Part A:

Validate the following probabilities of losing the white runners (busting) for the following two situations: When columns are (3, 8, 11), the chance of advancing is 76% and the chance of busting is 24%. (or 0.24151 to be exact). When columns are columns are (2, 4, 11) the chance of advancing is 63% and the chance of busting is 37%. (or 0.36574 to be exact).

Assumption:

Although the rules are clear, we are assuming that each player plays optimally, meaning if they can advance, then they will.

Model:

To calculate the chance of advancing and in turn the chance of busting in the game "Can't Stop" for given sets of columns.To calculate the probability of success given columns, a c++ script will achieve this with the following steps.

1. **Generate all possible dice rolls:** Create all possible combinations of four dice rolls, where each die takes a value between 1 and 6.
2. **Check if Pairs can form sums in given target columns:** For each combination of dice rolls, we can check if the sums of any pairs of dice match the target columns.
3. **Calculate probabilities:** Count the number of dice roll combinations that allow advancing and calculate the probabilities of advancing and busting.
4. **Give specific input and output results,** inputting the specific columns (3, 8, 11) and (2,4,11) to validate the probabilities.

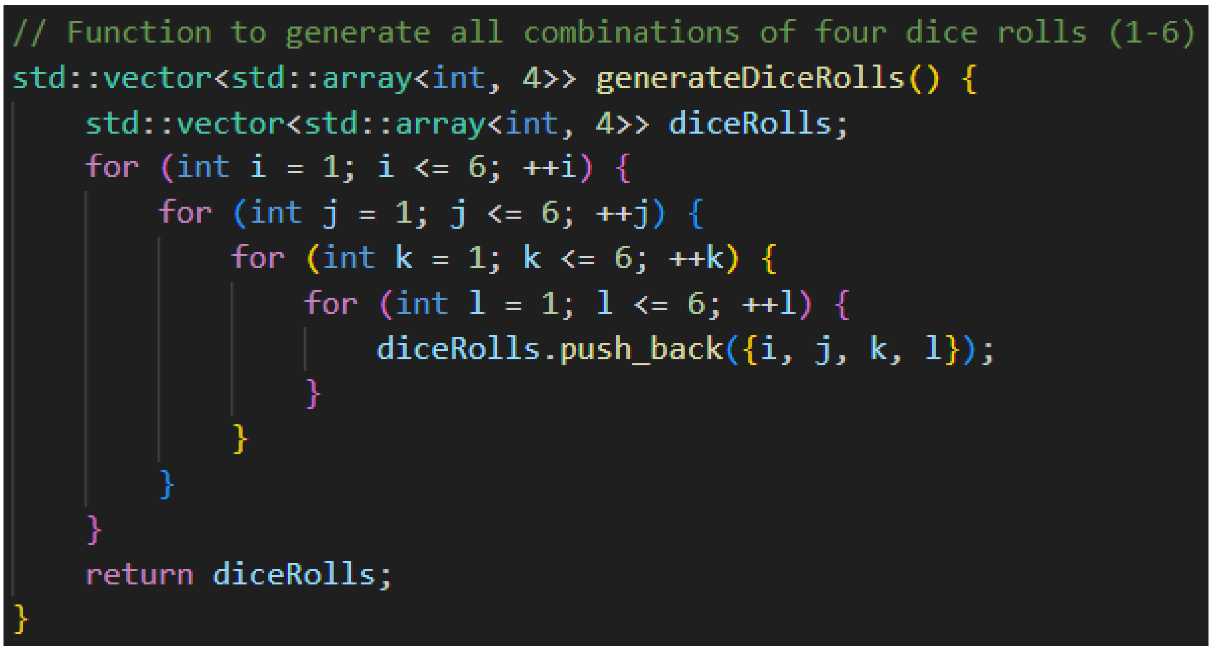
Generate all possible dice rolls:

This involves all possible outcomes of rolling four dice, with each die ranging from 1 to 6. Since each die is independent, the total number of possible outcomes $= 6^4 = 1296$. By generating all combinations, we ensure that our probability calculations are exhaustive, and accurate.

Approach:

- Use a function generateDiceRolls() which returns a vector of arrays, each of size 4.
- The value at index i in each array represents the roll for dice i + 1.

- Nested for loops will be used to iterate through all possible values for the four dice.



```cpp
// Function to generate all combinations of four dice rolls (1-6)
std::vector<std::array<int, 4>> generateDiceRolls() {
    std::vector<std::array<int, 4>> diceRolls;
    for (int i = 1; i <= 6; ++i) {
        for (int j = 1; j <= 6; ++j) {
            for (int k = 1; k <= 6; ++k) {
                for (int l = 1; l <= 6; ++l) {
                    diceRolls.push_back({i, j, k, l});
                }
            }
        }
    }
    return diceRolls;
}
```

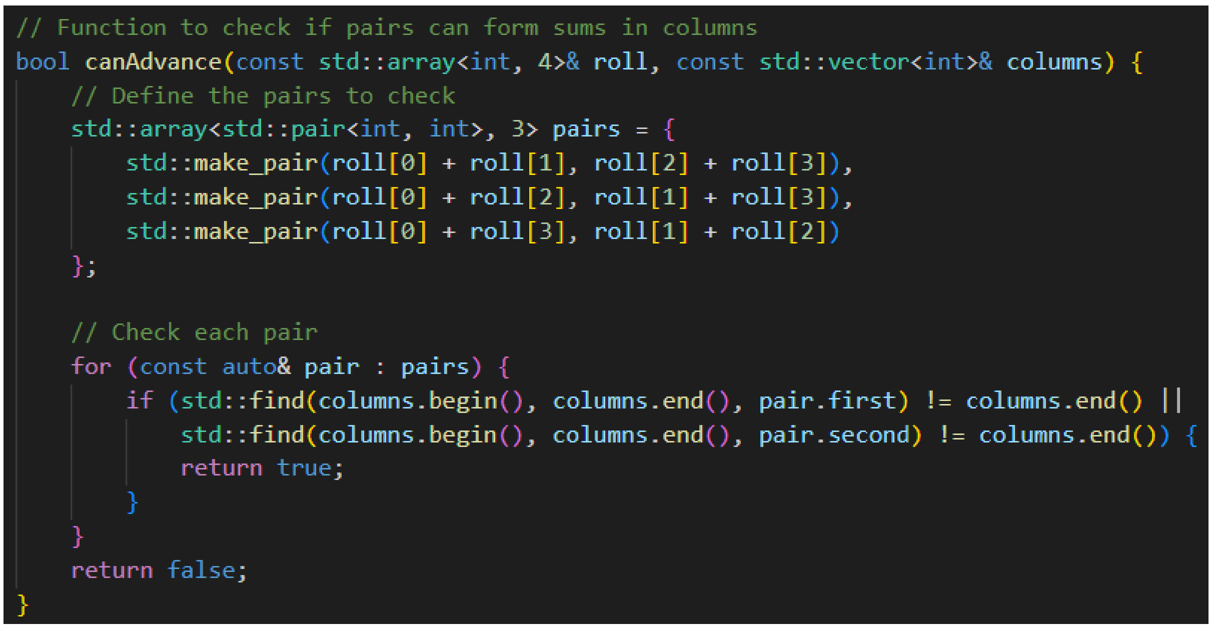
Check If Pairs Can Form Sums in Columns:

In this next part, we determine if any pairs of dice sums in a given roll match the target columns. Here is where we are assuming that the player will play optimally, and advance whenever possible.

For a given role of four dice, there are three unique ways to form two pairs since order doesn't matter (e.g. (1,3) is the same as (3,1)).

Approach:
- For each roll, form the three possible unique pairs of dice sums.
- Check if any of these pairs sums match the target columns.
- If any pair matches, we return true, indicating that the player can advance, otherwise we return false.
- To achieve this, the function will be called canAdvance(), and the input will be an array<int> of size 4, representing the four dice rolls, and a vector<int> where the value at index i represents a target sum column.

The code for the program can be observed below.



```cpp
// Function to check if pairs can form sums in columns
bool canAdvance(const std::array<int, 4>& roll, const std::vector<int>& columns) {
    // Define the pairs to check
    std::array<std::pair<int, int>, 3> pairs = {
        std::make_pair(roll[0] + roll[1], roll[2] + roll[3]),
        std::make_pair(roll[0] + roll[2], roll[1] + roll[3]),
        std::make_pair(roll[0] + roll[3], roll[1] + roll[2])
    };

    // Check each pair
    for (const auto& pair : pairs) {
        if (std::find(columns.begin(), columns.end(), pair.first) != columns.end() ||
            std::find(columns.begin(), columns.end(), pair.second) != columns.end()) {
            return true;
        }
    }
    return false;
}
```

Calculate Probabilities:

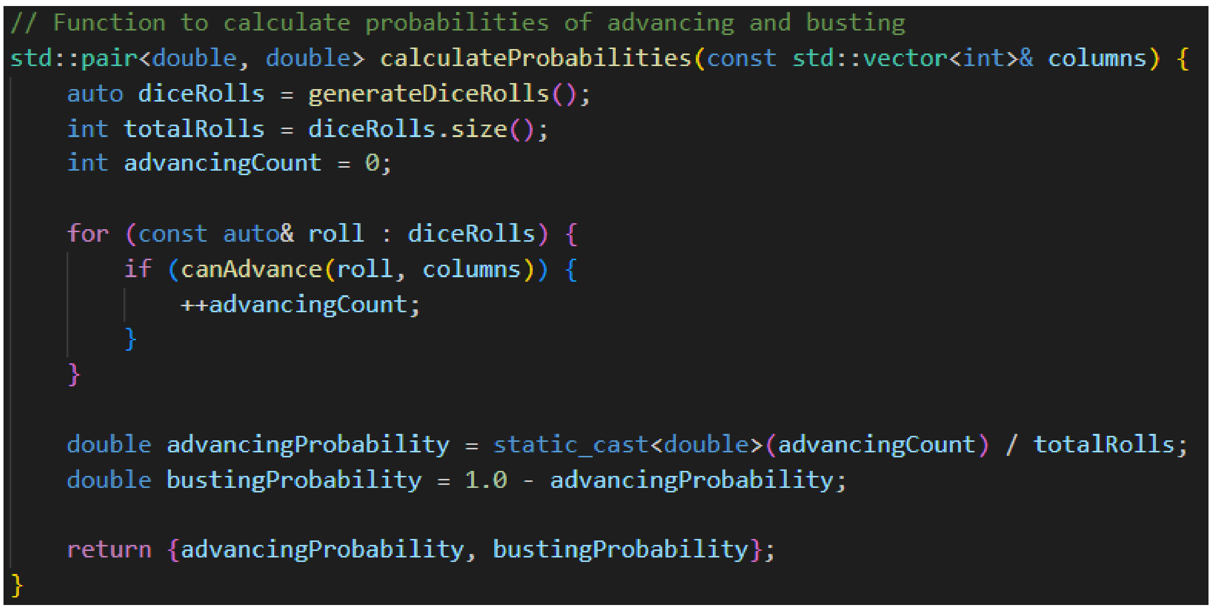
In this step, we iterate over all possible dice rolls and use the canAdvance function to count the number of rolls that allow advancing. This count is then used to calculate the probabilities.

Approach:

- Generate all possible dice rolls using generateDiceRolls().
- Initialise a counter advancingCount to track the number of rolls that allow advancing.
- For each roll, we will use canAdvance to check if it can form valid pairs.
- Calculate the advancing probability as the ratio of advancingCount to totalRolls.
- In turn, the busting probability is the complement of the advancing probability.

The function calculateProbabilities() will be used to perform these calculations. The input will be a vector<int> representing the target column sums.

The code of the calculateProbabilities() function can be observed in the image below.



```cpp
// Function to calculate probabilities of advancing and busting
std::pair<double, double> calculateProbabilities(const std::vector<int>& columns) {
    auto diceRolls = generateDiceRolls();
    int totalRolls = diceRolls.size();
    int advancingCount = 0;

    for (const auto& roll : diceRolls) {
        if (canAdvance(roll, columns)) {
            ++advancingCount;
        }
    }

    double advancingProbability = static_cast<double>(advancingCount) / totalRolls;
    double bustingProbability = 1.0 - advancingProbability;

    return {advancingProbability, bustingProbability};
}
```

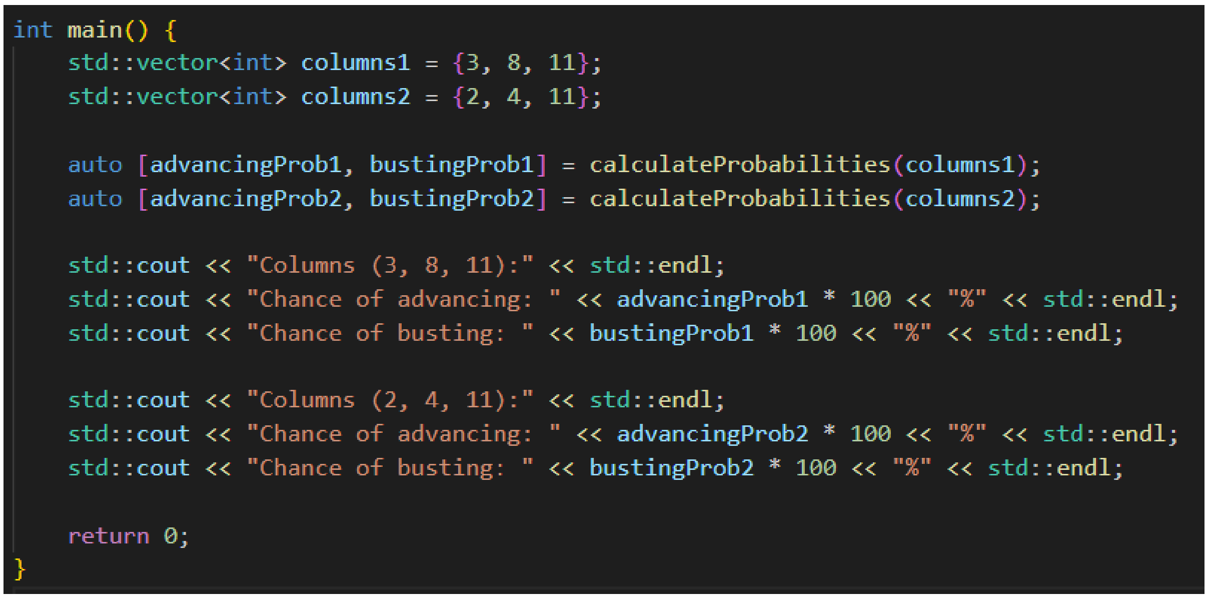
Give specific input and output the results:

In this step, we will print the calculated probabilities for the given columns, providing clear insights into the chances of advancing and busting.

Approach:

1. Define the target columns as intended above (3, 8, 11), and (2,4,11).
2. Call calculate probabilities for each set of columns to get the advancing and busting probabilities.
3. Print the results clearly, showing the chances of advancing and busting for each set of columns.

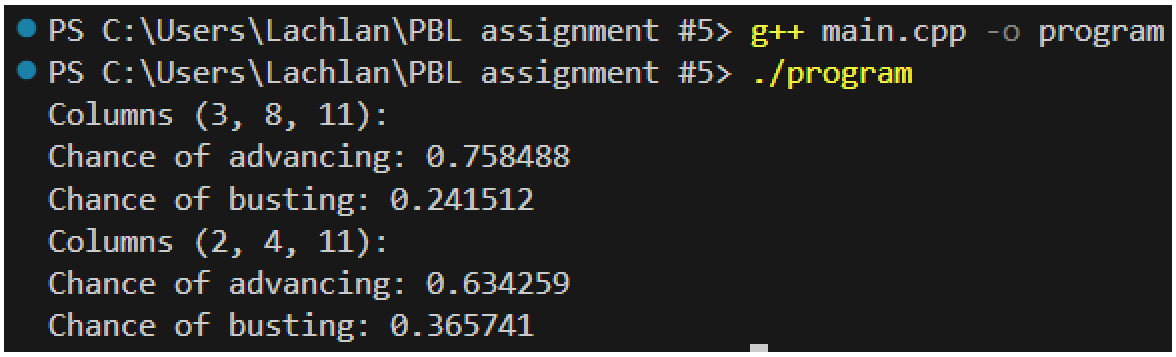
The main function will be used to execute these steps, and the input will be the predefined sets of target columns. The main function can be observed below.



```cpp
int main() {
    std::vector<int> columns1 = {3, 8, 11};
    std::vector<int> columns2 = {2, 4, 11};

    auto [advancingProb1, bustingProb1] = calculateProbabilities(columns1);
    auto [advancingProb2, bustingProb2] = calculateProbabilities(columns2);

    std::cout << "Columns (3, 8, 11):" << std::endl;
    std::cout << "Chance of advancing: " << advancingProb1 * 100 << "%" << std::endl;
    std::cout << "Chance of busting: " << bustingProb1 * 100 << "%" << std::endl;

    std::cout << "Columns (2, 4, 11):" << std::endl;
    std::cout << "Chance of advancing: " << advancingProb2 * 100 << "%" << std::endl;
    std::cout << "Chance of busting: " << bustingProb2 * 100 << "%" << std::endl;

    return 0;
}
```

Solution:

Upon running the program, the calculated probabilities were as follows:



```
PS C:\Users\Lachlan\PBL assignment #5> g++ main.cpp -o program
PS C:\Users\Lachlan\PBL assignment #5> ./program
Columns (3, 8, 11):
Chance of advancing: 0.758488
Chance of busting: 0.241512
Columns (2, 4, 11):
Chance of advancing: 0.634259
Chance of busting: 0.365741
```

Results table

Columns	**Calculated Chance of Advancing**	**Provided Chance of Advancing**	**Calculated Chance of Busting**	**Provided Chance of Busting**
(3, 8, 11)	0.758488	0.75849	0.241512	0.24151
(2, 4, 11)	0.634259	0.63426	0.365741	0.36574

The results above validate the chances stated in the question exactly, however the results calculated are slightly more precise as they are not rounded to 5 decimal places, but instead 6 decimal places.

Discussion

Methods for calculating probabilities often rely on random sampling or approximations, which can introduce a certain degree of uncertainty. This analysis, however employs a unique and clear approach by utilising exhaustive enumeration (considering every single possible dice roll combination for each target sum). This approach guarantees accurate results.

Despite clear game rules, one assumption was made, the player always chooses the optimal move, forming a pair to advance if possible. This is necessary, as a player could potentially not realise they can make a pair of their current columns with the dice that they rolled.

By generating all possible dice rolls (1296 combinations) we ensure that no possible outcome is missed. This approach guarantees the accuracy of the probabilities generated. The calculated probabilities match the provided probabilities, however, the provided values might be rounded to five decimal places, while the program offers greater precision with six decimal places. This slight discrepancy could be attributed to floating-point arithmetic limitations, rounding choices, or the use of exact versus approximated values in the provided probabilities.

We can understand the results by analysing the probabilities of the dice rolls that were investigated. To achieve a sum of 3, there are only two combinations, (1 & 2) or (2 & 1). Similarly, for a sum of 8, there are five combinations (2 & 6), (3 & 5), (4 & 4), (5 & 3) and (6 & 2). In contrast, achieving a sum of 2 requires just one combination (1 & 1), and reaching a sum of 4 has only three combinations (1 & 3), (2 & 2), and (3 & 1). Since there are more ways to roll a 3 or 8 compared to a 2 or 4, the total number of combinations that hit the target sums for columns (3, 8, 11) naturally outweighs those for columns (2, 4, 11), leading to a higher chance of advancing, which again validates the probabilities.

For columns (3, 8, 11), the higher chance of advancing suggests that these columns are more favourable for players to target compared to columns (2, 4, 11), which have a lower chance of advancing (63.43%). This means that the choice of columns significantly impacts the players strategy in the game, as targeting columns with higher advancing probabilities increases the likelihood of continued progress. However, it is important to consider the risk-reward trade-off when playing a game such as Can't Stop. While hitting a common target like 7 might be easier, successfully rolling a less likely sum like 2 or 4 offers a greater reward, as the player needs to advance fewer positions from a lower starting point. This introduces a strategic element, where players can weigh the risk of aiming for a less probable target against the potential reward of needing fewer successful rolls to advance.

Part B (i):

We want to understand the "ideal" number of throws, by working out the chance of busting after 1,2,3 or 4 throws. Simulate the game from the starting positions a large number of times to provide a reasonable approximation for these values.

Note: part B is split up into two parts, the first part being simulating the probabilities of the game, and the second part being utilising the simulation results to calculate expected progression.

Also, although the question says to investigate the chance of busting after 1, 2, 3, or 4 throws. I will also be investigating the chance of busting after 5 throws to better understand the ideal number of throws.

Assumptions:

- The player is starting from the very first position in the column.
- By "Ideal" number of throws, I am assuming this means optimising the expected progression at each decision point within the columns aforementioned. This means only focusing on total progression, and not focusing on the value of progressing on a less likely column.

Method (i):

This code simulates a game where a player rolls four dice multiple times to advance in specified columns. The simulation runs a large number of games to calculate the probabilities of busting after each throw (up to 4 throws). By analysing these probabilities, players can understand the risk associated with continuing to roll. For example, should they stop after one throw and secure a safe position, or risk another throw for potentially faster progress?

Following the simulation, the program will use the calculated bust probabilities to determine the "ideal" number of throws for each position within the column. This involves expected value calculations considering both the risk of busting and the potential reward of reaching higher positions faster.

To achieve this, the program will need to run multiple rounds of rolling the dice (5 in this case) and analyse probabilities from slightly further positions within the column. This additional step, explained in part B (ii) allows for a greater understanding of optimal play, and allows us to determine the optimal number of throws for the different columns mentioned in part A.

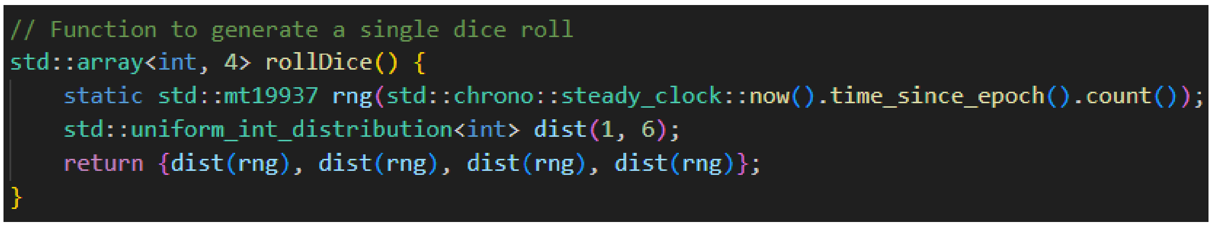
Function to generate a single dice roll:

Unlike part A, this time all possible values are not generated again as this will be too large for the simulation, instead this function returns the four dice rolls randomly. Provided enough trials, then this will produce an accurate result.

Approach:

- Utilises a random number generator, std::mt19937 initialised with the current time to ensure randomness.
- uniform_int_distribution<int> generates random numbers between 1 and 6 (inclusive).
- The function returns an array of four integers, each representing the outcome of a dice roll.

The function is shown below.



```cpp
// Function to generate a single dice roll
std::array<int, 4> rollDice() {
    static std::mt19937 rng(std::chrono::steady_clock::now().time_since_epoch().count());
    std::uniform_int_distribution<int> dist(1, 6);
    return {dist(rng), dist(rng), dist(rng), dist(rng)};
}
```

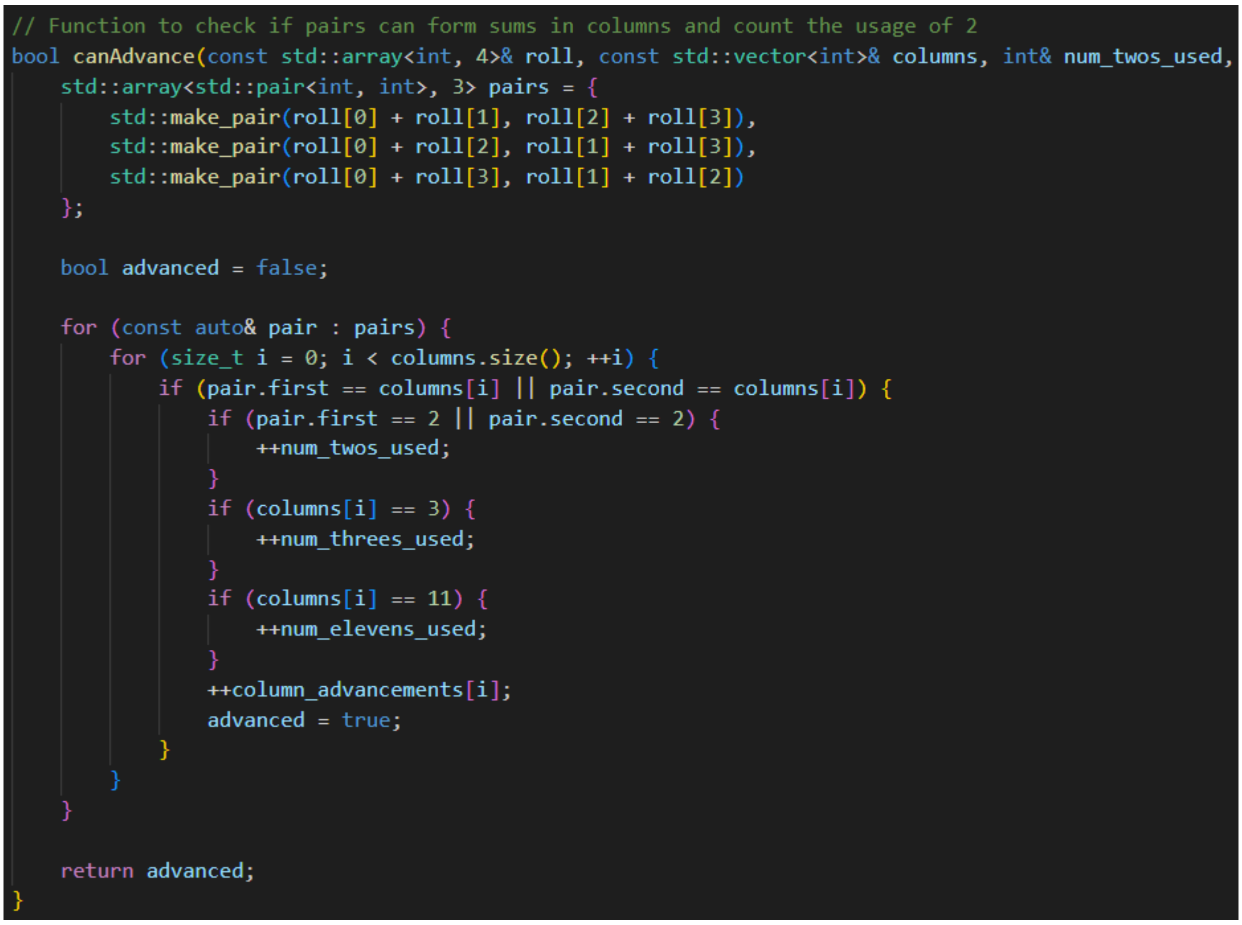
Function to check if pairs can form sums in columns, counting how many times the column's 2, 3, and 11 are progressed.

The function focuses on the columns identified in part A, under the assumption that the player starts in the first position. Columns 2, 3 and 11 are the only columns where reaching the top and busting is possible within 5 throws. As per the rules, if the player's traffic cone is at the end of the column, and that number is rolled, then it is a bust, hence it is essential to keep track of the amount of times the player has progressed in these columns. This function checks if any of the possible pairs of dice sums can advance the player in any of the specified columns.

Approach:

- It creates all possible pairs of sums from the dice rolls.
- It iterates over these pairs and the target columns to see if any pair matches a column.
- If a valid pair is found and it includes the number 2, 3 or 11, the respective count is incremented.
- The function returns true if any valid pair is found, and false otherwise

The code for this is shown below: **Function signature cut off as it was hard to read, the remaining signature is** int& num_elevens_used, std::vector<int>& column_advancements



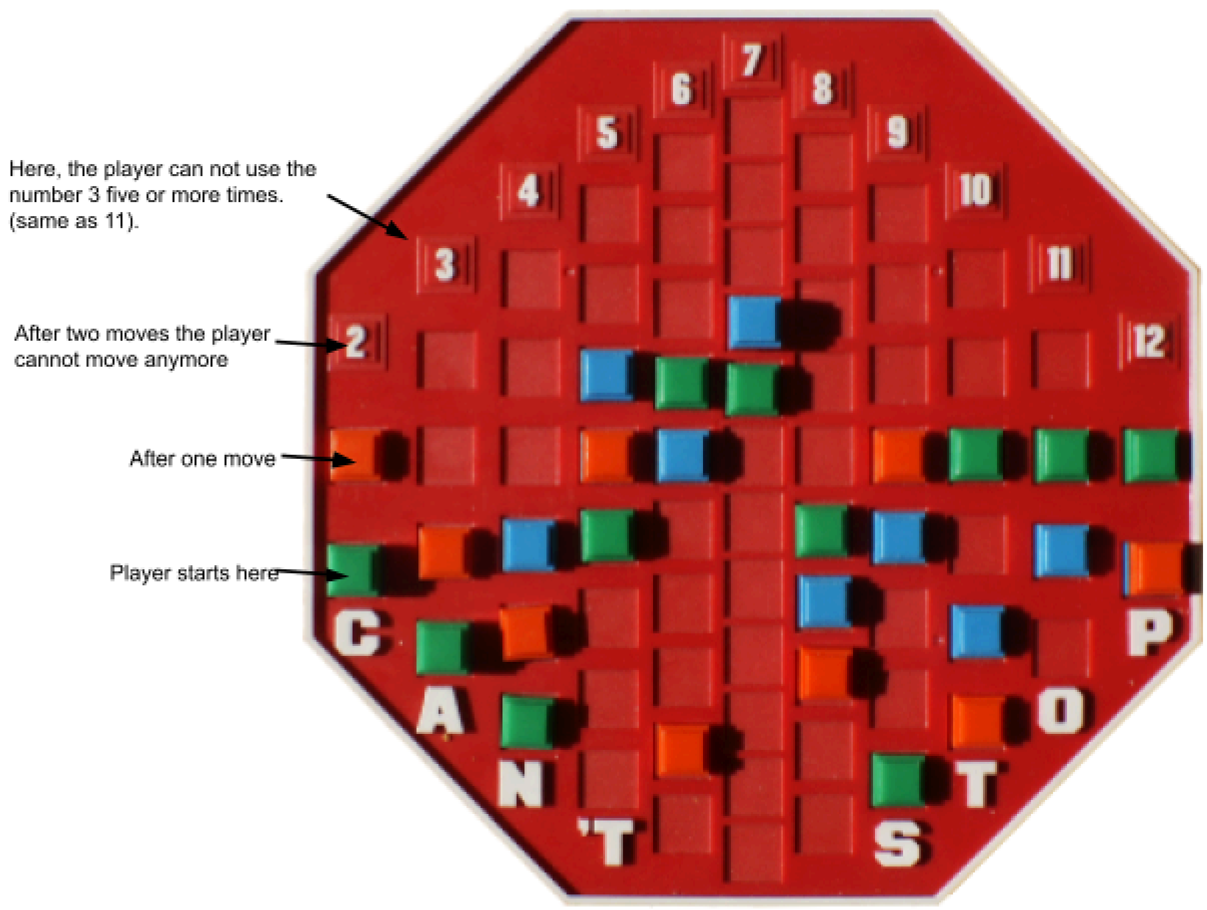
```cpp
// Function to check if pairs can form sums in columns and count the usage of 2
bool canAdvance(const std::array<int, 4>& roll, const std::vector<int>& columns, int& num_twos_used,
    std::array<std::pair<int, int>, 3> pairs = {
        std::make_pair(roll[0] + roll[1], roll[2] + roll[3]),
        std::make_pair(roll[0] + roll[2], roll[1] + roll[3]),
        std::make_pair(roll[0] + roll[3], roll[1] + roll[2])
    };

    bool advanced = false;

    for (const auto& pair : pairs) {
        for (size_t i = 0; i < columns.size(); ++i) {
            if (pair.first == columns[i] || pair.second == columns[i]) {
                if (pair.first == 2 || pair.second == 2) {
                    ++num_twos_used;
                }
                if (columns[i] == 3) {
                    ++num_threes_used;
                }
                if (columns[i] == 11) {
                    ++num_elevens_used;
                }
                ++column_advancements[i];
                advanced = true;
            }
        }
    }

    return advanced;
}
```

Function to simulate a single game and return the number of throws until busting or maximum number of throws reached.

This function simulates one game session, rolling the dice up to 5 times, and determines whether the player busts or reaches the maximum number of throws.

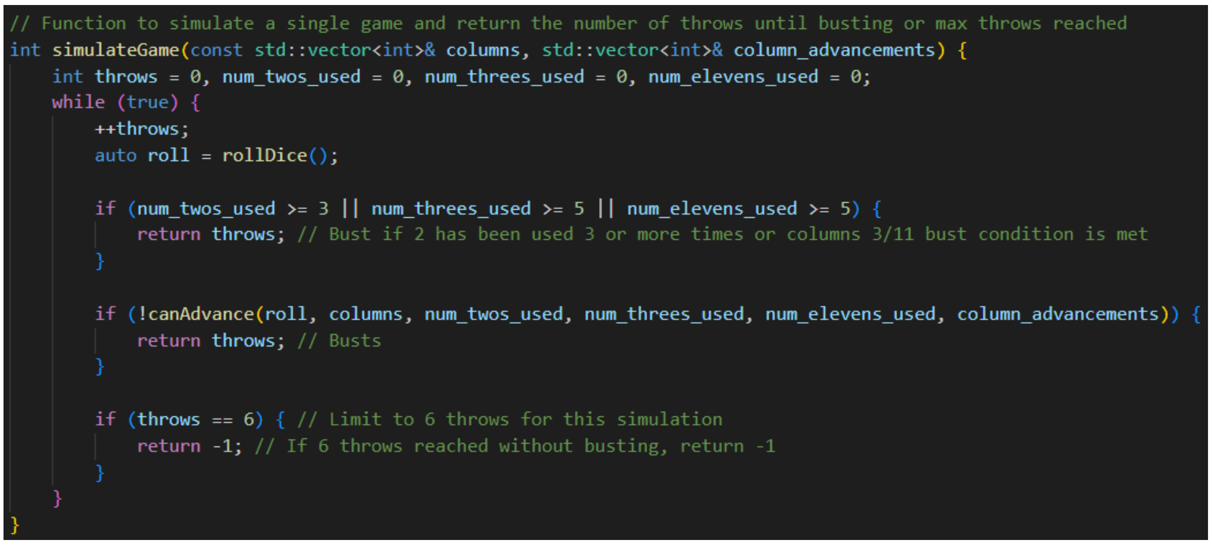
Approach

- The function keeps track of the number of throws and the number of times the numbers 2, 3 and 11 are used in valid pairs.
- It rolls the dice and checks for valid pairs using the canAdvance function.
- If the player advances on column 2 three or more times, then the game busts and the function returns the number of throws. The same logic is applied to the player busting if a 3, or 11 is used five or more times. This is illustrated below. The last position of all other columns which are being investigated cannot be reached as the game is limited to five throws.


Here, the player can not use the number 3 five or more times. (same as 11).
After two moves the player cannot move anymore
After one move
Player starts here
CAN'T STOP

- If the player cannot advance on a roll, the game busts, and the function returns the number of throws.
- If the player reaches five throws without busting, the function returns -1 (doesn't bust).

The code for this function can be observed below.



```
// Function to simulate a single game and return the number of throws until busting or max throws reached
int simulateGame(const std::vector<int>& columns, std::vector<int>& column_advancements) {
    int throws = 0, num_twos_used = 0, num_threes_used = 0, num_elevens_used = 0;
    while (true) {
        ++throws;
        auto roll = rollDice();

        if (num_twos_used >= 3 || num_threes_used >= 5 || num_elevens_used >= 5) {
            return throws; // Bust if 2 has been used 3 or more times or columns 3/11 bust condition is met
        }

        if (!canAdvance(roll, columns, num_twos_used, num_threes_used, num_elevens_used, column_advancements)) {
            return throws; // Busts
        }

        if (throws == 6) { // Limit to 6 throws for this simulation
            return -1; // If 6 throws reached without busting, return -1
        }
    }
}
```

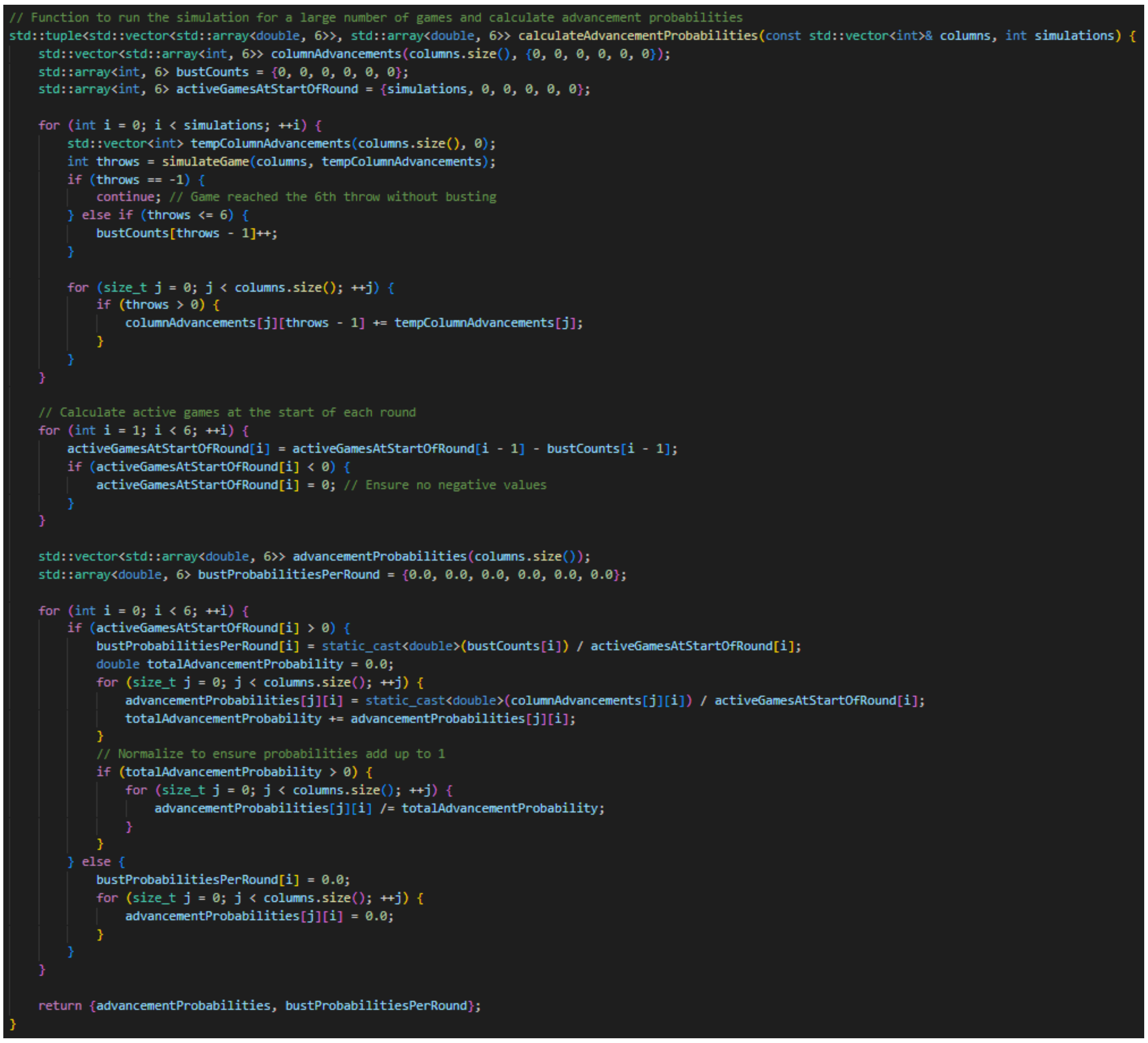
Function to run the simulation for a large number of games and calculate busting probabilities.

This function runs the simulation for a specified number of games, and calculates the probabilities of busting and advancing for each throw.

Approach:

- The function initialises arrays to count busts and track active games at the start of each round.
- Runs the simulation for the specified number of games, updating the bust counts and column advancements accordingly.
- Calculates the active games at the start of each round (an active game is a game where the player has not busted).
- It calculates the number of active games at the start of each round.
- Computes the bust probabilities and advancement probabilities for each round.
- Normalises the advancement probabilities to ensure they add up to 1.

The code for this can be observed below.



```cpp
// Function to run the simulation for a large number of games and calculate advancement probabilities
std::tuple<std::vector<std::array<double, 6>>, std::array<double, 6>> calculateAdvancementProbabilities(const std::vector<int>& columns, int simulations) {
    std::vector<std::array<int, 6>> columnAdvancements(columns.size(), {0, 0, 0, 0, 0, 0});
    std::array<int, 6> bustCounts = {0, 0, 0, 0, 0, 0};
    std::array<int, 6> activeGamesAtStartOfRound = {simulations, 0, 0, 0, 0, 0};

    for (int i = 0; i < simulations; ++i) {
        std::vector<int> tempColumnAdvancements(columns.size(), 0);
        int throws = simulateGame(columns, tempColumnAdvancements);
        if (throws == -1) {
            continue; // Game reached the 6th throw without busting
        } else if (throws <= 6) {
            bustCounts[throws - 1]++;
        }

        for (size_t j = 0; j < columns.size(); ++j) {
            if (throws > 0) {
                columnAdvancements[j][throws - 1] += tempColumnAdvancements[j];
            }
        }
    }

    // Calculate active games at the start of each round
    for (int i = 1; i < 6; ++i) {
        activeGamesAtStartOfRound[i] = activeGamesAtStartOfRound[i - 1] - bustCounts[i - 1];
        if (activeGamesAtStartOfRound[i] < 0) {
            activeGamesAtStartOfRound[i] = 0; // Ensure no negative values
        }
    }

    std::vector<std::array<double, 6>> advancementProbabilities(columns.size());
    std::array<double, 6> bustProbabilitiesPerRound = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0};

    for (int i = 0; i < 6; ++i) {
        if (activeGamesAtStartOfRound[i] > 0) {
            bustProbabilitiesPerRound[i] = static_cast<double>(bustCounts[i]) / activeGamesAtStartOfRound[i];
            double totalAdvancementProbability = 0.0;
            for (size_t j = 0; j < columns.size(); ++j) {
                advancementProbabilities[j][i] = static_cast<double>(columnAdvancements[j][i]) / activeGamesAtStartOfRound[i];
                totalAdvancementProbability += advancementProbabilities[j][i];
            }
            // Normalize to ensure probabilities add up to 1
            if (totalAdvancementProbability > 0) {
                for (size_t j = 0; j < columns.size(); ++j) {
                    advancementProbabilities[j][i] /= totalAdvancementProbability;
                }
            }
        } else {
            bustProbabilitiesPerRound[i] = 0.0;
            for (size_t j = 0; j < columns.size(); ++j) {
                advancementProbabilities[j][i] = 0.0;
            }
        }
    }

    return {advancementProbabilities, bustProbabilitiesPerRound};
}
```

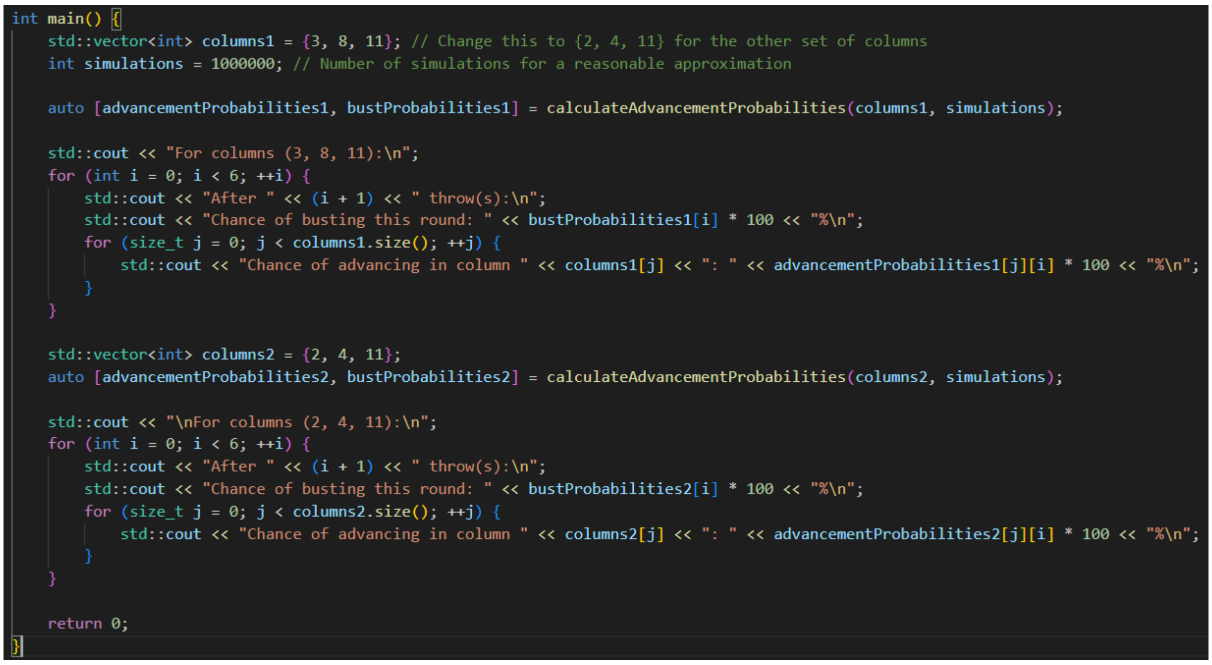
Main function to run the simulation and output the results

The main function sets up and runs the simulation, then outputs the results.

Approach:

- It defines the columns to be tested, ‘columns1’ and ‘columns2’, being the columns which were defined previously in part A.
- It runs the calculateAdvancementProbabilities function for each set of columns
- It prints the bust probabilities and advancement probabilities for each throw.

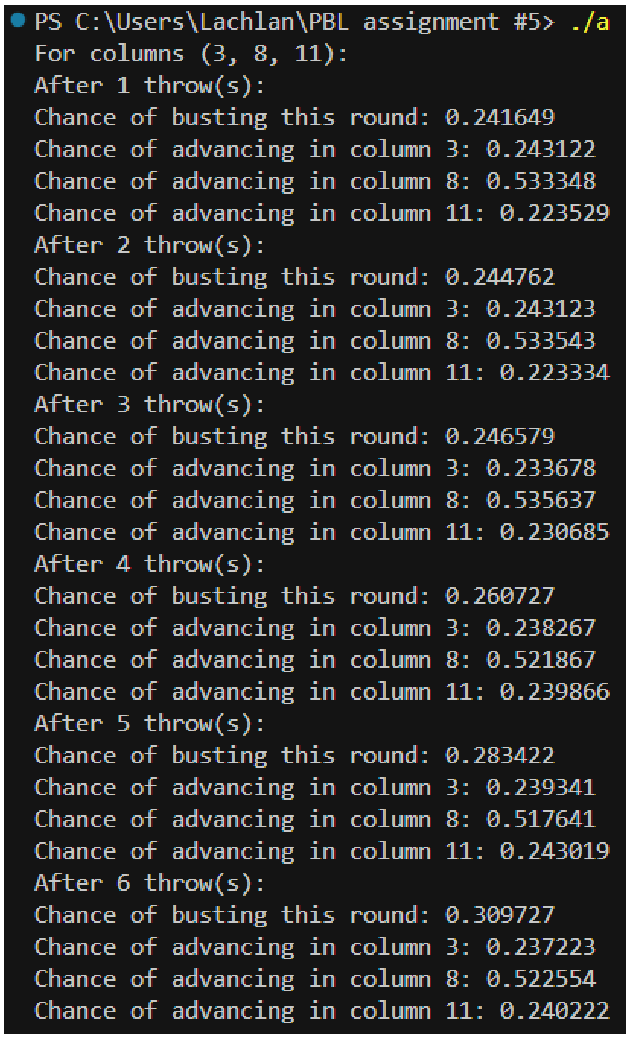
The code for the main function can be observed below:



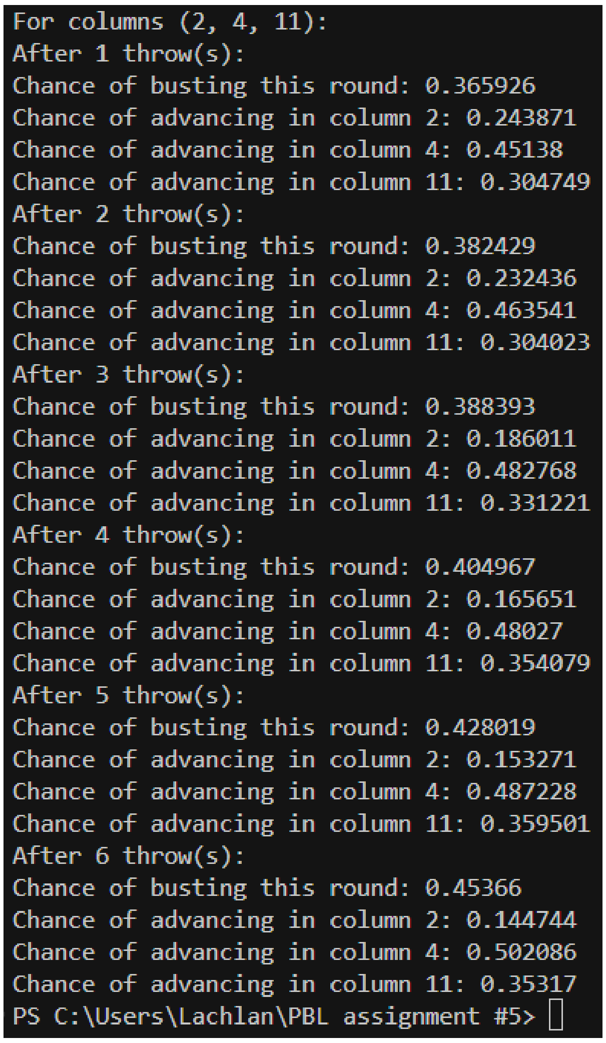
```cpp
int main() {
    std::vector<int> columns1 = {3, 8, 11}; // Change this to {2, 4, 11} for the other set of columns
    int simulations = 1000000; // Number of simulations for a reasonable approximation

    auto [advancementProbabilities1, bustProbabilities1] = calculateAdvancementProbabilities(columns1, simulations);

    std::cout << "For columns (3, 8, 11):\n";
    for (int i = 0; i < 6; ++i) {
        std::cout << "After " << (i + 1) << " throw(s):\n";
        std::cout << "Chance of busting this round: " << bustProbabilities1[i] * 100 << "%\n";
        for (size_t j = 0; j < columns1.size(); ++j) {
            std::cout << "Chance of advancing in column " << columns1[j] << ": " << advancementProbabilities1[j][i] * 100 << "%\n";
        }
    }

    std::vector<int> columns2 = {2, 4, 11};
    auto [advancementProbabilities2, bustProbabilities2] = calculateAdvancementProbabilities(columns2, simulations);

    std::cout << "\nFor columns (2, 4, 11):\n";
    for (int i = 0; i < 6; ++i) {
        std::cout << "After " << (i + 1) << " throw(s):\n";
        std::cout << "Chance of busting this round: " << bustProbabilities2[i] * 100 << "%\n";
        for (size_t j = 0; j < columns2.size(); ++j) {
            std::cout << "Chance of advancing in column " << columns2[j] << ": " << advancementProbabilities2[j][i] * 100 << "%\n";
        }
    }

    return 0;
}
```

Result (i):

Upon running the program, the following output can be observed below.



```
PS C:\Users\Lachlan\PBL assignment #5> ./a
For columns (3, 8, 11):
After 1 throw(s):
Chance of busting this round: 0.241649
Chance of advancing in column 3: 0.243122
Chance of advancing in column 8: 0.533348
Chance of advancing in column 11: 0.223529
After 2 throw(s):
Chance of busting this round: 0.244762
Chance of advancing in column 3: 0.243123
Chance of advancing in column 8: 0.533543
Chance of advancing in column 11: 0.223334
After 3 throw(s):
Chance of busting this round: 0.246579
Chance of advancing in column 3: 0.233678
Chance of advancing in column 8: 0.535637
Chance of advancing in column 11: 0.230685
After 4 throw(s):
Chance of busting this round: 0.260727
Chance of advancing in column 3: 0.238267
Chance of advancing in column 8: 0.521867
Chance of advancing in column 11: 0.239866
After 5 throw(s):
Chance of busting this round: 0.283422
Chance of advancing in column 3: 0.239341
Chance of advancing in column 8: 0.517641
Chance of advancing in column 11: 0.243019
After 6 throw(s):
Chance of busting this round: 0.309727
Chance of advancing in column 3: 0.237223
Chance of advancing in column 8: 0.522554
Chance of advancing in column 11: 0.240222
```



```
For columns (2, 4, 11):
After 1 throw(s):
Chance of busting this round: 0.365926
Chance of advancing in column 2: 0.243871
Chance of advancing in column 4: 0.45138
Chance of advancing in column 11: 0.304749
After 2 throw(s):
Chance of busting this round: 0.382429
Chance of advancing in column 2: 0.232436
Chance of advancing in column 4: 0.463541
Chance of advancing in column 11: 0.304023
After 3 throw(s):
Chance of busting this round: 0.388393
Chance of advancing in column 2: 0.186011
Chance of advancing in column 4: 0.482768
Chance of advancing in column 11: 0.331221
After 4 throw(s):
Chance of busting this round: 0.404967
Chance of advancing in column 2: 0.165651
Chance of advancing in column 4: 0.48027
Chance of advancing in column 11: 0.354079
After 5 throw(s):
Chance of busting this round: 0.428019
Chance of advancing in column 2: 0.153271
Chance of advancing in column 4: 0.487228
Chance of advancing in column 11: 0.359501
After 6 throw(s):
Chance of busting this round: 0.45366
Chance of advancing in column 2: 0.144744
Chance of advancing in column 4: 0.502086
Chance of advancing in column 11: 0.35317
PS C:\Users\Lachlan\PBL assignment #5>
```

Method (ii):

Building upon the bust probabilities calculated in part B (i), we can now determine the optimal number of throws for each position within the column using expected value calculations. Expected value considers both the risk of busting (based on probabilities) and the potential reward of reaching higher positions faster. In this section, we will calculate the expected progress a player can achieve for the current turn if they were to roll one more time. This calculation is based on the formula presented in this paper, being:

$$expected\, progress = p1 * (c + 1) + p2 * (c + 2) + (1 - (p1 + p2)) * 0$$

where:

$p1$ = Probability of making 1 unit progress on any of the 3 columns. Therefore p1 = probability of making 1 unit progress on any of the 3 columns.
$p2$ = Probability of making 2 units progress on any of the 3 columns. Therefore p2 = (p1 * probability of not busting in the next round).
c = total units of progress made on the 3 columns this turn.

p1 * (c + 1): This term represents the expected progress if the player advances by 1 unit on any of the columns after rolling again.
p2 * (c + 2): This term represents the expected progress if the player advances by 2 units on any of the columns after rolling again.
(1 - (p1 + p2)) * 0: This term accounts for the possibility of busting (represented by 1 minus the sum of p1 and p2) and assigns a progress value of 0 in that scenario.

By calculating the expected progress, the player can make informed decisions about whether to roll again. If the current progress, c is already greater than or equal to the expected progress after rolling again, the player should stop rolling to avoid busting. The decision to roll again can be expressed in terms of the boolean result of the inequality: (true meaning roll again, false meaning do not roll again).

$$p1 * (c + 1) + p2 * (c + 2) >= c$$

The expected progress calculations for columns (3, 8, 11) can be observed below. This is calculated using the simulation's probabilities of busting each round.

Throw	p**1**	**Probability of Not Busting Next Round**	p**2**	**Expected Progress**
1	1−0.241649=0.758351	1−0.244762=0.755238	0.758351×0.755238=0.573053	0.758351×1+0.573053×2= 1.904457
2	1−0.244762=0.755238	1−0.246579=0.753421	0.755238×0.753421=0.569168	0.755238×2+0.569168×3= 3.21798

3	1−0.246579=0.753421	1−0.260727=0.739273	0.753421×0.739273=0.556735	0.753421×3+0.556735×4= 4.487203
4	1−0.260727=0.739273	1−0.283422=0.716578	0.739273×0.716578=0.529718	0.739273×4+0.529718×5= 5.605682
5	1−0.283422=0.716578	1−0.309727=0.690273	0.716578×0.690273=0.494521	0.716578×5+0.494521×6= 6.550016

Expected progress calculations for columns (2, 4, 11) can be observed below, the calculations are again achieved by using the simulated probabilities.

Throw	p**1**	**Probability of Not Busting Next Round**	p**2**	**Expected Progress**
1	1−0.365926=0.634074	1−0.382429=0.617571	0.634074×0.617571=0.391815	0.634074×1+0.391815×2= 1.417704
2	1−0.382429=0.617571	1−0.388393=0.611607	0.617571×0.611607=0.377488	0.617571×2+0.377488×3= 2.367105
3	1−0.388393=0.611607	1−0.404967=0.595033	0.611607×0.595033=0.364089	0.611607×3+0.364089×4= 3.096069
4	1−0.404967=0.595033	1−0.428019=0.571981	0.595033×0.571981=0.340533	0.595033×4+0.340533×5= 3.71523
5	1−0.428019=0.571981	1−0.45366=0.54634	0.571981×0.54634=0.312491	0.571981×5+0.312491×6= 4.321396

The table below summarises the decision making process for each throw. We compare the expected progress (calculated above) for each set of columns with the current progress (assumed to be 0 at the start and incremented with each successful roll). If the expected progress is greater than or equal to the current progress, it is beneficial to continue rolling, as explained earlier.

Throw	**Columns (3, 8, 11)** **Expected progress**	**Decision (3, 8, 11)**	**Columns (2, 4, 11)** **Expected progress**	**Decision (2, 4, 11)**
1	1.904457	Yes, continue rolling as expected progress (1.904457) > current progress (0)	1.417704	Yes, continue rolling as expected progress (1.417704) > current progress (0)
2	3.21798	Yes, continue rolling as expected progress (3.21798) > current progress (1)	2.367105	Yes, continue rolling as expected progress (2.367105) > current progress (1)
3	4.487203	Yes, continue rolling as expected progress (4.487203) > current progress (2)	3.096069	Yes, continue rolling as expected progress (3.096069) > current progress (2)
4	5.605682	Yes, continue rolling as expected progress (5.605682) > current progress (3)	3.71523	Yes, continue rolling as expected progress (3.71523) > current progress (3)
5	6.550016	Yes, continue rolling as expected progress (6.550016) > current progress (4)	4.321396	No, do not continue rolling as expected progress (4.321396) < current progress (4)

Solution part B (ii):

Based on the expected progress calculations for both sets of columns (3, 8, 11) and (2, 4, 11), we can determine the optimal decision-making strategy for continuing to roll or not. For columns (3, 8, 11), the expected progress consistently exceeds the current progress for each throw. This indicates that it is advantageous to continue rolling up to at least 5 rolls, as the expected gains outweigh the risk. However, for columns (2, 4, 11), the expected progress remains higher than the current progress until throw 5, where it is advantageous to stop rolling.

Discussion part B (i):

For columns (3, 8, 11), we calculated the chance of busting and the chance of advancing in each column after each throw. The results showed that the chance of busting increased slightly with each throw. For columns (2, 4, 11), the pattern was similar, with the chance of busting increasing as the player progressed with more throws.

The probabilities of advancing appear to be very accurate for the first throw, even though it is a monte carlo simulation, and we did not enumerate all possibilities such as part A. This was indicated by the first rolls percentages deviating from the true percentage (calculated in part A from the first roll) of busting or advancing by only four decimal places. This accuracy is verified by comparing the simulated chance of busting for columns (2, 4, 11) at 0.365926 with the previously calculated probability of 0.36574 . Similarly, for columns (3, 8, 11), the Monte Carlo simulation yields a busting chance of 0.241649, closely aligning with the calculated probability of 0.24151. This close alignment with the true percentages can be attributed to the significant amount of times the game was simulated, as it was simulated one million times.

The probabilities of advancing in each column are calculated for each throw. For columns (3, 8, 11), the chances of advancing in column 8 are consistently the highest, around 53%. This makes sense as column 8 has the most combinations of the three columns, validating the results. It was expected that columns 3 and 11 would see a slight decrease in advancement probabilities after the fifth throw due to the increased likelihood of busts if five consecutive advancements occurred on that specific column. However, this scenario is so unlikely that it did not noticeably affect the results.

For columns (2, 4, 11), the chances of advancing in column 4 are the highest, around 45%, with columns 2 and 11 showing lower advancement probabilities. Column 2's probability drops from 24% to as low as 14.5% as the number of throws increases. This drop is due to the higher probability of a player hitting three 3’s, leading to busts. Therefore, the calculated probabilities of advancing validate the simulation results and confirm their correctness.

The results for different columns and throws allow us to make informed decisions about the risk and reward associated with each roll, which was used in comparing the two sets of columns; we observe that columns (3, 8, 11) present a lower risk of busting compared to columns (2, 4, 11). This makes columns (3, 8, 11) a slightly safer choice for advancing. However, it is noted that if the simulation was to be run again, the results would vary slightly due to the random nature of the simulation.

These results indicate that columns (3, 8, 11) present a lower risk of busting compared to columns (2, 4, 11), making them a safer choice for advancement. However, the inherent randomness of the simulation means results could vary slightly on repeated runs.

Discussion part B (ii):

The expected progress calculations provide strategic insights into the game, helping players decide whether to continue rolling or to stop based on the expected progress compared to the current progress.

For columns (3, 8, 11) the expected progress remains higher than the current progress up to the fifth throw, suggesting that continuing to roll is advantageous for maximising progress. This indicates that players targeting these columns can expect a higher rate of advancement, making it ideal to continue rolling within the first five throws.

For columns (2, 4, 11), the chance of busting is higher, a similar pattern is observed initially. The expected progress exceeds the current progress for the first four throws, suggesting that continuing to roll is generally beneficial. However after the fifth throw, the expected progress drops below the current progress, indicating that it is advantageous to stop rolling beyond this point.

Although the assignment specified investigating up to four rolls, calculations up to five rolls were conducted to demonstrate the point at which it becomes advantageous to stop rolling. This extended analysis allows players to make better informed decisions within the game.

These expected progress calculations provide a clear and unique solution for each throw, aiding the decision-making process of a player. It is important to note that this analysis focuses on maximising the number of columns hit, without considering the value of hitting a column (hitting a 2 is much more valuable to hitting a seven, as you only need to roll two 2's to reach the end of the column).

In real-life gameplay, a player's strategy may be influenced by their opponent's progress, for instance if an opponent is significantly ahead, a player may choose to take more risks to catch up. Thus although expected progress is a good indicator of expecting to advance, players' strategies must be adaptive to the game's context, and the opponent is playing.

By comparing the two sets of columns, columns (3, 8, 11) consistently offer higher expected progress, making it more optimal not necessarily for winning the game, but a better choice for continued rolling. This indicates that a player with columns (3, 8, 11) is likely to hit more columns compared to columns (2, 4, 11).

The assumptions in the expected progress calculations are based on the probabilities of busting and advancing, which was derived from the simulation data, where it was assumed that a player starts each of their cones from the first position on the board. It is noted that there are other valid interpretations of the question asked, however working out the expected progress of a player is a valid interpretation of the question.

In conclusion, the solution is reliable and provides valuable insights for decision-making in the game "Can't Stop". By understanding the expected progress and the associated risks, players can make informed decisions to increase their chances of winning by combining this knowledge based on the game context and opponent's progress.