Tri Dung Nguyen – a1899360

COMP SCI 1010 – Assignment 5

Part 1:

1. When columns are (3, 8, 11):

DESCRIPTION:

- Model:

We need to validate if 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). My approach is to write a simulation program in C++ to confirm these probabilities. The program would loop through all possible values of 4 dice in each turn, and check if we can take any 2 dice out of to make a sum equal to 3, 8 or 11. If we can, we can advance our runners. If not, we bust. After counting how many times we can advance/bust, we can calculate the probability of advancing/busting.

Here is the code: Part1-Q1.cpp

Part1-Q1.cpp

Part1-Q1.cpp > main()

```cpp
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    int pass = 0; // initialize the number of times the player advances
    int bust = 0; // initialize the number of times the player busts

    // nested loops to simulate the 4 dice throws
    for (int throw1 = 1; throw1 <= 6; throw1++) {
        for (int throw2 = 1; throw2 <= 6; throw2++) {
            for (int throw3 = 1; throw3 <= 6; throw3++) {
                for (int throw4 = 1; throw4 <= 6; throw4++) {
                    // calculate the possible sums of 2 out of 4 dice throws
                    int sum1 = throw1 + throw2;
                    int sum2 = throw1 + throw3;
                    int sum3 = throw1 + throw4;
                    int sum4 = throw2 + throw3;
                    int sum5 = throw2 + throw4;
                    int sum6 = throw3 + throw4;

                    // check if the 1st possible sum is 3, 8, or 11
                    // if so, increment the pass counter
                    if (sum1 == 3 || sum1 == 8 || sum1 == 11) {
                        pass++;
                    }

                    // else if the 2nd possible sum is 3, 8, or 11
                    // if so, increment the pass counter
```

Tri Dung Nguyen – a1899360

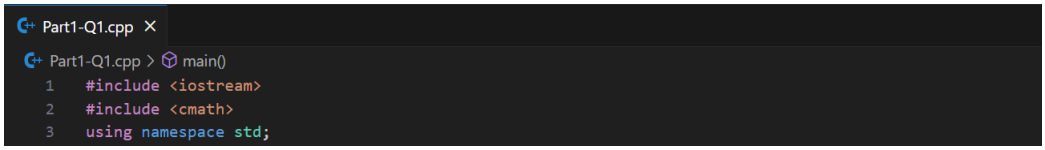
Part1-Q1.cpp

```cpp
int main() {
                    // if so, increment the pass counter
                    else if (sum2 == 3 || sum2 == 8 || sum2 == 11) {
                        pass++;
                    }

                    // else if the 3rd possible sum is 3, 8, or 11
                    // if so, increment the pass counter
                    else if (sum3 == 3 || sum3 == 8 || sum3 == 11) {
                        pass++;
                    }

                    // else if the 4th possible sum is 3, 8, or 11
                    // if so, increment the pass counter
                    else if (sum4 == 3 || sum4 == 8 || sum4 == 11) {
                        pass++;
                    }

                    // else if the 5th possible sum is 3, 8, or 11
                    // if so, increment the pass counter
                    else if (sum5 == 3 || sum5 == 8 || sum5 == 11) {
                        pass++;
                    }

                    // else if the 6th possible sum is 3, 8, or 11
                    // if so, increment the pass counter
                    else if (sum6 == 3 || sum6 == 8 || sum6 == 11) {
                        pass++;
                    }
```

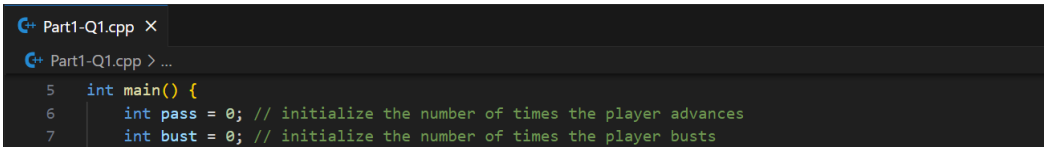
Part1-Q1.cpp

```cpp
int main() {
                    else if (sum5 == 3 || sum5 == 8 || sum5 == 11) {
                        pass++;
                    }

                    // else if the 6th possible sum is 3, 8, or 11
                    // if so, increment the pass counter
                    else if (sum6 == 3 || sum6 == 8 || sum6 == 11) {
                        pass++;
                    }

                    // else, it means that there is no possible sum equal to 3, 8, or 11
                    // hence, increment the bust counter
                    else {
                        bust++;
                    }
                }
            }
        }
    }

    // calculate the probability of advancing and busting
    double pass_prob = pass / pow(6, 4);
    double bust_prob = bust / pow(6, 4);

    // output the probabilities
    cout << "The probability of advancing is: " << pass_prob << endl;
    cout << "The probability of busting is: " << bust_prob << endl;
}
```

- Explanation:

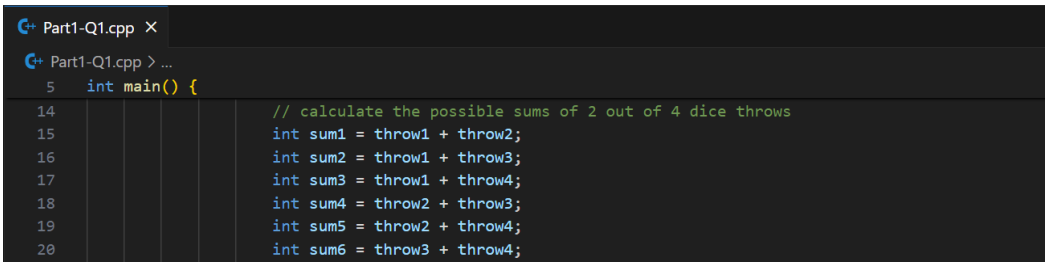


These lines of code include necessary libraries that allow us to use functions and operations in the program. Using namespace std tells the compiler that we can use entities such as cout, endl without needing to prefix them with std::.

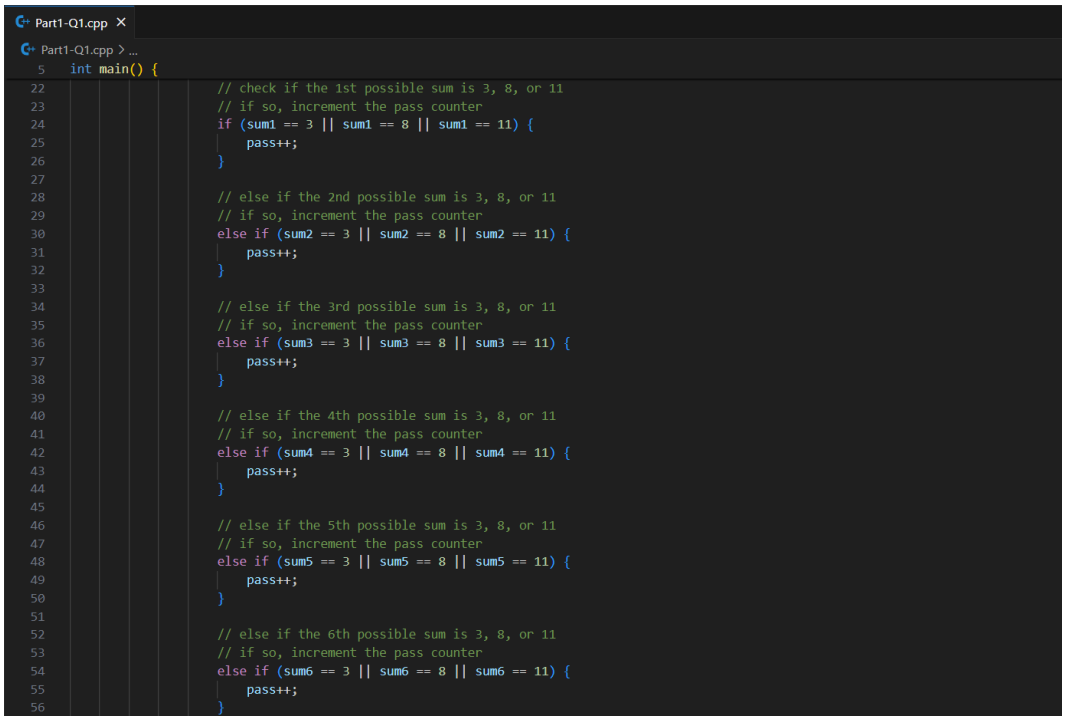


Starting the main function, we initialize the 2 variables: *pass* and *bust* of integer (int) data types and set them to 0. Variable *pass* represents the number of times we can advance our runners. Variable *bust* represents the number of times we cannot advance our runners anymore (bust).

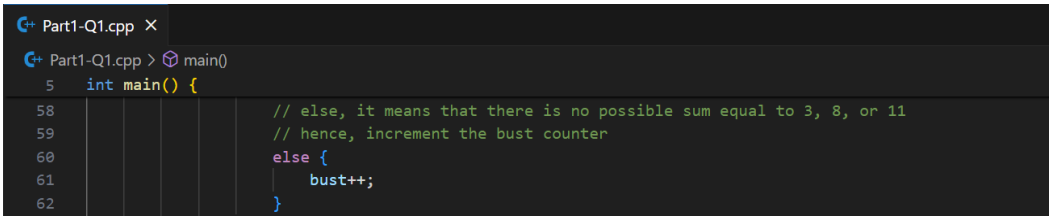
This nested loop will check all possible combinations of 4 dice (throw1, throw2, throw3, throw4), with each one can take integer values from 1 to 6 (inclusively).



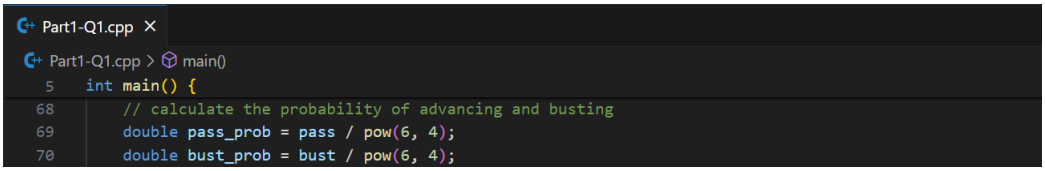
Here, we create 6 different sums: sum1, sum2, sum3, sum4, sum5, sum6, which are the sums of any 2 out of 4 dice we throw in each turn. We need 6 different sums because there are $\binom{4}{2} = \frac{4!}{2!(4-2)!} = \frac{24}{2\times2} = 6$ different combinations of pairs out of 4 dice.



These if-else statements will check if the 6 different sums are equal to 3, 8 or 11. If one of these is true, it means that we can advance our runners this turn, and we increment the variable *pass* (by 1).



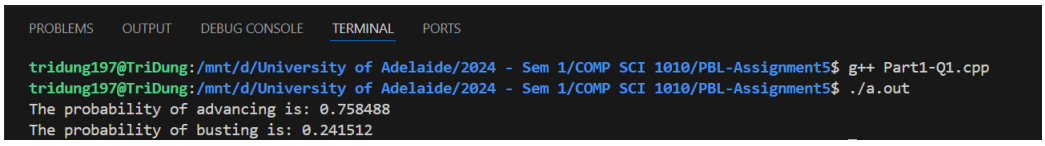
If none of these statements is true, it means that we cannot advance our runners this turn, and we increment the variable *bust* (by 1).



After getting the number of times we can advance/bust (in variable *pass/bust*) by checking all possible values of 4 dice in each turn, we calculate the probability of advancing/busting by dividing the number of times we can advance/bust (*pass/bust*) by the total number of times we can throw dice, which is $6^4 = 1296$ (represented as pow(6, 4) is C++).

The final step is to display the chance of advancing/busting to the terminal.

RESULT:



Compiling the program (the first 2 lines are commands to compile), we obtain the results: the probability of advancing is 0.758488, and the probability of busting is 0.241512. These results match the probabilities needed to validate, which means that the given statement is true.

2. When columns are (2, 4, 11):

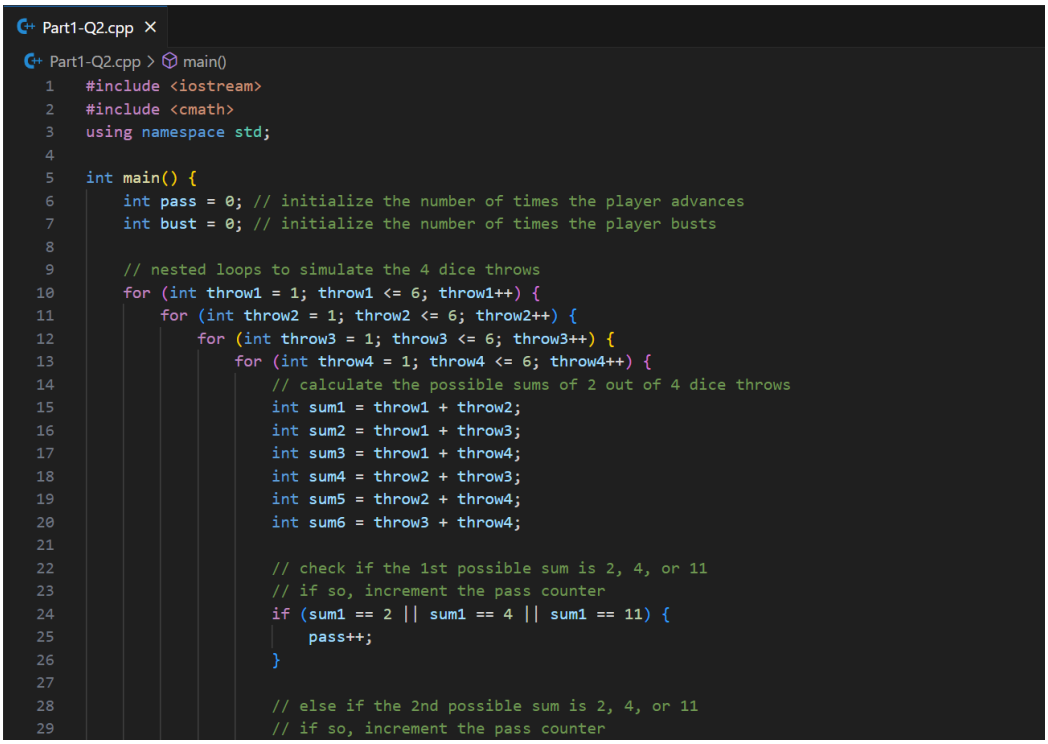
DESCRIPTION:

- Model:

We need to validate if when columns are (2, 4, 11), the chance of advancing is 63% and the chance of busting is 37% (or 0.36574 to be exact). I use the same

approach as when columns are (3, 8, 11). We still loop through all possible values of 4 dice in each turn, then count the number of times we can advance/bust and calculate the probabilities.

Here is the code: Part1-Q2.cpp



```cpp
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    int pass = 0; // initialize the number of times the player advances
    int bust = 0; // initialize the number of times the player busts

    // nested loops to simulate the 4 dice throws
    for (int throw1 = 1; throw1 <= 6; throw1++) {
        for (int throw2 = 1; throw2 <= 6; throw2++) {
            for (int throw3 = 1; throw3 <= 6; throw3++) {
                for (int throw4 = 1; throw4 <= 6; throw4++) {
                    // calculate the possible sums of 2 out of 4 dice throws
                    int sum1 = throw1 + throw2;
                    int sum2 = throw1 + throw3;
                    int sum3 = throw1 + throw4;
                    int sum4 = throw2 + throw3;
                    int sum5 = throw2 + throw4;
                    int sum6 = throw3 + throw4;

                    // check if the 1st possible sum is 2, 4, or 11
                    // if so, increment the pass counter
                    if (sum1 == 2 || sum1 == 4 || sum1 == 11) {
                        pass++;
                    }

                    // else if the 2nd possible sum is 2, 4, or 11
                    // if so, increment the pass counter
```

Tri Dung Nguyen – a1899360

Part1-Q2.cpp

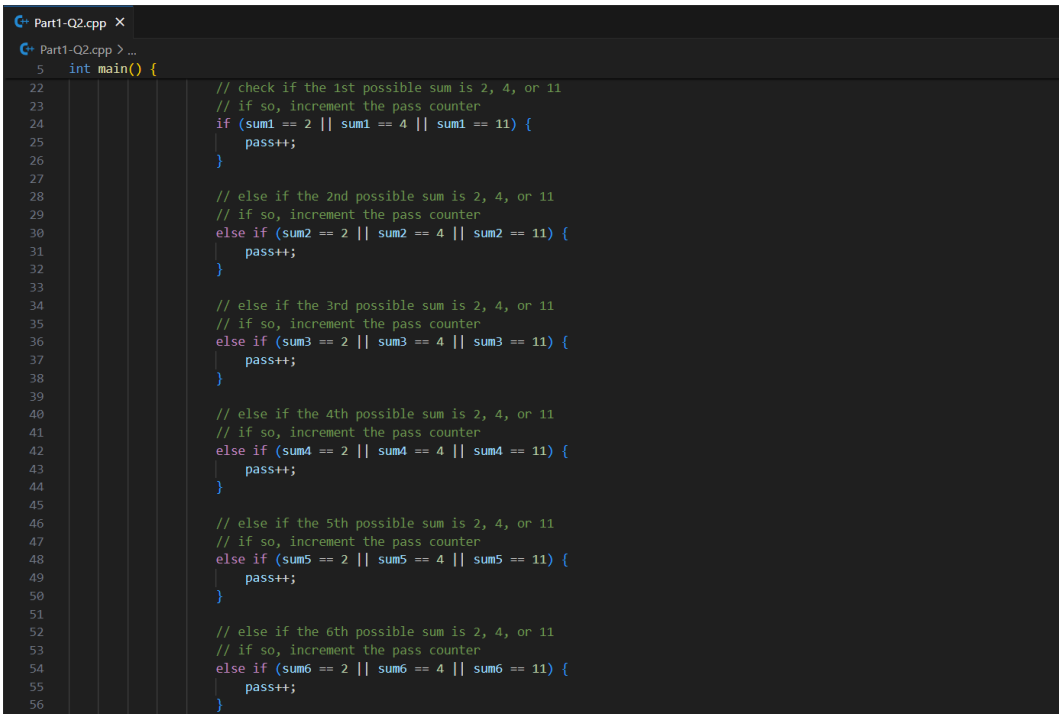
```cpp
int main() {
                    // if so, increment the pass counter
                    else if (sum2 == 2 || sum2 == 4 || sum2 == 11) {
                        pass++;
                    }

                    // else if the 3rd possible sum is 2, 4, or 11
                    // if so, increment the pass counter
                    else if (sum3 == 2 || sum3 == 4 || sum3 == 11) {
                        pass++;
                    }

                    // else if the 4th possible sum is 2, 4, or 11
                    // if so, increment the pass counter
                    else if (sum4 == 2 || sum4 == 4 || sum4 == 11) {
                        pass++;
                    }

                    // else if the 5th possible sum is 2, 4, or 11
                    // if so, increment the pass counter
                    else if (sum5 == 2 || sum5 == 4 || sum5 == 11) {
                        pass++;
                    }

                    // else if the 6th possible sum is 2, 4, or 11
                    // if so, increment the pass counter
                    else if (sum6 == 2 || sum6 == 4 || sum6 == 11) {
                        pass++;
                    }
```

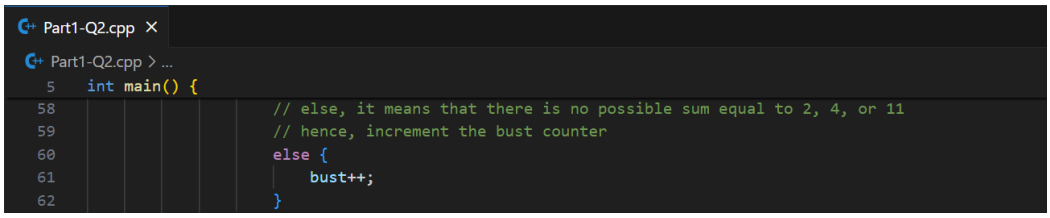
Part1-Q2.cpp

```cpp
int main() {
                    else if (sum5 == 2 || sum5 == 4 || sum5 == 11) {
                        pass++;
                    }

                    // else if the 6th possible sum is 2, 4, or 11
                    // if so, increment the pass counter
                    else if (sum6 == 2 || sum6 == 4 || sum6 == 11) {
                        pass++;
                    }

                    // else, it means that there is no possible sum equal to 2, 4, or 11
                    // hence, increment the bust counter
                    else {
                        bust++;
                    }
                }
            }
        }
    }

    // calculate the probability of advancing and busting
    double pass_prob = pass / pow(6, 4);
    double bust_prob = bust / pow(6, 4);

    // output the probabilities
    cout << "The probability of advancing is: " << pass_prob << endl;
    cout << "The probability of busting is: " << bust_prob << endl;
}
```

- Explanation:

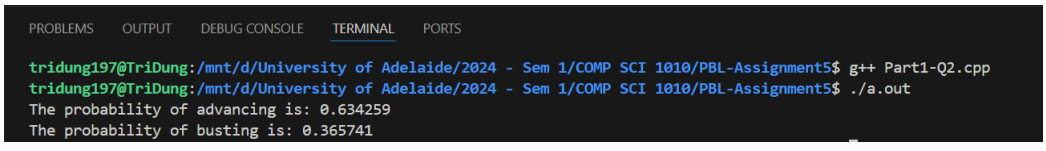
The code is almost similar to the one when columns are (3, 8, 11). The only difference is in the if-else statements that we check if we can choose 2 out of 4 dice that make sum equal to 2, 4 or 11. Here is the if-else statements when columns are (2, 4, 11):





Again, if none of the if-else statements is true, it means that we cannot advance our runners this turn, and we increment the variable *bust* (by 1).

RESULT:



Compiling the program (the first 2 lines are commands to compile), we obtain the results: the probability of advancing is 0.634259, and the probability of

busting is 0.365741. These results match the probabilities needed to validate, which means that the given statement is true.

Part 2:

1. When columns are (3, 8, 11):

DESCRIPTION:

- Model:

Our goal is to find the ideal number of throws in our turns. We need to calculate the probability of busting after 1, 2, 3 or 4 throw turns. For this part, I still use C++ to simulate the game for the given starting position: (3, 8, 11). Before going to the code, we need to define 2 things explicitly:

- The ideal number of throws is the maximum number of throws that retains the successful chances ≥ 50% (or the failed chances ≤ 50%) after a large number of simulated games (in my code, I run the games 100,000 times).
- We only calculate the chance of busting after having all 3 white runners on the board. It means that, before we have 3 white runners on the board, we already play a certain number of throws.

Here is the code: Part2-Q1.cpp

Tri Dung Nguyen – a1899360

Part2-Q1.cpp

```cpp
#include <iostream>
#include <ctime>
using namespace std;

int main() {
    srand(time(0)); // generate random numbers differently each time the program runs
    int passes[4] = {0}; // init array passes with 4 elements to store the number of passes before bust
    int total_games = 100000; // set the total number of games to be 100000

    // run the simulation 100000 times
    for (int i = 0; i < 100000; i++) {
        int pass_temp = 0; // (re)set the number of passes before busting
        bool bust = false; // set the bust flag to false when start each game

        // keep throwing the dice until the player busts
        while (bust == false) {
            // generate random numbers between 1 and 6 for the 4 dice throws
            int throw1 = rand() % 6 + 1;
            int throw2 = rand() % 6 + 1;
            int throw3 = rand() % 6 + 1;
            int throw4 = rand() % 6 + 1;

            pass_temp++; // increment the number of throw turns

            // calculate the possible sums of 2 out of 4 dice throws
            int sum1 = throw1 + throw2;
            int sum2 = throw1 + throw3;
            int sum3 = throw1 + throw4;
            int sum4 = throw2 + throw3;
```

Part2-Q1.cpp

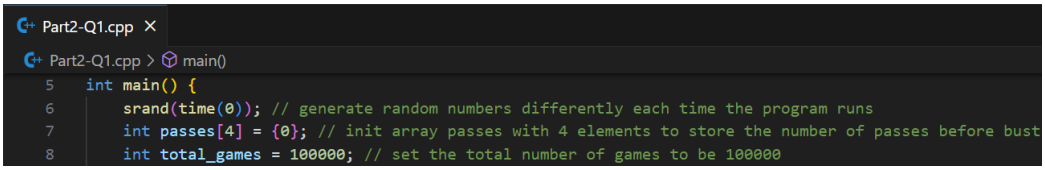
```cpp
int main() {
            int sum1 = throw1 + throw2;
            int sum2 = throw1 + throw3;
            int sum3 = throw1 + throw4;
            int sum4 = throw2 + throw3;
            int sum5 = throw2 + throw4;
            int sum6 = throw3 + throw4;

            // check if none of 6 different sums is 3, 8, or 11
            // if so, set the bust flag to true
            if (sum1 != 3 && sum1 != 8 && sum1 != 11) {
                if (sum2 != 3 && sum2 != 8 && sum2 != 11) {
                    if (sum3 != 3 && sum3 != 8 && sum3 != 11) {
                        if (sum4 != 3 && sum4 != 8 && sum4 != 11) {
                            if (sum5 != 3 && sum5 != 8 && sum5 != 11) {
                                if (sum6 != 3 && sum6 != 8 && sum6 != 11) {
                                    bust = true;
                                }
                            }
                        }
                    }
                }
            }
        }

        // increment the number of passes before busting after 1, 2, 3 or 4 throws
        if (pass_temp <= 4) {
            passes[pass_temp - 1]++;
        }
```

```
Part2-Q1.cpp
Part2-Q1.cpp > main()
5   int main() {
40                                  if (sum6 != 3 && sum6 != 8 && sum6 != 11) {
41                                      bust = true;
42                                  }
43                              }
44                          }
45                      }
46                  }
47              }
48          }
49
50          // increment the number of passes before busting after 1, 2, 3 or 4 throws
51          if (pass_temp <= 4) {
52              passes[pass_temp - 1]++;
53          }
54      }
55
56      // calculate the probability of busting after 1, 2, 3 or 4 throws
57      double bust_prob1 = double(passes[0]) / total_games;
58      double bust_prob2 = bust_prob1 + double(passes[1]) / total_games;
59      double bust_prob3 = bust_prob2 + double(passes[2]) / total_games;
60      double bust_prob4 = bust_prob3 + double(passes[3]) / total_games;
61
62      // output the probabilities
63      cout << "Busting after 1 throw(s): " << bust_prob1 << endl;
64      cout << "Busting after 2 throw(s): " << bust_prob2 << endl;
65      cout << "Busting after 3 throw(s): " << bust_prob3 << endl;
66      cout << "Busting after 4 throw(s): " << bust_prob4 << endl;
67  }
```

- Explanation:

```
Part2-Q1.cpp
Part2-Q1.cpp > main()
1   #include <iostream>
2   #include <ctime>
3   using namespace std;
```

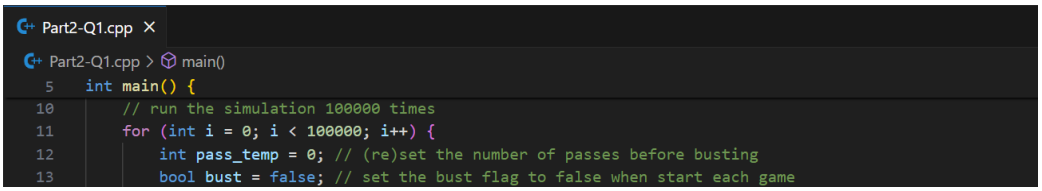
These lines of code include necessary libraries that allow us to use functions and operations in the program. Using namespace std tells the compiler that we can use entities such as cout, endl without needing to prefix them with std::.



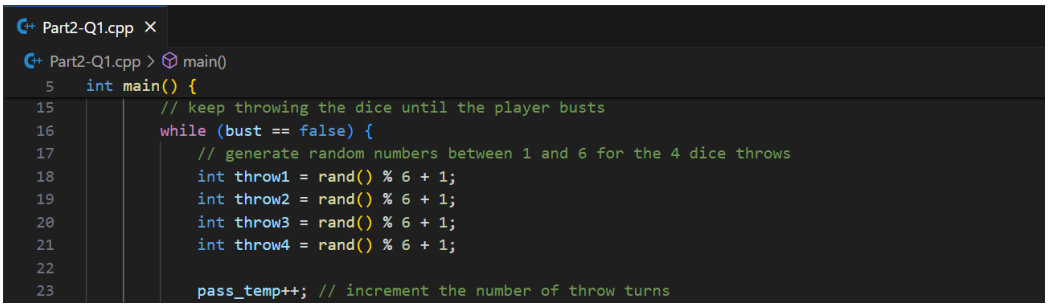
```
Part2-Q1.cpp
Part2-Q1.cpp > main()
5   int main() {
6       srand(time(0)); // generate random numbers differently each time the program runs
7       int passes[4] = {0}; // init array passes with 4 elements to store the number of passes before bust
8       int total_games = 100000; // set the total number of games to be 100000
```

Then, we initialize array *passes* with 4 elements, which stores the number of throws before busting in each game and set variable *total_games* to be

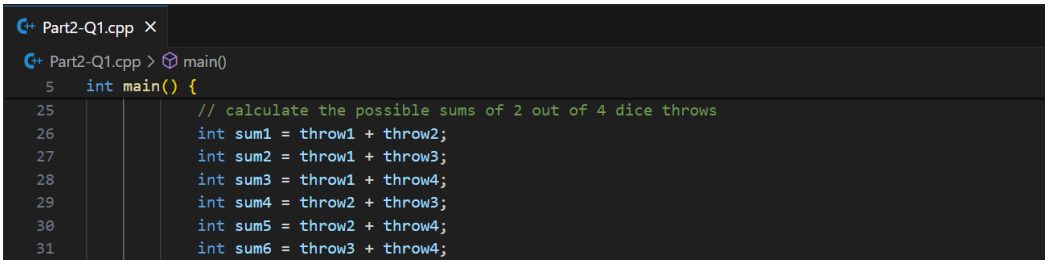
100000 (we run simulated games 100000 times). Note that, srand(time(0)) is used to generate random numbers differently each time the program runs.



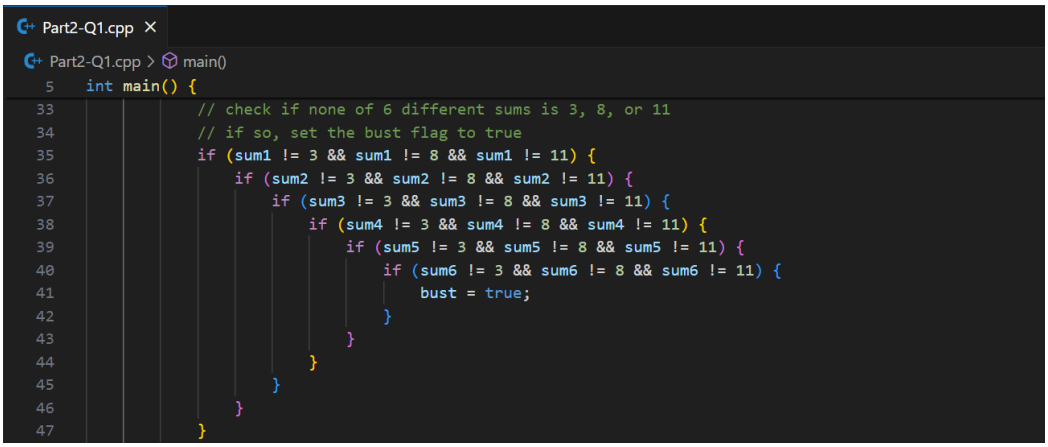
We run the game 100000 times. At the start of each game, (re)set the variable *pass_temp* to 0, which counts the number of throws before busting. Also, we create variable *bust* of bool data type, which is used to check if we bust or not.



For each game, we throw the dice until the bust happens. Each turn, we use rand() % 6 + 1 to generate an integer between 1 and 6 (inclusively) to represent the value of each dice. Note that, we also need to increment the number of throws (*pass_temp)*.

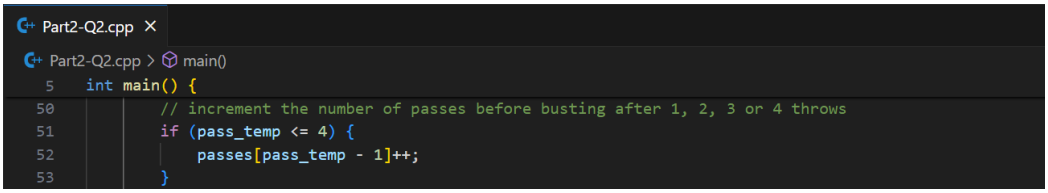


After having the value of each dice, we then calculate the 6 possible sums of any 2 out of 4 dice. We have 6 different sums because there are $\binom{4}{2} = \frac{4!}{2!(4-2)!} = \frac{24}{2\times 2} = 6$ different combinations of pairs out of 4 dice.



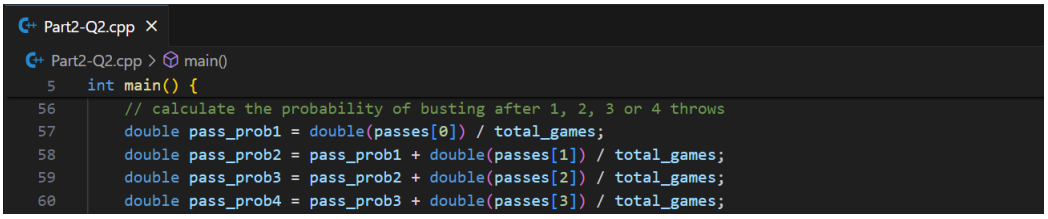
```
Part2-Q1.cpp
Part2-Q1.cpp > main()
5   int main() {
33              // check if none of 6 different sums is 3, 8, or 11
34              // if so, set the bust flag to true
35              if (sum1 != 3 && sum1 != 8 && sum1 != 11) {
36                  if (sum2 != 3 && sum2 != 8 && sum2 != 11) {
37                      if (sum3 != 3 && sum3 != 8 && sum3 != 11) {
38                          if (sum4 != 3 && sum4 != 8 && sum4 != 11) {
39                              if (sum5 != 3 && sum5 != 8 && sum5 != 11) {
40                                  if (sum6 != 3 && sum6 != 8 && sum6 != 11) {
41                                      bust = true;
42                                  }
43                              }
44                          }
45                      }
46                  }
47              }
```

Then, we check if none of 6 different sums above is equal to 3, 8 or 11. If that is true, we set variable *bust* to true, which means that we stop the current game. If not, we can keep playing the game.



```
Part2-Q2.cpp
Part2-Q2.cpp > main()
5   int main() {
50          // increment the number of passes before busting after 1, 2, 3 or 4 throws
51          if (pass_temp <= 4) {
52              passes[pass_temp - 1]++;
53          }
```

Once the game is stopped, we will check how many throw turns in that game. If the number is ≤ 4, increment the corresponding element in array *passes*.



```
Part2-Q2.cpp
Part2-Q2.cpp > main()
5   int main() {
56      // calculate the probability of busting after 1, 2, 3 or 4 throws
57      double pass_prob1 = double(passes[0]) / total_games;
58      double pass_prob2 = pass_prob1 + double(passes[1]) / total_games;
59      double pass_prob3 = pass_prob2 + double(passes[2]) / total_games;
60      double pass_prob4 = pass_prob3 + double(passes[3]) / total_games;
```

Once all the games are simulated, now we calculate the probability of busting after 1, 2, 3, 4 throw turns. Note that, this probability is cumulative, which means that:

$$\begin{aligned} &The\ probability\ of\ busting\ after\ n\ throws \\ &\quad = The\ probabiltity\ of\ busting\ after\ (n-1)\ throws \\ &\quad + The\ probability\ of\ busting\ at\ EXACTLY\ n\ throws \end{aligned}$$

$$(for\ n \geq 2)$$

```
Part2-Q2.cpp ×
Part2-Q2.cpp > main()
 5  int main() {
62      // output the probabilities
63      cout << "Busting after 1 throw(s): " << pass_prob1 << endl;
64      cout << "Busting after 2 throw(s): " << pass_prob2 << endl;
65      cout << "Busting after 3 throw(s): " << pass_prob3 << endl;
66      cout << "Busting after 4 throw(s): " << pass_prob4 << endl;
67  }
```

The final step is to output the results to the terminal.

RESULT:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

tridung197@TriDung:/mnt/d/University of Adelaide/2024 - Sem 1/COMP SCI 1010/PBL-Assignment5$ g++ Part2-Q1.cpp
tridung197@TriDung:/mnt/d/University of Adelaide/2024 - Sem 1/COMP SCI 1010/PBL-Assignment5$ ./a.out
Busting after 1 throw(s): 0.24301
Busting after 2 throw(s): 0.42586
Busting after 3 throw(s): 0.56503
Busting after 4 throw(s): 0.66953
```

Compiling the program (the first 2 lines are commands to compile), it shows the chance of busting after 1, 2, 3 or 4 throws are: 0.24301, 0.42586, 0.56503, 0.66953, respectively. Note that these probabilities may vary slightly for each time we run the program, because these are calculated based on simulation, not by solid mathematical formula.

As defined, the ideal number of throws is the maximum number that remains the probability of busting ≤ 50% (or 0.5). Therefore, when columns are (3, 8, 11), the ideal number of throws is 2.

2. When columns are (2, 4, 11):

DESCRIPTION:

- Model:

We need to find the ideal number of throws in our turns. The approach used here is the same as the one above when columns are (3, 8, 11). Again, we define 2 things explicitly:

- The ideal number of throws is the maximum number of throws that retains the successful chances ≥ 50% (or the failed chances ≤ 50%)

after a large number of simulated games (in my code, I run the games 100,000 times).

- We only calculate the chance of busting after having all 3 white runners on the board. It means that, before we have 3 white runners on the board, we already play a certain number of throws.

Here is the code: Part2-Q2.cpp

Part2-Q2.cpp

```cpp
#include <iostream>
#include <ctime>
using namespace std;

int main() {
    srand(time(0)); // generate random numbers differently each time the program runs
    int passes[4] = {0}; // init array passes with 4 elements to store the number of passes before bust
    int total_games = 100000; // set the total number of games to be 100000

    // run the simulation 100000 times
    for (int i = 0; i < 100000; i++) {
        int pass_temp = 0; // (re)set the number of passes before busting
        bool bust = false; // set the bust flag to false when start each game

        // keep throwing the dice until the player busts
        while (bust == false) {
            // generate random numbers between 1 and 6 for the 4 dice throws
            int throw1 = rand() % 6 + 1;
            int throw2 = rand() % 6 + 1;
            int throw3 = rand() % 6 + 1;
            int throw4 = rand() % 6 + 1;

            pass_temp++; // increment the number of throw turns

            // calculate the possible sums of 2 out of 4 dice throws
            int sum1 = throw1 + throw2;
            int sum2 = throw1 + throw3;
            int sum3 = throw1 + throw4;
            int sum4 = throw2 + throw3;
```

Tri Dung Nguyen – a1899360

Part2-Q2.cpp

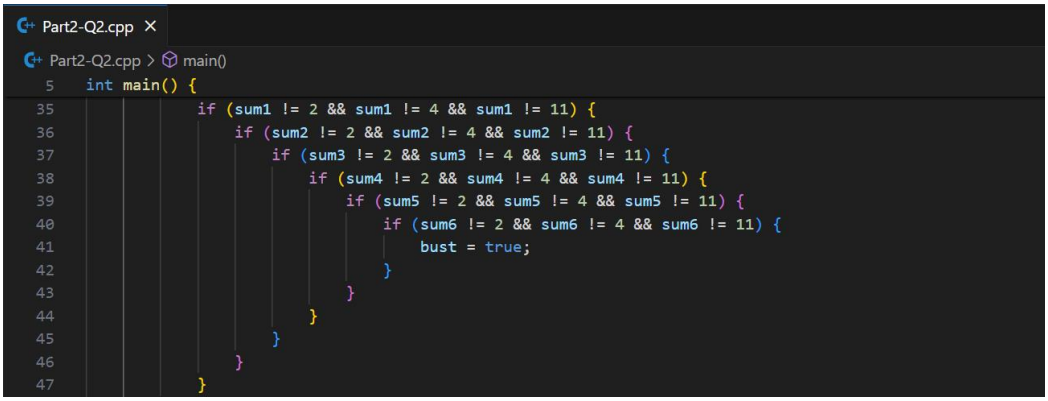
```cpp
int main() {
            int sum1 = throw1 + throw2;
            int sum2 = throw1 + throw3;
            int sum3 = throw1 + throw4;
            int sum4 = throw2 + throw3;
            int sum5 = throw2 + throw4;
            int sum6 = throw3 + throw4;

            // check if none of 6 different sums is 2, 4, or 11
            // if so, set the bust flag to true
            if (sum1 != 2 && sum1 != 4 && sum1 != 11) {
                if (sum2 != 2 && sum2 != 4 && sum2 != 11) {
                    if (sum3 != 2 && sum3 != 4 && sum3 != 11) {
                        if (sum4 != 2 && sum4 != 4 && sum4 != 11) {
                            if (sum5 != 2 && sum5 != 4 && sum5 != 11) {
                                if (sum6 != 2 && sum6 != 4 && sum6 != 11) {
                                    bust = true;
                                }
                            }
                        }
                    }
                }
            }

            // increment the number of passes before busting after 1, 2, 3 or 4 throws
            if (pass_temp <= 4) {
                passes[pass_temp - 1]++;
            }
```

Part2-Q2.cpp

```cpp
int main() {
                                if (sum6 != 2 && sum6 != 4 && sum6 != 11) {
                                    bust = true;
                                }
                            }
                        }
                    }
                }
            }

            // increment the number of passes before busting after 1, 2, 3 or 4 throws
            if (pass_temp <= 4) {
                passes[pass_temp - 1]++;
            }
        }

        // calculate the probability of busting after 1, 2, 3 or 4 throws
        double pass_prob1 = double(passes[0]) / total_games;
        double pass_prob2 = pass_prob1 + double(passes[1]) / total_games;
        double pass_prob3 = pass_prob2 + double(passes[2]) / total_games;
        double pass_prob4 = pass_prob3 + double(passes[3]) / total_games;

        // output the probabilities
        cout << "Busting after 1 throw(s): " << pass_prob1 << endl;
        cout << "Busting after 2 throw(s): " << pass_prob2 << endl;
        cout << "Busting after 3 throw(s): " << pass_prob3 << endl;
        cout << "Busting after 4 throw(s): " << pass_prob4 << endl;
}
```

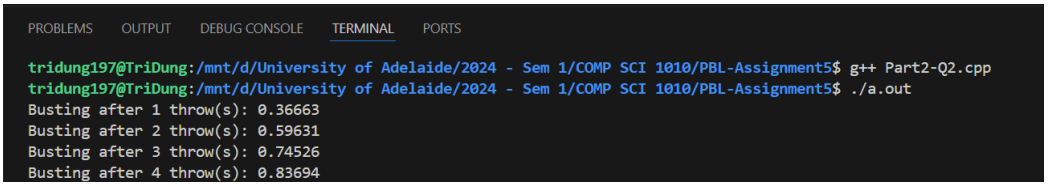
- Explanation:

The code is almost similar to the one when columns are (3, 8, 11). The only difference is in the if statements that we check if none of 6 different sums is equal to 2, 4 or 11. Here is the if statements when columns are (2, 4, 11):



Again, if all the if statements are true, it means that we cannot advance our runners this turn, and we set the variable *bust* to true (stop the current game).

RESULT:



Compiling the program (the first 2 lines are commands to compile), it shows the chance of busting after 1, 2, 3 or 4 throws are: 0.36663, 0.59631, 0.74526, 0.83694, respectively. Note that these probabilities may vary slightly for each time we run the program, because these are calculated based on simulation, not by solid mathematical formula.

As defined, the ideal number of throws is the maximum number that remains the probability of busting ≤ 50% (or 0.5). Therefore, when columns are (2, 4, 11), the ideal number of throws is 1.

Discussion:

- The definition of “ideal” depends on each individual. For example, when columns are (3, 8, 11), one may assume that the ideal number of throws should be 3, and that is still reasonable because its probability is about 0.56, which is quite near our threshold of 0.5.

- In actual games, in some cases, insistently following the fixed ideal number of throws is not a wise decision. For example, if our opponent is likely to win the game in their next move, we should be riskier (can’t stop) at keep throwing dice, otherwise we lose.

- For special cases like (2, 4, 11), actually we can only roll sums of 2 twice to win this column (we already have all 3 white runners on the board). If we do not stop, and can only roll sum of 2 again, we will bust. Therefore, we should take it into account in the model to improve the accuracy of our calculations, but overall, it does not affect much the precision of our results.

- From 2 to 12, the nearer the middle number (7), the more likely we can make sum of it. That is because there are more possible pairs that can be chosen to make that sum. Therefore, the nearer the middle number, the more cells the corresponding column has. It hence would balance the game.