) + rook(p - 1, q) equals the number of ways to go from (1, 1) to (p, q)

  $\therefore S(N) \implies S(N + 1)$

$\therefore$The statement S was proven true by the principle of mathematical induction.
Now, we will show that $rook1 \equiv rook$ and it is thus correct.
We have, fun rook1(p,q) = C(p+q−2,p−1);

- **To Prove:** $S \equiv$ C(p + q − 2, p − 1) = rook(p, q) $\forall p, q \geq 1$
  We will prove this by induction on $p + q - 2$
- **Base Case:** $p + q - 2 = 0 \implies p = 1, q = 1 \because p, q \geq 1$
  For $p = q = 1$ C(p + q − 2, p − 1) = C(0, 0) = 1
  rook(p, q) = rook(1, 1) = 1 So, S is true in the base case.
- **Induction hypothesis:** $S = S(N)$ is true for all p, q such that $p + q = N$

- **Induction step:** Consider any p, qsuch that$p + q = N + 1$

  $$\text{Then, } C(p+q-2, p-1) = \begin{cases} 1 \text{ if } p+q-2 = p-1 \text{ or } p-1 = 0 \implies p = q = 1 \\ C((p-1)+q-2, (p-1)-1) + C(p+(q-1)-2, p-1) \end{cases}$$

  Now, by the induction hypothesis:

  $$C((p-1)+q-2, (p-1)-1) + C(p+(q-1)-2, p-1) = rook(p-1, q) + rook(p, q-1)$$

  $$\therefore C(p+q-2, p-1) = \begin{cases} 1 \text{ if } p = 1 \text{ or } q = 1 \\ rook(p-1, q) + rook(p, q-1) \end{cases} = rook(p, q)$$

  $\therefore S(N) \implies S(N+1)$

So, by the principle of mathematical induction, S is true $\forall N \in \mathbb{N}$ That is, $rook1 \equiv rook$ and is thus correct. All the time and space complexity analysis is the same for rook1 and rook. So, we have only done the analysis for rook function. (Note that there is an additional storage requirement in rook1 but since it is constant, the asymptotic space complexity is the same.)

(b) Estimate its time and space complexity.

Solution:

- **Time Complexity:**

  We estimate time complexity by the number of function calls made. So, we have the following recurrence:

  $$T(p, q) = \begin{cases} 0 \text{ if } p = 1 \text{ or } q = 1 \\ 2 + T(p-1, q) + T(p, q-1) \text{ otherwise} \end{cases}$$

  Let $p \geq 1$ then

  $$\begin{aligned} T(p, 1) &= 0 \\ T(p, 2) &= 2 + T(p-1, 2) + \cancelto{0}{T(p, 1)} \\ &= 4 + T(p-2, 2) \\ &= 2k + T(p-k, 2) \\ &= 2(p-1) + \cancelto{0}{T(1, 2)} \\ &= 2(p-1) \\ T(p, 3) &= 2p + T(p-1, 3) \\ &= 2\{p + (p-1) + \cdots + 2\} + \cancelto{0}{T(1, 3)} \\ &= 2\left\{\frac{p(p+1)}{2} - 1\right\} \\ &= 2\,{}^{p+1}C_2 - 2 \end{aligned}$$

  This leads to hypothesise that $T(p, q) = 2\,{}^{p+q-2}C_{q-1} - 2\,\forall p, q \geq 1$

  Assuming that it is true $\forall p, q$ such that $p + q = N$, consider another p, q such that $p + q = N + 1$; then

  $$\begin{aligned} T(p, q) &= 2 + T(p-1, q) + T(p, q-1) \\ &= 2 + 2\,{}^{p+q-3}C_{q-1} - 2 + 2\,{}^{p+q-3}C_{q-2} - 2 \\ &= 2\,{}^{p+q-2}C_{q-1} - 2 \end{aligned}$$

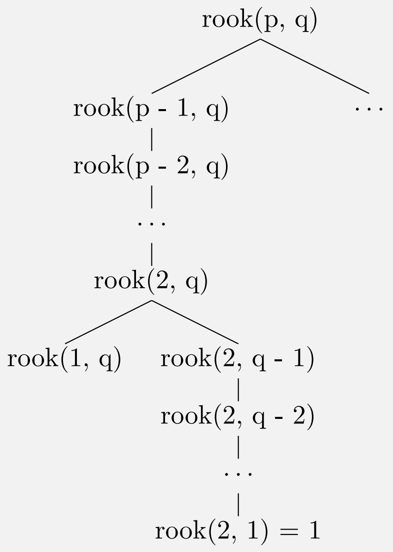
This proves that $T(p, q) = 2\,{}^{p+q-2}C_{q-1} - 2\,\forall p, q \geq 1$

$$\begin{aligned}
\frac{n!}{(n-k)!} &= n(n-1)\ldots(n-k+1) = \mathcal{O}\left(n^k\right) \\
\implies {}^nC_k &= \frac{n!}{k!(n-k)!} \\
&= \mathcal{O}\left(\frac{n^k}{k!}\right) \\
\text{Now, } T(p, q) &=^{p+q-2} C_{q-1} - 2 \\
&\quad \text{We know that binomial coefficients are maximum in the middle} \\
T(p, q) &= \mathcal{O}\left((2p-2)C_{p-1}\right) \\
&= \mathcal{O}\left(\frac{(2p-2)^{p-1}}{(p-1)!}\right) \\
&= \mathcal{O}\left(2^{p-1}\frac{(p-1)^{p-1}}{(p-1)!}\right) \\
&= \mathcal{O}\left(2^p\right)
\end{aligned}$$

So, the time complexity is $\mathcal{O}\left(2^p\right)$ in the worst case.

- **Space Complexity:**

  We are considering the space complexity as the maximum space the program would require at one point of time. Being a recursive algorithm, we observe that extra storage is required when the recursive calls are made and the computations have to deferred. So, we find the length of the longest branch of function calls and the space required for those would give us the space complexity. We observe that following is the longest branch of function calls:



  This branch has a length of $(p-1) + (q-1) = p + q - 2$, and each function call requires 2 units of storage.

$$\begin{aligned}
\text{So, } S(p, q) &= 2 \times (p + q - 2) \\
&= 2p + 2q - 4 \\
&= \mathcal{O}\left(p + q\right)
\end{aligned}$$

  $\therefore$ Space complexity $= \mathcal{O}\left(p + q\right)$

(c) Work out a more efficient iterative version. (Hint: Read about Pascal's triangle).

(d) Develop a Python program for your iterative algorithm.

Solution: Following is the python program for iterative algorithm:

```
def choose(n, m):
    a = [1]
    i = 0

    #INV: For all 0 <= r <= i, a[r] = (i)choose(r); 0 <= i <= n
    while i < n:
        new_a = [0] * (i + 2)
        j = 1
        new_a[0] = 1
        new_a[i + 1] = 1

        #INV: for k in [0..j-1] new_a[k] = (i + 1)choose(k); 1<=j<=i+1
        while(j <= i):
            new_a[j] = a[j] + a[j - 1]
            #(i + 1)Cj = iCj + iC(j - 1)
            j += 1

        #assert: for k in [0..i+1] new_a[k] = (i + 1)choose(k)
        i += 1

        a = new_a

    #assert: For all 0 <= r <= n, a[r] = (n)choose(r)
    return a[m]

def rook(p, q):
    #To find: (p + q - 2) C (q - 1)
    return choose(p + q - 2, q - 1)
```

Proof of correctness is encoded in the invariants and assertions written as comments in the code. We now find the algorithms time and space complexities:

- **Time Complexity:** Since the main work is done in the choose function, we first evaluate its time complexity. For a particular i, the inner while loop runs i times. There are 2 statements in this inner while loop. So, there are $6 + 2i$ statements executed for a particular i in the outer while loop. Assuming each statement to take same, constant time.

$$\begin{aligned} T'(m,n) &= \sum_{i=0}^{n} 6 + 2i \\ &= 6(n+1) + 2n(n+1) \\ &= 2(n+1)(n+3) \\ &= \mathcal{O}\left(n^2\right) \end{aligned}$$

  Since rook(p, q) = choose(p + q − 2, q − 1),
  $T(p,q) = T'(p+q-2, q-1) = \mathcal{O}\left((p+q-2)^2\right) = \mathcal{O}\left((p+q)^2\right)$

- **Space Complexity:** It is easily observable that the maximum storage required by the program at any point of time will be due to array a or new_a, which would clearly be $\mathcal{O}(n)$. So, $S'(m,n) = \mathcal{O}(n) \implies S(p,q) = S'(p+q-2, q-1) = \mathcal{O}(p+q-2) = \mathcal{O}(p+q)$

We observe that for the same asymptotic space complexity, this iterative algorithm has a quadratic time complexity while the recursive one has exponential, implying that this algorithm is quite a bit more efficient than the recursive one.

2. Suppose you have an infinite supply of coins of denomination 50p, 25p, 10p, 5p and 1p. In how many ways can you generate change for a given amount, say for 100p?

   (a) Consider this ML function for the problem and analyse its correctness and efficiency.

```
exception OnlyFiveTypesOfCoins;
fun denom n =
    if n = 1 then 1
    else if n = 2 then 5
    else if n = 3 then 10
    else if n = 4 then 25
    else if n = 5 then 50
    else raise OnlyFiveTypesOfCoins;

fun coin(m,n) =
    if (m < 0) orelse (n = 0) then 0
    else if (m=0) then 1
    else coin(m,n-1)+coin(m-denom(n),n);
```

Solution:

- **Correctness:**
  Considering the coin function, let $N = m + n$ be the induction variable.
  - **To Prove:** $S \equiv$ coin(m, n) computes the number of ways to generate change for an amount m using n coins, which are given by $denom(1), denom(2), \ldots, denom(n)$ if $n > 0$; $m, n \geq 0$
  - **Base case:** $N = 0$
    Since $n \geq 0, m \geq 0, m + n = 0 \implies m = n = 0$ There is no way to generate change for amount 0 using 0 coins, since we don't have any coins left. We also see that rook(0, 0) = 0. $\therefore S$ holds for the base case
  - **Induction hypothesis:** $S = S(N)$ is true for all $m, n \geq 0$ for which $m + n \leq N$
  - **Induction step:** Consider any $m, n \geq 0$ such that $m + n = N + 1$
    * $m \leq 0$: There is no way to generate a negative amount using coins of positive denominations. Also, rook(m, n) = 0 if $m < 0$
    * $n = 0$: We can't generate any change without having any coins.
      Also, rook(m, 0) = 0.
    * $n \neq 0$ and $m = 0$: In this case, we have a positive number of denominations to choose from and we have to generate 0 amount of change. This can be done by choosing 0 coins of all denominations available to us, i.e. in one way. We also see that rook(0, n) = 1 if $n \neq 0$
    * $n > 0$ and $m > 0$: We have to generate change for a non-zero amount using some positive number of denominations available to use. At this step, we can either choose to use a coin of the n^{th} denomination or to not use a coin of that denomination. By the law of excluded middle, there can only be these two possibilities.
      · In the first case, the amount decreases by $d(n)$ and the number of denominations available remain the same. So, the number of ways to generate change in this case is equal to ways for generating change for amount $m - d(n)$ with n coins.

We observe that $(m-d(n))+n <= m+n-1 (\because d(n) >= 1) \implies (m-d(n))+n <= N$, for which the number of ways is given by rook(m − d(n), n) (by the induction hypothesis)
$\therefore$ Number of ways in this case = rook(m − d(n), n)

· In the second case, the amount remains the same and the number of denominations available for use decreases by one, i.e. they become $n-1$. We observe that $m+(n-1) = N \leq N$, for which the number of ways is given by rook(m, n − 1) (by the induction hypothesis)
$\therefore$ Number of ways in this case = rook(m, n − 1)

Since these are cases, by the fundamental law of counting, total number of ways = rook(m - d(n), n) + rook(m, n - 1). We observe that the program also returns the same thing.

$\therefore S(N) \implies S(N+1)$

By the principle of mathematical induction, S is true $\forall m, n \geq 0$
This proves the correctness of the algorithm.

- **Efficiency:**
  - **Time comlpexity:** We estimate time complexity by the number of function calls made. So, we have the following recurrence:

$$T(m,n) = \begin{cases} 0 \text{ if } m \leq 0 \text{ or } n = 0 \\ 2 + T(m, n-1) + T(m - d(n), n) \text{ otherwise} \end{cases}$$

Since the function calls would stop when $m \leq 0$ which decreases by d(n) each time, we observe that the worst case (maximum number of function calls) would be when all the denominations available would be 1. So, the worst case would be when $d(n) = 1 \forall n \geq 1$. Assuming this, our recurrence simplifies as:

$$T(m,n) = \begin{cases} 0 \text{ if } m \leq 0 \text{ or } n = 0 \\ 2 + T(m, n-1) + T(m-1, n) \text{ otherwise} \end{cases}$$

Let $m \geq 0$, then:

$$\begin{aligned} T(m,0) &= 0 \\ T(m,1) &= 2 + \cancelto{0}{T(m,0)} + T(m-1,1) \\ &= 2 + T(m-1,1) \\ &= 4 + T(m-2,1) \\ &= 2m + \cancelto{0}{T(m-m,1)} \\ &= 2m \\ &= \mathcal{O}(m) \\ T(m,2) &= 2 + T(m,1) + T(m-1,2) \\ &= 2 + 2m + T(m-1,2) \\ &= 2(m+1) + T(m-1,2) \\ &= 2(m+1) + 2m + T(m-2,2) \\ &= 2\{(m+1) + m + \ldots + 2\} + \cancelto{0}{T(0,2)} \\ &= \frac{(m+1)(m+2)}{2} - 2 \end{aligned}$$

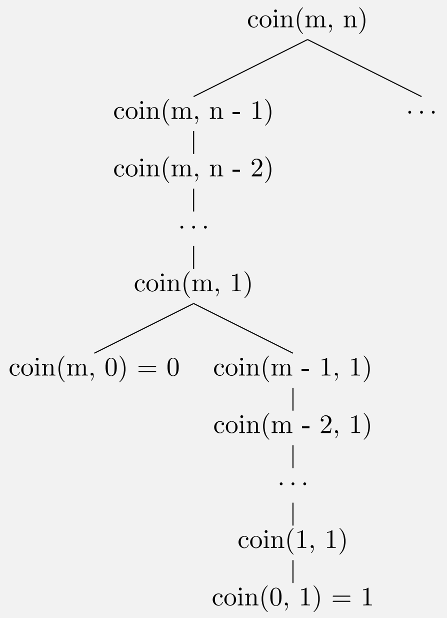
$$\begin{aligned}
&= \mathcal{O}\left(m^2\right) \\
T(m,3) &= 2 + T(m,2) + T(m-1,3) \\
&= 2 + \mathcal{O}\left(m^2\right) + T(m-1,3) \\
&= \mathcal{O}\left(m^2\right) + T(m-1,3) \\
\implies T(m,3) &= \mathcal{O}\left(m^3\right)
\end{aligned}$$

This leads us to hypothesise that $T(m,n) = \mathcal{O}(m^n)$. Let it be true for all m, n such that $m + n = N$. Consider any m, n for which $m + n = N + 1$.

$$\begin{aligned}
T(m,n) &= 2 + T(m,n-1) + T(m-1,n) \\
&= 2 + \mathcal{O}\left(m^{n-1}\right) + \mathcal{O}\left((m-1)^n\right) \text{(By Induction hypothsis)} \\
&= \mathcal{O}(m^n)
\end{aligned}$$

This proves that $T(m,n) = \mathcal{O}(m^n) \, \forall m, n \geq 0$
$\therefore$ Time complexity $= \mathcal{O}(m^n)$

- **Space complexity:**
We are considering the space complexity as the maximum space the program would require at one point of time. Being a recursive algorithm, we observe that extra storage is required when the recursive calls are made and the computations have to deferred. So, we find the length of the longest branch of function calls and the space required for those would give us the space complexity. We observe that following is the longest branch of function calls:



This branch has a length of m + n, and each function call requires 2 units of storage.

$$\begin{aligned}
S(m,n) &= 2(m+n) \\
&= \mathcal{O}(m+n)
\end{aligned}$$

$\therefore$ Space complexity $= \mathcal{O}(m+n)$

(b) Show that a suitably defined iterative function will be more efficient for the given problem.

(c) Develop a Python program for your iterative algorithm.

Solution: Following is the python program for iterative algorithm:

```
#Finds number of ways to give a change for 'total' amount using
#denominations given in 'denom' list
def coin(total, denom):
    #assert: total >= 0 and denom is an array with positive
    #integer values
    n = len(denom)

    ways = [0] * (total + 1)
    j = 0
    ways[0] = 1
    #INV: for i in [0..total] ways[i] is the number of ways to
    #generate change i using j types of coins, 0 <= j <= n
    while j < n:
        k = 0
        #INV: for i in [0..k - 1], ways[i] is the number of ways
        # to generate
        #change i using j + 1 types of coins; 0 <= k <= total + 1
        while k <= total:
            if k >= denom[j]:
                ways[k] = ways[k - denom[j]] + ways[k]
            k = k + 1

        #assert: for i in [0..total], ways[i] is the number of ways
        # to generate
        #change i using j + 1 types of coins
        j = j + 1

    #assert: for i in [0..total] ways[i] is the number of ways to
    # generate change i
    return ways[total]
```

We will now show that this iterative version is more efficient than the given recursive version. We will denote total as m and len(denom) as n for the sake of consistency between the iterative and recursive versions.

- **Time Complexity:** The inner while loop runs a total of $m + 1$ times and in the worst case, 3 statments would be executed each time. So, there are $2 + 3m$ statements run in the outer while loop. That loop runs n times. So the total statements run is $n \times (2 + 3m) = 2n + 3mn = \mathcal{O}(mn)$.
  $\therefore$ Time complexity $= \mathcal{O}(mn)$
- **Space complexity:** It can be easily observed that the variable space is required by the ways list, making the space complexity $O(n)$

Here, we compare the time and space complexities of the two versions side by side:

	Recursive	**Iterative**
Time complexity	$\mathcal{O}(m^n)$	$\mathcal{O}(mn)$
Space comlpexity	$\mathcal{O}(m + n)$	$\mathcal{O}(n)$

This shows that the iterative version is more efficient than the iterative version