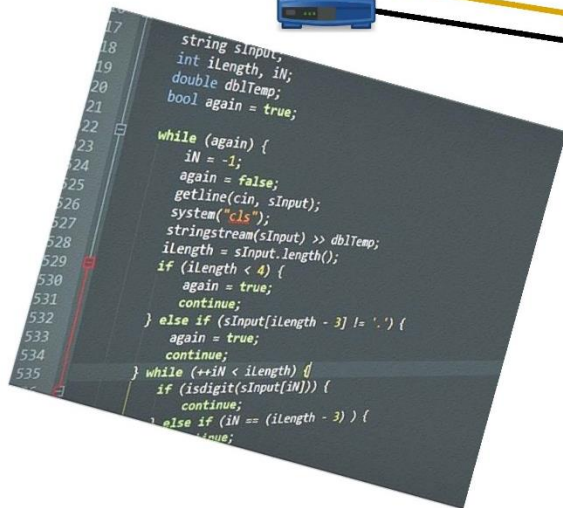


[illegible]

We are pleased that you are considering the study of Computer Science at A level.

More information about the Computer Science specification can be found here: tinyurl.com/RHCCompSciAQA

Ralph Ellis (Course Manager for Computer Science) - ralphe@richuish.ac.uk

Computer Science students will develop an understanding of why computers work in the way that they do as well as learning how to create computer programs using a variety of programming paradigms.

To prepare students to study Computer Science at Richard Huish College, we have included in this booklet some links to videos and online activities as well as some summer homework to complete before you start with us in September. You need to use the examples provided to create programs for three tasks.

The focus of year 1 will be to develop your knowledge and understanding of programming that will include both the theory and practical application. To develop your knowledge, we are going to use the C# programming language.

Some useful weblinks are below.

Visual Studio Download

Download the Community Edition of Visual Studio here. This is the free version.

Getting Started with C#

Get started with some C# programming using Visual Studio. There are lots of tutorials on this site to help learn C# both in console and form based apps.

C# YouTube Tutorials

These tutorials start right at the beginning from installing Visual Studio 2022 through to basic coding concepts such as data types, sequencing, iteration and selection all the way to more advanced object-oriented techniques.

BBC Bitesize GCSE Computer Science (AQA)

Make sure that you are prepared for A Level by reviewing the GCSE content. This will be useful whether you have studied Computer Science already or not.

Is It Still Worth Learning to Code in 2025?

A very interesting discussion point: does the easily accessible availability of AI make the reasons to learn to code obsolete?

Table of Contents

Summer Homework:.....	3
Sequencing	4
Selection.....	5
Definite iteration	6
Indefinite Iteration	7
Randomness	8
Integer Division and Modulus	10
Array (of Strings)	11
Array (of Integers).....	12
Tasks to complete.....	13
Task 1: Dice Cricket.....	13
Task 2: Converting Decimal to Binary	14
Task 3: Bubble Sort.....	14
Extension tasks.....	15

Summer Homework:

You must write three short programs in C# for the following:

- ⇒ Task 1: Dice Cricket
- ⇒ Task 2: Converting Decimal to Binary
- ⇒ Task 3: Bubble Sort

All the code required, with examples, are explained over the next few pages.

If you get stuck, there are videos to show how to code the three tasks here: tinyurl.com/CSharpExamples

The three tasks will utilise a number of key basic programming concepts:

Sequencing – this refers to the specific order in which instructions or statements in a program are executed.

Selection – the use of conditional statements to choose between different paths of execution based on whether a condition is true or false.

Definite iteration – this refers to looping where the number of repetitions is known beforehand, typically using a for loop.

Indefinite iteration – this refers to looping where the number of repetitions is not known in advance. The loop continues until a certain condition is no longer true.

Randomness – this refers to the generation of unpredictable values, often used in simulations, games.

Integer division (DIV) – this is a division operation where the result is the **whole number part only**, and any remainder is discarded. It's often used when you want to know how many times one number fits into another **without fractions**.

- $7 \div 2$ gives 3.5 (normal division)
- $7 \text{ DIV } 2$ gives 3 (integer division)

Modulus (MOD) – the modulus operation gives the **remainder** after integer division. It's useful for checking things like whether a number is even or odd, or for wrapping around values (like in circular lists or clocks).

- $7 \text{ MOD } 2 = 1$ (because 2 goes into 7 three times, with 1 left over)

Arrays – an array is a **collection of values** stored under a single variable name. Each value in the array is called an **element**, and you access them using an **index** (starting from 0 in most languages).

Arrays are useful when you want to store and work with **multiple values of the same type**, like exam scores, names, or temperatures.

The C# code for each of these is explained on the next few pages. You should test these in a Visual Studio console app or an online compiler such as dotnetfiddle.net.

You must copy and paste your code for each of the three tasks onto a Word document, or similar, and bring it with you electronically for the first lesson after the summer holidays. We do not want written or printed solutions.

Sequencing

Each line in C# ends with a semi-colon (;)

The example below shows an example of sequencing that will calculate the area and circumference of a circle.

```
1 int radius;
2 double circumference, area;
3 double pi = 3.14;
4 Console.WriteLine("Enter the radius of the circle:");
5 radius = Convert.ToInt32(Console.ReadLine());
6 circumference = 2 * pi * radius;
7 area = pi * radius * radius;
8 Console.WriteLine("Circumference of the circle: " + circumference);
9 Console.WriteLine("Area of the circle: " + area);
```

What the Code Does

This program:

1. Asks the user to enter the radius of a circle.
2. Calculates the **circumference** and **area** of the circle using that radius.
3. Displays the results.

Line-by-Line Breakdown

1. Declares a variable named radius of type int (integer).
This will store the user's input for the radius of the circle.
2. Declares two variables of type double (used for numbers with a decimal part): circumference and area.
3. Declares and initializes a double variable pi with the value 3.14.
This is an approximation of the mathematical constant π (pi).
4. Displays a message prompting the user to enter the radius.
5. Reads the user's input from the console as a string.
Converts that string to an integer using `Convert.ToInt32()` and stores it in radius.
6. Calculates the **circumference** of the circle using the formula:
$$\text{Circumference} = 2 \times \pi \times \text{radius}$$

Stores the result in the circumference variable.
7. Calculates the **area** of the circle using the formula:
$$\text{Area} = \pi \times \text{radius} \times \text{radius}$$

Stores the result in the area variable.
8. Displays the calculated circumference.
9. Displays the calculated area.

Key Concepts

- **Variables:** Used to store data.
- **Data Types:** int for whole numbers, double for numbers with a decimal part (sometimes called real or float).
- **Input/Output:** `Console.ReadLine()` for input, `Console.WriteLine()` for output.
- **Type Conversion:** `Convert.ToInt32()` changes string input to an integer.
- **Math Operations:** Basic arithmetic to compute values.
- **Sequencing:** Each line runs in order, and later lines depend on earlier ones.

Blocks of code in C# sit between open and closed curly brackets (braces).

Selection

An example of a selection statement using `if` and `else` is below.

```
1 if (radius <=0)
2 {
3     Console.WriteLine("Circumference and Area cannot be calculated.");
4 }
5 else
6 {
7     circumference = 2 * pi * radius;
8     area = pi * radius * radius;
9 }
```

What the Code Does

This code checks if the radius is **less than or equal to zero**. If it is, it displays a message saying the calculations can't be done. Otherwise, it calculates the **circumference** and **area** of a circle.

Line-by-Line Breakdown

1. This is a **conditional statement**.
 - It checks whether the value of radius is **less than or equal to 0**.
 - If this condition is **true**, the code inside the `if` block runs.
 - If it's **false**, the code inside the `else` block runs instead.
2. This block runs **only if** the condition `radius <= 0` is true.
3. It prints a message to the user explaining that the calculations can't be done with a non-positive radius.
5. This keyword introduces an **alternative path**.
6. If the `if` condition is **false** (i.e., radius is greater than 0), the code inside this block will run.
7. These lines calculate the **circumference** and **area** of the circle using the formulae:

Circumference: $2 \times \pi \times \text{radius}$

Area: $\pi \times \text{radius} \times \text{radius}$

Key Concepts

- **Selection (Decision Making):** Using `if` and `else` to choose between two paths.
- **Relational Operator:** `<=` checks if one value is less than or equal to another.
- **Code Blocks:** `{ ... }` group multiple lines of code to be executed together.
- **Input Validation:** This is a basic example of checking if user input is valid before performing calculations.

Why This Is Important

This kind of selection is crucial in real-world programs to:

- Prevent errors (like dividing by zero or using invalid input).
- Make decisions based on user input or program state.
- Control the flow of the program.

Relational operators

Operator	Name	Example	Result (if a = 5, b = 10)
<code>==</code>	Equal to (<i>note: single equals (=) is used for assigning values</i>)	<code>a == b</code>	false
<code>!=</code>	Not equal to	<code>a != b</code>	true
<code>></code>	Greater than	<code>a > b</code>	false
<code><</code>	Less than	<code>a < b</code>	true
<code>>=</code>	Greater than or equal	<code>a >= b</code>	false
<code><=</code>	Less than or equal	<code>a <= b</code>	true

These operators are typically used in conditional statements like `if`, `while`, and `for`.

Definite iteration

An example of definite iteration to find the circumferences of circles with radius from 1 up to 10 is below.

```
1  for (int r = 1; r <= 10; r++)
2  {
3      circumference = 2 * pi * r;
4      Console.WriteLine("radius: " + r + ", circumference: " + circumference);
5  }
```

What the Code Does

This program uses a for loop to:

1. Go through radius values from **1 to 10**.
2. Calculate the **circumference** for each radius.
3. Print the radius and its corresponding circumference.

Line-by-Line Breakdown

1. This is a **for loop**, which is used to repeat a block of code a specific number of times.
int r = 1: This initializes the loop variable r to 1. It represents the radius.
r <= 10: This is the **condition**. The loop will keep running as long as r is less than or equal to 10.
r++: This increases the value of r by 1 after each loop iteration.
So, the loop will run with r values: 1, 2, 3, ..., up to 10.
3. Inside the loop, this line calculates the **circumference** using the formula:
$$\text{Circumference} = 2 \times \pi \times \text{radius}$$

Here, r is the current radius value in the loop.
4. This prints the current radius and its corresponding circumference to the console.
The output will look something like:
radius: 1, circumference: 6.28
radius: 2, circumference: 12.56
...
radius: 10, circumference: 62.8

Key Concepts

- **Iteration:** Repeating a task multiple times using a loop.
- **for Loop Structure:** Repeating a task a fixed number of times.
- **Loop Variable:** r changes with each iteration, allowing you to perform calculations for different values.
- **Output:** Each loop iteration prints a new line with updated values.

Why This Is Useful

- Helps automate repetitive tasks.
- Great for generating tables, performing batch calculations, or processing lists of data.

Indefinite Iteration

An example of indefinite iteration to find the radius and area of increasingly large circles up to an area of 100 is given below.

```
1 radius = 1; // Start with radius 1
2 while (area <= 100)
3 {
4     area = pi * radius * radius;
5     Console.WriteLine("Area: " + area);
6     radius++;
7 }
```

What the Code Does

This program:

1. Starts with a radius of 1.
2. Calculates the area of a circle using that radius.
3. Continues increasing the radius and recalculating the area **until the area becomes greater than 100**.
4. Prints the area at each step.

Line-by-Line Breakdown

1. Initialises the radius variable to 1 (note the use of comments in the code).
2. This is the starting point for the loop.
This is a **while loop**, which repeats as long as the condition is true.
The loop will continue running **as long as area is less than or equal to 100**.
4. Calculates the area of a circle using the formula:
$$\text{Area} = \pi \times \text{radius} \times \text{radius}$$

Stores the result in the area variable.
5. Prints the current area to the console.
6. Increases the radius by 1 for the next loop iteration.

Key Concepts

- **Iteration with while:** Repeats a block of code as long as a condition is true.
- **Variable Update:** `radius++` ensures the loop progresses and eventually ends.
- **Condition Checking:** The loop stops as soon as the area becomes greater than 100.

Randomness

The following example shows how a number between 1 and 100 can be randomly generated as well as how indefinite iteration can be used to create a guessing game.

```
1  int randomNumber;
2  Random random = new Random();
3  randomNumber = random.Next(1, 101); // Random number between 1 and 100
4  int userGuess;
5  do
6  {
7      Console.WriteLine("Guess the random number (between 1 and 100):");
8      userGuess = Convert.ToInt32(Console.ReadLine());
10     if (userGuess < randomNumber)
11     {
12         Console.WriteLine("Too low! Try again.");
13     }
14     else if (userGuess > randomNumber)
15     {
16         Console.WriteLine("Too high! Try again.");
17     }
18     else
19     {
20         Console.WriteLine("Congratulations! You've guessed the number: " + randomNumber);
21     }
22 }
23 while (userGuess != randomNumber);
```

What the Code Does

This program:

1. Generates a random number between 1 and 100.
2. Asks the user to guess the number.
3. Gives feedback on whether the guess is too low, too high, or correct.
4. Repeats until the user guesses the correct number.

Line-by-Line Breakdown

2. `Random random = new Random();` creates a new random number generator.
3. `random.Next(1, 101)` generates a random integer between **1 and 100** (inclusive).
The result is stored in `randomNumber`.
4. Declares a variable to store the user's guess.

The do...while Loop

5. This block **always runs at least once**, regardless of the condition.
7. It prompts the user to enter a guess and converts the input to an integer.
10. This is a **selection structure** (if...else if...else) that gives feedback based on the guess:
 - Too low → prompt to guess higher.
 - Too high → prompt to guess lower.
 - Correct → congratulate the user.
23. The loop **continues** as long as the guess is **not equal** to the random number.

Comparison with Previous Iteration Structures

Feature	while Loop	do...while Loop
Condition Checked	Before the loop starts	After the loop runs once
Guaranteed Execution	May not run at all if condition is false	Always runs at least once
Use Case	When you might skip the loop entirely	When you need to run the loop at least once

In this guessing game, the do...while loop is perfect because we **want the user to guess at least once** before checking if they're correct.

Key Concepts

- **Randomness:** `Random.Next(min, max)` generates unpredictable values. The random number could include min, but not max.
- **Iteration:** `do...while` ensures the loop runs at least once.
- **Selection:** `if...else` gives different responses based on conditions.
- **User Input:** `Console.ReadLine()` and `Convert.ToInt32()` handle input and conversion.

Integer Division and Modulus

The following C# code demonstrates how to use **integer division (/)** and the **modulus operator (%)** to break down a total number of minutes into **hours and minutes**.

```
1 Console.WriteLine("Enter the total number of minutes: ");
2 int minutes = Convert.ToInt32(Console.ReadLine());
3 int hours = minutes / 60; // integer division to get hours
4 int remainingMinutes = minutes % 60; // modulus to get remaining minutes
5 Console.WriteLine("Total time: {0} hours and {1} minutes", hours, remainingMinutes);
```

What the Code Does

The program:

1. Asks the user to enter a total number of minutes.
2. Converts that total into **hours** and **remaining minutes**.
3. Displays the result in a readable format.

Line-by-Line Breakdown

1. Prompts the user to enter a number.
2. Reads the user's input as a string.
Converts it to an integer using `Convert.ToInt32()`.
Stores it in the variable `minutes`.

Integer Division

3. This uses **integer division**.
It divides the total minutes by 60 (since there are 60 minutes in an hour).
The result is the **whole number of hours**.
Example: If `minutes = 135`, then `hours = 135 / 60 = 2`.

Modulus Operator

4. The **modulus operator %** gives the **remainder** after division.
This tells us how many minutes are **left over** after converting to hours.
Example: `remainingMinutes = 135 % 60 = 15`.

Output

5. Displays the result using **formatted output**.
`{0}` is replaced by `hours`, and `{1}` is replaced by `remainingMinutes`.
Example output:
Total time: 2 hours and 15 minutes

Key Concepts

- **Integer Division (/)**: Gives the whole number part of a division.
- **Modulus (%)**: Gives the remainder of a division.
- **User Input**: `Console.ReadLine()` and `Convert.ToInt32()` are used to read and convert input.
- **Formatted Output**: `Console.WriteLine()` can use placeholders for cleaner output.

Array (of Strings)

The following C# code sets up an **array** of strings (text data types) in C# along with **user input** and **searching**.

```
1 string[] vegetables = { "Carrot", "Potato", "Tomato", "Cabbage", "Spinach" };
2 Console.WriteLine("Enter a vegetable to search for:");
3 string findVegetable = Console.ReadLine();
4 if (vegetables.Contains(findVegetable))
5 {
6     Console.WriteLine("Vegetable found: " + findVegetable);
7 }
8 else
9 {
10    Console.WriteLine("Vegetable not found.");
11 }
```

What the Code Does

This program:

1. Stores a list of vegetables in an array.
2. Asks the user to enter the name of a vegetable.
3. Checks if that vegetable is in the list.
4. Displays a message saying whether it was found or not.

Line-by-Line Breakdown

1. Declares and initializes an **array of strings** called `vegetables`.
The `[]` after the array's data type (`string`) defines it as an array (of strings).
Arrays are used to store **multiple values** of the same type.
This array contains 5 vegetable names.
2. Prompts the user to type in the name of a vegetable.
3. Reads the user's input from the console and stores it in a variable called `findVegetable`.

Searching the Array

4. This line checks if the array `vegetables` **contains** the value entered by the user.
The method `.Contains()` is a **built-in method** that returns true if the item is found, and false otherwise.
6. If the vegetable is found, this message is displayed.
10. If the vegetable is **not** found in the array, this message is shown instead.

Key Concepts

- **Arrays:** Used to store a fixed-size collection of elements of the same type.
- **String Array:** `string[]` holds multiple text values.
- **User Input:** `Console.ReadLine()` reads what the user types.
- **Searching:** `.Contains()` checks if a value exists in the array.
- **Conditional Logic:** `if...else` decides what message to show based on the result.

Array (of Integers)

The following C# code shows how to use an **array** to store and generate the **Fibonacci sequence** using a for loop

```
1  int[] fibonacci = new int[10];
2  fibonacci[0] = 0; // First Fibonacci number
3  fibonacci[1] = 1; // Second Fibonacci number
4  for (int i = 2; i < fibonacci.Length; i++)
5  {
6      fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2]; // Calculate next Fibonacci number
7  }
8  Console.WriteLine("Fibonacci Series:");
9  foreach (int n in fibonacci)
10 {
11     Console.Write(n + " "); // Print each Fibonacci number
12 }
```

What the Code Does

This program:

1. Creates an array to hold the first 10 Fibonacci numbers.
2. Initialises the first two numbers of the sequence.
3. Uses a loop to calculate the rest of the sequence.
4. Prints all the numbers in the sequence.

Line-by-Line Breakdown

1. Declares an **array of integers** named `fibonacci` with **10 elements**.
2. Manually sets the first two numbers of the Fibonacci sequence:

```
fibonacci[0] = 0
fibonacci[1] = 1
```

Generating the Sequence

4. This for loop starts at index `i = 2` and goes up to `i = 9` (since the array has 10 elements and it starts at zero).
6. Each new number is the **sum of the two previous numbers**:
`fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2]`
This is the defining rule of the Fibonacci sequence.

Displaying the Sequence

8. Prints a heading before showing the numbers.
9. This foreach loop goes through each number in the `fibonacci` array.
11. It prints each number followed by a space.

Key Concepts

- **Arrays:** Used to store a fixed number of values.
- **Indexing:** Accessing elements using their position (starting from 0).
- **Loops:**
 - for loop: Used to calculate values based on previous elements.
 - foreach loop: Used to read and display each value.
- **Fibonacci Sequence:** A series where each number is the sum of the two before it.

Tasks to complete

Use the code examples above to create the following programs using a C# Console App in Visual Studio. If you have not been able to download Visual Studio then use an online editor such as .NET Fiddle (dotnetfiddle.net) or W3 Schools Online Compiler (www.w3schools.com/cs/cs_compiler.php).

Task 1: Dice Cricket

This is a very basic idea for a simple dice-based game. The rules are that each player takes it in turns to keep rolling a dice, adding up the cumulative score, until they get a 5 at which point they are out. The player with the highest score wins.

The code required for each player's turn could be described as follows:

1. Start with a total score of zero.
2. While the dice roll is not a 5
 - Roll the dice
 - If the result is not 5 then add it to the total score.
 - If the result is a 5 then the player is out.
3. Display the total score.

As an example for a short piece of code, this introduces some of the most useful basic coding concepts: Sequencing, Selection, Indefinite iteration, and Randomness.

Variables:

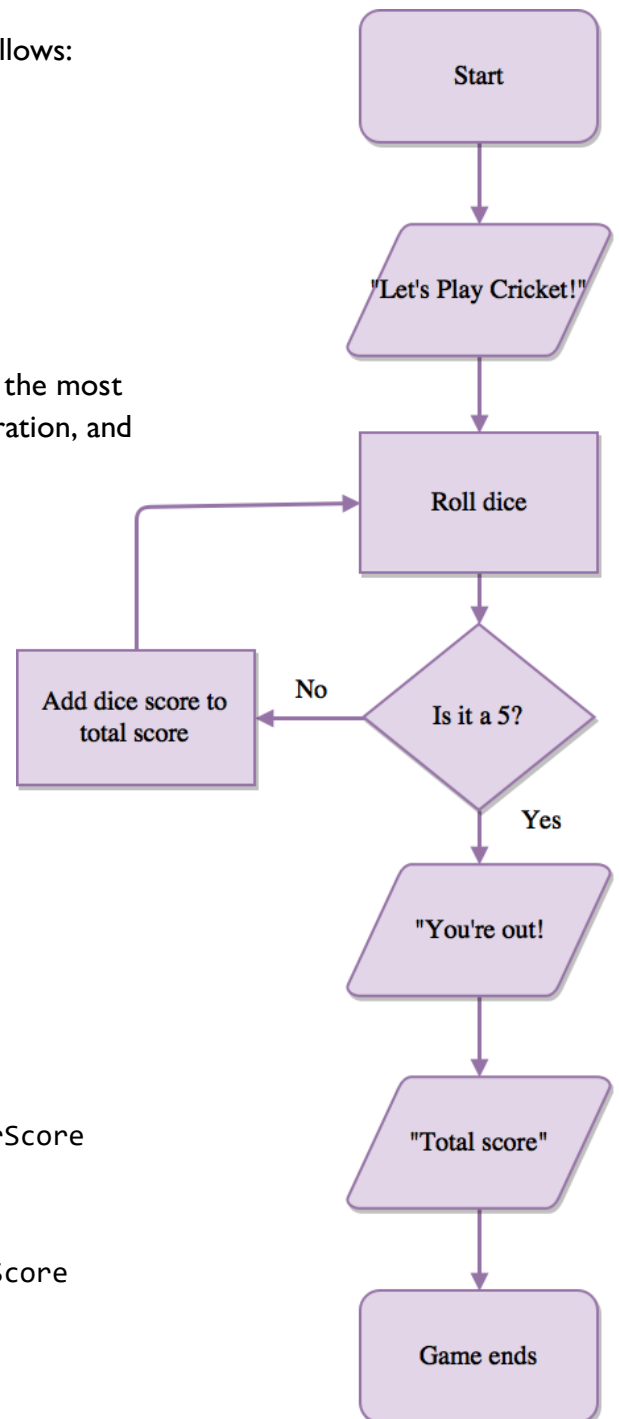
- `playerScore` (Integer, stores the player's current score)
- `diceRoll` (Integer, stores the result of the dice roll)

Pseudocode:

```
playerScore ← 0
diceRoll ← 1
OUTPUT "Welcome to Dice Cricket!"
OUTPUT "Press Enter to roll the dice."

WHILE diceRoll ≠ 5
    INPUT // Wait for user to press Enter
    diceRoll ← RANDOM(1 TO 6)
    OUTPUT "You rolled: " + diceRoll
    IF diceRoll ≠ 5
        THEN
            playerScore ← playerScore + diceRoll
            OUTPUT "Your current score is: " + playerScore
        ENDIF
    ENDWHILE

OUTPUT "You got out! Your final score is: " + playerScore
```



Task 2: Converting Decimal to Binary

Takes any positive integer value and find its binary equivalent. For example $42 = 101010$

Variables:

- decNum (Integer, the number to convert)
- bit (Integer, takes the remainder of the division (0 or 1))
- binStr (String, stores the bits into a string)


Pseudocode:

```
OUTPUT "Enter a decimal integer to convert:"  
INPUT decNum
```

```
binStr ← ""
```

```
WHILE decNum > 0  
    bit ← decNum MOD 2  
    binStr ← bit + binStr  
    decNum ← decNum DIV 2  
ENDWHILE
```

```
OUTPUT "Binary representation: " + binStr
```



decNum > 0	bit	binStr	decNum
True	$42 \text{ MOD } 2 = 0$	0	$42 \text{ DIV } 2 = 21$
True	$21 \text{ MOD } 2 = 1$	10	$21 \text{ DIV } 2 = 10$
True	$10 \text{ MOD } 2 = 0$	010	$10 \text{ DIV } 2 = 5$
True	$5 \text{ MOD } 2 = 1$	1010	$5 \text{ DIV } 2 = 2$
True	$2 \text{ MOD } 2 = 0$	01010	$2 \text{ DIV } 2 = 1$
True	$1 \text{ MOD } 2 = 1$	101010	$1 \text{ DIV } 2 = 0$
False			

Task 3: Bubble Sort

A basic bubble sort that takes an array of numbers, checks each pair in turn, and swaps them if they are in the wrong order. This repeats a sufficient number of times to ensure all numbers are correctly placed in order.

Variables:

- Numbers[0..7] (Array of Integer, stores 8 values to be sorted)
- Temp (Integer, used for swapping values)
- Pass (Integer, tracks the number of passes through the array)
- Index (Integer, tracks the current position in the array in each pass)

Pseudocode:

```
FOR Pass ← 0 TO 6  
    FOR Index ← 0 TO 6  
        IF Numbers[Index] > Numbers[Index + 1]  
            THEN  
                Temp ← Numbers[Index]  
                Numbers[Index] ← Numbers[Index + 1]  
                Numbers[Index + 1] ← Temp  
            ENDIF  
        ENDFOR  
    ENDFOR
```

```
OUTPUT "Sorted array:"  
FOR i ← 0 TO 7  
    OUTPUT Numbers[i]  
ENDFOR
```



Extension tasks

Dice cricket:

- Add random methods for the player to be 'out' e.g. bowled, stumped, caught, run out, etc.
- Turn it into a two player game.
- Allow the second player to be the computer.
- Create a leader board of high scores.

Decimal to binary converter:

- Add Input Validation. Make sure the user enters valid numbers (greater than zero).
- Add leading zeros to a binary number so that there are always (multiples of) 8 bits.
For example: 42 = 00101010 and 2025 = 00000111 11101001

Bubble sort:

The bubble sort can be made more efficient in two ways.

- (1) Reduce the numbers of pairs checked in each pass by one each time (the largest number(s) 'bubble' to the end on each pass).
- (2) Only continue to run through a new pass if no swaps have been made.
This will require a 'boolean flag' e.g. `bool noSwaps`, and the outer for loop needs to be changed to a while loop using the flag as a condition.