IX1500 HT24 Project 1

Rasmus Norberg Thörn & Adalet Adiljan

September 2024

Combinatorics and Set Theory

rasmusnt@kth.se
adalat@kth.se

1 Introduction

In this project assignment, we'll be focusing on analyzing discrete mathematical paths in a grid setting, where a particle moves in a specified up and right "U" or down and right "L" transition, also known as a lattice path problem. More closely, it involves investigating various constraints for the movement of the particle from one point to another. We'll be exploring both basic and more advanced combinatorial problems, applying techniques like the ballot theorem and binomial coefficients. The goal is to compute, explain and visualize paths that satisfy certain conditions in a given constraint, such as never crossing the x-axis or passing through specific points. As for the last assignment, we'll handle a classic ballot problem that involves predicting the probability that a candidate A is strictly ahead of candidate B during the vote count.

2 Summary of the results

In this section, we will be explaining how the following tasks has been solved. The following code for this project can be found at this: Github Repo In task a), the total number of possible paths was calculated to 35. For task b), after applying the condition, the number of valid paths was reduced to 7 paths, while considering the opposite constraint for task c) was found to be 28 paths. Task d) in 1703 valid paths after applying the reflection principle. Lastly in the final task, the probability of A to always stay ahead of B was calculated to be approximately 63,6 percent corresponding to 35 paths.

2.1 Task a)

The purpose with task a is to determine all possible paths the particle can take to move from the start (0, 3) to the end (7, 2), using only the movement rules "U" and "L" respectively, and upholding the constraints of these moves when calculating all the possible paths. The movement describes a combinatorial problem involving finding all the possible ways to arrange a fixed number of a given sequence(s), which in this case involves the "U" and "L" ways. To enforce this mathematically, we first need to calculate the number of downward and upward moves given in (1) and (2). Then, given the formula in (3), we receive the solution:

$$\text{Number of paths} = \binom{7}{3} = \frac{7!}{3!(7-3)!} = 35 \tag{4}$$

The code for this calculation can be seen below. We simply calculate the total number of possible paths from the total number of moves (in this case 7) and the total number of upward moves (in this case 3) and then taking the binomial coefficient from these two numbers. This works because we are looking at all the possible ways to arrange to arrange 3 'U' moves without order and without repetition among 7 possible "slots".

```
def calc_total_paths(start_point, end_point):
    x1, y1 = start_point
    x2, y2 = end_point

    total_moves = x2 - x1
    upward_moves = (total_moves + (y2 - y1)) // 2
    total_paths = binomial_coefficient(total_moves, upward_moves)
    return total_paths
```

The function was reused in task b, c and d to calculate the total number of paths between some conditions.

The movement rules from (1) and (2) was used to create a path function that iteratively adjusting the coordinates based on the move "U" and "L". The function then returns all the added coordinates in a list.

```
def combinations():
    moves = ['U'] * 3 + ['L'] * 4  # 3 U's and 4 L's
    unique_paths = set(itertools.permutations(moves))
    return unique_paths

 def ul_to_coords(path):
    x, y = 0,3 # Start coordinate

    for move in path:
        if move == 'U':
            x += 1
            y += 1
        elif move == 'L':
            x += 1
            y -= 1

        coordinates.append((x,y))
    return coordinates
```

The possible unique combinations can be generated effectively using the inbuilt permutations function from the `itertools` library, applying it to the "U" and "L" sequence stored in the "move" variable. These two functions were used throughout all tasks to validate the answers, and plot the lattice paths.

2.2 Task b)

Further on to task b, we are interested in paths that cross or touch the x-axis at least once. Considering the figure in the first task, this happens at (3,0) and (5,0). To solve this, we can use the symmetry of the lattice path to reflect the end point (also called Andre's Reflection Principle) and calculate the number paths that cross the x-axis by taking the binomial coefficient of this reflected point (7, -2). This gives us:

$$\text{Number of paths} = \binom{7}{1} = \frac{7!}{1!(7-1)!} = 7 \tag{5}$$

To verify this solution, and identify each particular path, we added the following condition for the code in task a:

```
def isolate_paths(combinations):
    return [ul_to_coords(path) for path in combinations if (3, 0)
    in ul_to_coords(path) and (5, 0) in ul_to_coords(path)]
```

This condition ensures that only the paths fulfilling the condition is included in the subset of the permutations.

2.3 Task c)

Compared to this task, we are asked to determine all the valid paths given our start and end coordinator that never touch or cross the x-axis which simultaneously implies that the y-coordinate remains positive throughout. Again we use the Reflection Principle, but now we subtract the number of paths that cross the x-axis from the total number of paths:

$$\text{Paths that cross x-axis} = \binom{7}{1} = \frac{7!}{1!(7-1)!} = 7 \tag{6}$$

$$\text{Paths that don't cross x-axis} = 35 - 7 \tag{7}$$

Again to verify and get each singular path we take all the permutations of the moves, and exclude all the paths that violate the constraints. To enforce this constraint, we simulate each sequence of moves and track the current position (x,y). After each move, we check whether the y-coordinate has dropped 0 or below. Mathematically, it can be reasoned as:

$$y_i > 0 \text{ for all steps } i \in [x_1, x_2]$$

As we simulate each path and only keep the coordinates that remains above the x-axis, we change the condition from task 3 to instead disregard these paths by following:

```
def isolate_paths(combinations):
    return [ul_to_coords(path) for path in
    combinations if (3, 0) not in ul_to_coords(path) and (5, 0)
    not in ul_to_coords(path)]
```

2.4 Task d)

This can be solved by the reflection principle, but this time using the new points (7, 6) and (20, 5), reflecting (20, 5) to (20, -5).

If it has, the path is marked as invalid. Only those paths that always keep the y-coordinate positive and end at the correct destination (7,2) are considered valid paths.

Given the movement rules at (1), we are tasked to find all the valid paths that never crosses or touches the x-axis between different coordinates: (7,6) and (20,5). Mathematically, the total number of paths can be solved as following:

$$y_2 - y_1 = 20 - 7 = 13 \quad \text{steps up}$$

$$x_2 - x_1 = 5 - 6 = -1 \quad \text{steps down}$$

Applying the (1) and (2) will give the system equations:

$$2u = 12 \quad \Rightarrow \quad u = 6$$

Substituting into $u + d = 13$:

$$6 + d = 13 \quad \Rightarrow \quad d = 7$$

Thus, giving us 6 upward "U" and 7 downward "L" moves in total. Applying the binomial coefficient formula, we can calculate the total number of paths to be: Since we only want paths that don't cross nor touch the x-axis, we are interested in the valid paths given by a constraint. Reflection principle was used to count the number of invalid paths, which can be simulated by reflecting the y-coordinate of the endpoint across the x-axis. The reflected coordinate for (20, 5) is (20, -5), and setting this into the binomial coefficient:

$$\frac{13 + (-5 - 6)}{2} = 1 \Rightarrow \binom{13}{1} = 13$$

Thus, the direct calculation will give us:

$$\text{Total Paths} = \binom{13}{6} = \frac{13!}{6!7!} = 1716$$

The valid paths are:

$$\text{Total paths} - \text{Invalid paths} = 1716 - 13 = 1703$$

The corresponding code for the reflection principle was applied in a similar way like the theory:

```
reflection = (end_point[0], -end_point[1])
invalid_paths = calc_total_paths(start_point, reflection)
valid_paths = total_paths - invalid_paths
```

where we calculate the total combinations of the invalid paths, subtracting it from the total paths. Given different upward and downward paths, we then use these values to update our values in our combinations function:

```
def combinations():
    moves = ['U'] * 6 + ['L'] * 7
```

And then filtered with the constraints below to exclude all paths that cross x-axis after the first move (since it starts at (0, 0):

```
return [ul_to_coords(path) for path in combinations if all(y != 0
↪  for x, y in ul_to_coords(path))]
```

2.4.1 Task 3.2

The next task was solved by using the Ballot Theorem, which states that given $a > b$, the probability that A is strictly ahead of B at all times during the counting process is given by (5): Finally, $a = 9$ votes for A and $b = 2$ votes for B:

$$\frac{a-b}{a+b} = \frac{9-2}{9+2} = \frac{7}{11} \approx 0.6364$$

The total number of ways to interleave the votes is calculated by using the binomial formula:

$$\binom{9+2}{2} = \binom{11}{2} = \frac{11*10}{2*1} = 55$$

Given the number of total ways (55) and the probability for A to stay strictly ahead (0.6364) we get the number of ways by multiplying:

$$\text{Valid Ways} = 55 \cdot 0.6364 = 35$$

The corresponding code implementation of this task is given by the probabilistic interpretation of the theorem:

```
def ballot_theorem(a, b):
    #step 1:
    if a <= b:
        return 0 #invalid, a cannot always be ahead of b

    #step 2:
    #ballot theorem, probability that a is strictly ahead of b,
    # given that a > b = (a-b)/(a+b)
    probability = ((a - b)) / (a + b)

    #step 3:
    total_ways = binomial_coefficient(a + b, b)
    valid_ways = total_ways * probability

    return probability, total_ways, valid_ways
```

The first step separates the invalid values where the function returns 0 if a is less than b. The next step involves calculating the probability of A and storing it in the "probability" variable. The last step arranges all the possible ways of the binomial coefficient, and printing out the results of the number of ways A stays ahead of B.

3 Mathematical Formulas and Equations

In this section, we explain the techniques used to solve the tasks of determining valid paths between specified points under given constraints. Given that we'll be counting and analyzing possible paths based on specific movement rules all while ensuring that the paths follow the predefined conditions. The following movement rules have been applied consistently as a fundamental reference for navigating through the lattice paths in all tasks:

$$U : (m, n) \to (m + 1, n + 1) \tag{1}$$
$$L : (m, n) \to (m + 1, n - 1) \tag{2}$$

3.1 Binomial Coefficients

Used consistently throughout the project, where n stands for number of items, corresponding to the total number of steps in the path and k: number of steps in the "L" or "U" direction. The denominator tells us the n given ways of arranging the n distinct elements, while $n - k$ are left choosing k elements:

$$\text{Number of paths} = \binom{n}{k} = \frac{n!}{k!(n-k)!} \tag{3}$$

3.2 Ballot Theorem

If candidate A receives a votes and candidate B receives b votes where $a > b$ at all times, then the probability for A to always be ahead of B is given by:

$$P(A > B) = \frac{a-b}{a+b}$$

This implies that at any point, A should be greater than the number of votes for B. Together with the binomial coefficient formula we will get that there are:

$$\binom{a+b}{b} = \frac{(a+b)!}{b!(a+b-b)!} = \frac{(a+b)!}{b!(a!)}$$

number of ways to combine a votes for A and b votes for B to be true. By only considering the valid ways, we get that there will be this many different ways to distribute the votes:

$$\text{Valid Ways} = \binom{a+b}{b} \cdot \frac{a-b}{a+b} \tag{5}$$

3.3 Python Libraries

In this project, several Python libraries were utilized to efficiently generate, filter and visualize valid paths given the movement rules. The matplotlib library has important visualizing tools to plot and graphically represent the customized graphs. The itertools library provided a permutations functions, playing a crucial role in generating possible combinations of the moves. As the sets became more complex by the given constraints and the set became larger, the sets was efficiently generated.

4 Explanatory Diagrams

This part of the report visually illustrates the valid paths across the Cartesian plane, providing an intuitive understanding of the overall solution process. With the number of "U"s and "L"s calculated earlier, each path consists of a combination of 3 upward and 4 upward movements. Each dot represents the coordinates after each move, with lines representing the path taken by following the "L" and "U" moves. Given the following graphs, the starting point for all path begins at (0,3) and ends at (7,2).

4.1 Task a)

The first task involved calculating all the possible unique combinations of the two different movements (combinatorial set of paths), which visually represents a plane on a 2D grid. Each path is a unique sequence of "U"s and "L"s, with their unique trajectory creating a "net" of lines. What looks like a plane, hints us that these are valid paths that span a region of the plane.

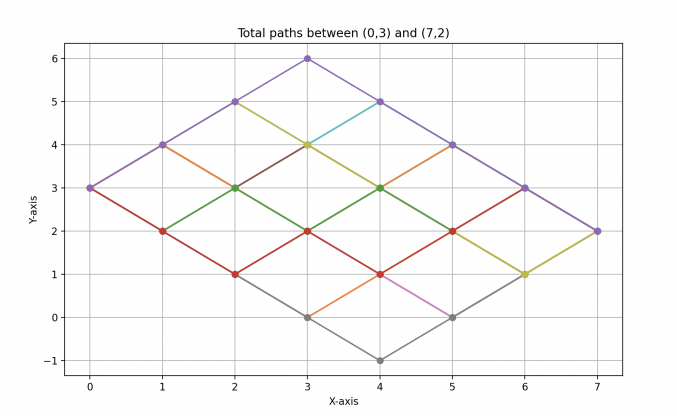

Figure 1: All the possible paths between (0,3) to (7,2)

4.2 Task b)

The diagram represents both upward and downward diagonal moves, crossing the x-axis at (3,0) or (5,0) at least once, which happens twice in this graph.

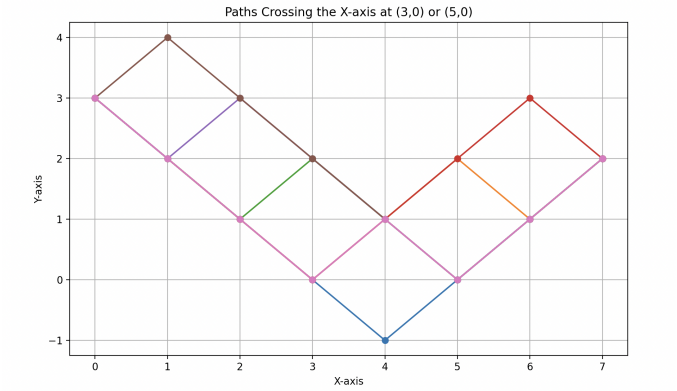

Figure 2: Possible paths where the graph crosses the x-axis at least once

4.3 Task c)

In the third task, we are constrained to calculate all the paths that doesn't cross the x-axis. Therefore, any path that would normally drop below the x-axis is invalid and therefore not shown.

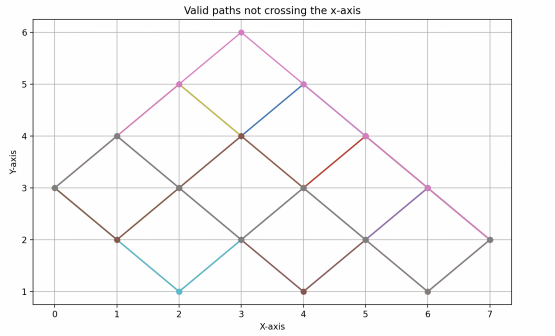

Figure 3: Possible paths where the graph never crosses the x-axis

4.4 Task d)

Represents all the set of possible paths where the paths never crosses the x-axis.

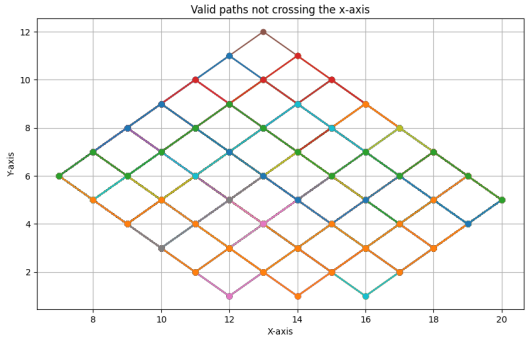

Figure 4: Possible paths where the graph never crosses the x-axis.

4.5 Task 3.2

The paths illustrate how A maintains the lead throughout the process, staying consistently above the origin point. Given the election scenario, this represents all the possible paths where A is always ahead of B.

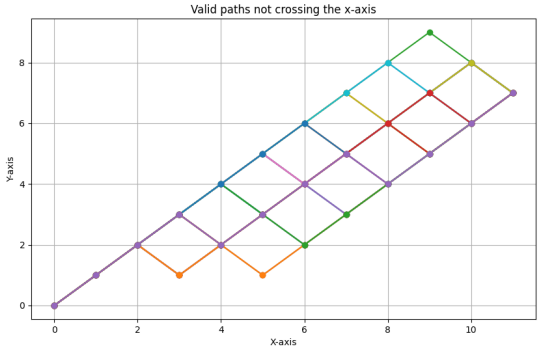

Figure 5: valid paths where candidate A is constantly before candidate B.

5 Brief Discussion

In this project, we've learned to apply both theoretical and computational methods to analyze the valid paths that satisfy certain conditions. In the last assignment,

the Ballot Theorem was a key tool that allowed us to compute the theoretical probability for A to stay ahead of B throughout the voting process. When reaching the fourth task, we realized the drawbacks of permuting all possible combination (itertools) when the number of permutations grew. Meaning, as the paths grew, the problem's complexity grew making it impractical to generate plots for thousands of possible outcomes. Eventually, we managed to plot all the possible paths, but it took a substantial amount of time. Nevertheless, it was insightful in enhancing our current knowledge of the importance of interpreting mathematical frameworks in a broader context.

6 Conclusions

The results emphasizes the importance of using combinatorics when solving real-world problems, especially where the outcome depends on curtain constraints. In a real-life scenario, this could be interpreted as maintaining as how curtain advantages needs to be maintained in order to reach a desired result, which is highly relevant in decision-making processes. The path-dependent, combinatorial outcomes provide valuable insights into how different choices or events lead to distinct results, illustrating how, depending on which decision it was taken, affects the future possibilities. It was very useful in task 3.2, to see how a seemingly unrelated problem can be converted to (or thought of as) as simple lattice path problem, making it easier to visualize and programatically solve. This project so far has been extremely useful as a foundation to be further specializing in fields such as finance, strategic planning and other competitive scenarios. By adhering to specific constraints, these methods can be critical for achieving success in these environments.

7 Code

Code used throughout the project that was mentioned in this report:

```
\begin{minted}{python}
#TASK A:
import math
import itertools
import matplotlib.pyplot as plt

start_point = (0, 3)
end_point = (7, 2)

def L(m, n):
    return (m+1, n-1)
def U(m, n):
    return (m+1, n+1)

def binomial_coefficient(n, k):
    return math.factorial(n) / (math.factorial(k) * math.factorial(n - k))

def calc_total_paths(start_point, end_point):
    x1, y1 = start_point
    x2, y2 = end_point
```

```
    #total nr of moves
    total_moves = x2 - x1
    #moves up (U)
    upward_moves = (total_moves + (y2 - y1)) // 2
    #moves down (L)
    downward_moves = total_moves - upward_moves
    #total number of paths
    total_paths = binomial_coefficient(total_moves, upward_moves)
    return total_paths

#produces all combinations of UL-paths possible
def ul_combinations():
    moves = ['U'] * 3 + ['L'] * 4  # 3 U's and 4 L's
    unique_paths = set(itertools.permutations(moves))  #generate all unique permutations

    return unique_paths

#makes UL path into x,y path
def ul_to_coords(path):
    """
    Simulates the path based on the sequence of moves ('U' and 'L')
    and returns the list of coordinates.
    """
    #start at (0, 3)
    x, y = 0, 3
    coordinates = [(x, y)]  #start coordinate

    #apply each move in the path
    for move in path:
        if move == 'U':
            x, y = U(x, y)  #use U function to move up
        elif move == 'L':
            x, y = L(x, y)  #use L function to move down
        coordinates.append((x, y))  #add the new coordinates

    return coordinates

def coord_paths(combinations):
    #use list comprehension to simulate paths and check if they cross (3,0) or (5,0)
    return [ul_to_coords(path) for path in combinations]

def plot_paths(paths):
    plt.figure(figsize=(10, 6))  # Set the figure size

    # Plot each path
    for path in paths:
        x_values, y_values = zip(*path)  # Separate x and y coordinates
        plt.plot(x_values, y_values, marker='o')

    # Configure the plot's appearance
    plt.title("Valid paths not crossing the x-axis")
    plt.xlabel("X-axis")
    plt.ylabel("Y-axis")
    plt.grid(True)
    plt.show()  # Display the plot

total_paths = calc_total_paths(start_point, end_point)
```

```
print(f"Total paths from point {start_point} to point {end_point}: {total_paths}")
unique_paths = ul_combinations()
path_coords = coord_paths(unique_paths)


n = 1
for path in path_coords:
    path_str = " -> ".join([f"({x},{y})" for x, y in path])
    print(f"{path_str} n = {n}")
    n += 1

plot_paths(path_coords)

#TASK B:
import math
import itertools
import matplotlib.pyplot as plt

start_point = (0, 3)
end_point = (7, 2)

def L(m, n):
    return (m+1, n-1)
def U(m, n):
    return (m+1, n+1)

def binomial_coefficient(n, k):
    return math.factorial(n) / (math.factorial(k) * math.factorial(n - k))



def calc_total_paths(start_point, end_point):
    x1, y1 = start_point
    x2, y2 = end_point

    #total nr of moves
    total_moves = x2 - x1
    #moves up (U)
    upward_moves = (total_moves + (y2 - y1)) // 2
    #total number of paths
    total_paths = binomial_coefficient(total_moves, upward_moves)
    return total_paths

#produces all combinations of UL-paths possible
def ul_combinations():
    moves = ['U'] * 3 + ['L'] * 4  # 3 U's and 4 L's
    unique_paths = set(itertools.permutations(moves))  # Generate all unique permutations

    return unique_paths

#makes UL path into x,y path
def ul_to_coords(path):
    """
    Simulates the path based on the sequence of moves ('U' and 'L')
    and returns the list of coordinates.
    """
    #start at (0, 3)
```

```
    x, y = 0, 3
    coordinates = [(x, y)]  # Start coordinate

    #apply each move in the path
    for move in path:
        if move == 'U':
            x, y = U(x, y)  #use U function to move up
        elif move == 'L':
            x, y = L(x, y)  #use L function to move down
        coordinates.append((x, y))  #add the new coordinates

    return coordinates

def isolate_paths(combinations):
    #use list comprehension to simulate paths and check if they cross (3,0) or (5,0)
    return [ul_to_coords(path) for path in combinations if (3, 0) in ul_to_coords(path) or

def plot_paths(paths):
    plt.figure(figsize=(10, 6))  # Set the figure size

    # Plot each path
    for path in paths:
        x_values, y_values = zip(*path)  # Separate x and y coordinates
        plt.plot(x_values, y_values, marker='o')

    # Configure the plot's appearance
    plt.title("Valid paths not crossing the x-axis")
    plt.xlabel("X-axis")
    plt.ylabel("Y-axis")
    plt.grid(True)
    plt.show()  # Display the plot

total_paths = calc_total_paths(start_point, end_point)
#andre's reflection principle
reflection = (end_point[0], -end_point[1])
#using reflection as end point we calculate total nr of points that do reach x-axis
valid_paths = calc_total_paths(start_point, reflection)
invalid_paths = total_paths - valid_paths
ul_paths = ul_combinations()
isolated_paths = isolate_paths(ul_paths)

print(f"Total paths from point {start_point} to point {end_point}: {total_paths}")
print(f"Number of invalid paths: {invalid_paths}")
print(f"Number of valid paths: {valid_paths}")

n = 1
for path in isolated_paths:
    path_str = " -> ".join([f"({x},{y})" for x, y in path])
    print(f"{path_str} n = {n}")
    n += 1

plot_paths(isolated_paths)

#TASK C:
import math
import itertools
import matplotlib.pyplot as plt
```

```python
start_point = (0, 3)
end_point = (7, 2)

def L(m, n):
    return (m+1, n-1)
def U(m, n):
    return (m+1, n+1)

def binomial_coefficient(n, k):
    return math.factorial(n) / (math.factorial(k) * math.factorial(n - k))



def calc_total_paths(start_point, end_point):
    x1, y1 = start_point
    x2, y2 = end_point

    #total nr of moves
    total_moves = x2 - x1
    #moves up (U)
    upward_moves = (total_moves + (y2 - y1)) // 2
    #total number of paths
    total_paths = binomial_coefficient(total_moves, upward_moves)
    return total_paths

#produces all combinations of UL-paths possible
def ul_combinations():
    moves = ['U'] * 3 + ['L'] * 4  # 3 U's and 4 L's
    unique_paths = set(itertools.permutations(moves))  # Generate all unique permutations

    return unique_paths

#makes UL path into x,y path
def ul_to_coords(path):
    """
    Simulates the path based on the sequence of moves ('U' and 'L')
    and returns the list of coordinates.
    """
    #start at (0, 3)
    x, y = 0, 3
    coordinates = [(x, y)]  #start coordinate

    #apply each move in the path
    for move in path:
        if move == 'U':
            x, y = U(x, y)  #use U function to move up
        elif move == 'L':
            x, y = L(x, y)  #use L function to move down
        coordinates.append((x, y))  #add the new coordinates

    return coordinates

def isolate_paths(combinations):
    #use list comprehension to simulate paths and check if they cross (3,0) or (5,0)
    return [ul_to_coords(path) for path in combinations if (3, 0) not in ul_to_coords(path)
```

```python
def plot_paths(paths):
    plt.figure(figsize=(10, 6))  # Set the figure size

    # Plot each path
    for path in paths:
        x_values, y_values = zip(*path)  # Separate x and y coordinates
        plt.plot(x_values, y_values, marker='o')

    # Configure the plot's appearance
    plt.title("Valid paths not crossing the x-axis")
    plt.xlabel("X-axis")
    plt.ylabel("Y-axis")
    plt.grid(True)
    plt.show()  # Display the plot

total_paths = calc_total_paths(start_point, end_point)
#andre's reflection principle
reflection = (end_point[0], -end_point[1])
#using reflection as end point we calculate total nr of points that do reach x-axis
invalid_paths = calc_total_paths(start_point, reflection)
valid_paths = total_paths - invalid_paths
ul_paths = ul_combinations()
isolated_paths = isolate_paths(ul_paths)

print(f"Total paths from point {start_point} to point {end_point}: {total_paths}")
print(f"Number of invalid paths: {invalid_paths}")
print(f"Number of valid paths: {valid_paths}")


n = 1
for path in isolated_paths:
    path_str = " -> ".join([f"({x},{y})" for x, y in path])
    print(f"{path_str} n = {n}")
    n += 1

plot_paths(isolated_paths)

#TASK D:
import math
import itertools
import matplotlib.pyplot as plt

start_point = (7, 6)
end_point = (20, 5)

def L(m, n):
    return (m+1, n-1)
def U(m, n):
    return (m+1, n+1)

def binomial_coefficient(n, k):
    return math.factorial(n) / (math.factorial(k) * math.factorial(n - k))


def calc_total_paths(start_point, end_point):
    x1, y1 = start_point
    x2, y2 = end_point
```

```
    #total nr of moves
    total_moves = x2 - x1
    #moves up (U)
    upward_moves = (total_moves + (y2 - y1)) // 2
    #total number of paths
    total_paths = binomial_coefficient(total_moves, upward_moves)
    return total_paths

#produces all combinations of UL-paths possible
def ul_combinations():
    moves = ['U'] * 6 + ['L'] * 7  # 6 U's and 7 L's
    unique_paths = set(itertools.permutations(moves))  # Generate all unique permutations

    return unique_paths

#makes UL path into x,y path
def ul_to_coords(path):

    #start at (0, 3)
    x, y = 7, 6
    coordinates = [(x, y)]  #start coordinate

    #apply each move in the path
    for move in path:
        if move == 'U':
            x, y = U(x, y)  #use U function to move up
        elif move == 'L':
            x, y = L(x, y)  #use L function to move down
        coordinates.append((x, y))  #add the new coordinates

    return coordinates

def isolate_paths(combinations):

    #filter paths where y never equals 0, excluding all paths that touch x-axis
    return [ul_to_coords(path) for path in combinations if all(y != 0 for x, y in ul_to_coo

def plot_paths(paths):
    plt.figure(figsize=(10, 6))  # Set the figure size

    # Plot each path
    for path in paths:
        x_values, y_values = zip(*path)  # Separate x and y coordinates
        plt.plot(x_values, y_values, marker='o')

    # Configure the plot's appearance
    plt.title("Valid paths not crossing the x-axis")
    plt.xlabel("X-axis")
    plt.ylabel("Y-axis")
    plt.grid(True)
    plt.show()  # Display the plot


total_paths = calc_total_paths(start_point, end_point)
#andre's reflection principle
reflection = (end_point[0], -end_point[1])
```

```python
#using reflection as end point we calculate total nr of points that do reach x-axis
invalid_paths = calc_total_paths(start_point, reflection)
valid_paths = total_paths - invalid_paths
ul_paths = ul_combinations()
isolated_paths = isolate_paths(ul_paths)


print(f"Total paths: {total_paths}")
print(f"Invalid paths: {invalid_paths}")
print(f"Valid paths {valid_paths}")

"""
n = 1
for path in isolated_paths:
    path_str = " -> ".join([f"({x},{y})" for x, y in path])
    print(f"{path_str} n = {n}")
    n += 1
"""

plot_paths(isolated_paths)



#TASK 3.2:
import math
import itertools
import matplotlib.pyplot as plt

a_votes = 9
b_votes = 2

def L(m, n):
    return (m+1, n-1)
def U(m, n):
    return (m+1, n+1)

def binomial_coefficient(n, k):
    return math.factorial(n) / (math.factorial(k) * math.factorial(n - k))

def ballot_theorem(a, b):
    if a <= b:
        return 0 #invalid, a cannot always be ahead of b

    #ballot theorem, probability that a is strictly ahead of b, given that a > b = (a-b)/(a
    probability = ((a - b)) / (a + b)

    total_ways = binomial_coefficient(a + b, b)
    valid_ways = total_ways * probability

    return probability, total_ways, valid_ways

#produces all combinations of UL-paths possible
def ul_combinations():
    moves = ['U'] * 9 + ['L'] * 2  # 9 U's and 2 L's
    unique_paths = set(itertools.permutations(moves))  # Generate all unique permutations
```

```
    return unique_paths

#makes UL path into x,y path
def ul_to_coords(path):
    """
    Simulates the path based on the sequence of moves ('U' and 'L')
    and returns the list of coordinates.
    """
    #start at (0, 0)
    x, y = 0, 0
    coordinates = [(x, y)]  #start coordinate

    #apply each move in the path
    for move in path:
        if move == 'U':
            x, y = U(x, y)  #use U function to move up
        elif move == 'L':
            x, y = L(x, y)  #use L function to move down
        coordinates.append((x, y))  #add the new coordinates

    return coordinates

def isolate_paths(combinations):
    """
    Returns only the paths where y is never equal to 0 for any x-coordinate,
    excluding the starting point (0, 0).
    """
    return [ul_to_coords(path) for path in combinations if all(y != 0 for x, y in ul_to_coor

def plot_paths(paths):
    plt.figure(figsize=(10, 6))  # Set the figure size

    # Plot each path
    for path in paths:
        x_values, y_values = zip(*path)  # Separate x and y coordinates
        plt.plot(x_values, y_values, marker='o')

    # Configure the plot's appearance
    plt.title("Valid paths not crossing the x-axis")
    plt.xlabel("X-axis")
    plt.ylabel("Y-axis")
    plt.grid(True)
    plt.show()  # Display the plot


probability, total_ways, valid_ways = ballot_theorem(a_votes, b_votes)

print(f"There are {valid_ways} ways that A is always ahead of B, out of {total_ways} total w
print(f"The probability that A is always ahead of B is {(probability) * 100:.2f}%")

ul_paths = ul_combinations()
isolated_paths = isolate_paths(ul_paths)

n = 1
for path in isolated_paths:
    path_str = " -> ".join([f"({x},{y})" for x, y in path])
    print(f"{path_str} n = {n}")
```

```
    n += 1

plot_paths(isolated_paths)
```