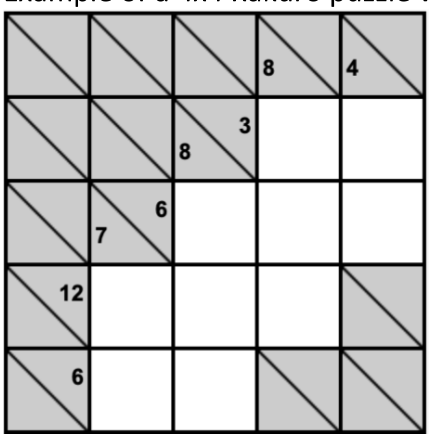
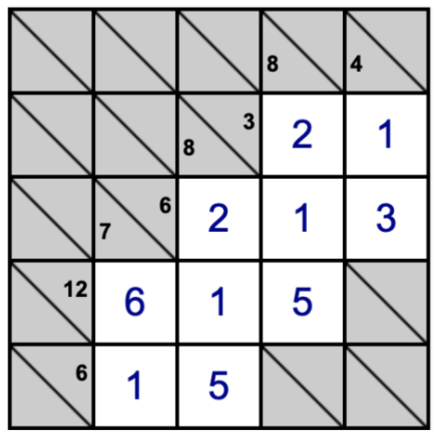
Kakuro

Kakuro is a type of puzzle game that originated in Japan. It is sometimes also called "Cross Sums" or "Kakro" and is similar to crossword puzzles and Sudoku. The game is played on a grid, typically a square grid, which is divided into cells. Some of the cells are blank, while others contain a number from 1 to 9.
The objective of the game is to fill in the blank cells with numbers such that the sum of the numbers in each "run" or "block" of consecutive cells matches a given target sum. The runs can be horizontal or vertical and can consist of one or more cells.

Example of a 4x4 Kakuro puzzle with its solution:





For this project we will be using a 4x4 Kakuro puzzle with a different constraint i.e., digits 0 – 3. Based on these constraints the max possible horizontal or vertical sum would be 1 + 2 + 3 = 6.

We would be using this 4x4 Kakuro example from classic.io for our problem.

	5	3	1
3	X0	X1	
5	X2	X3	X4
1		X5	X6

Steps to Solution

Find missing cells:
For the above problem, the missing cells are X0 – X6, which means nVariables (number of variables) will be 7.

Possible values for missing cells:
For a regular Kakuro puzzle possible values would range from 1 – 9, but we would be limiting the possible values for this project from 0 – 3.
This can be represented using 2 qubits i.e.

- 00 > 0
- 10 > 1
- 01 > 2
- 11 > 3

Get the constraints:
The constraints for this problem would be:

1. X0 != X1
2. X0 != X2
3. X1 != X3
4. X1 != X5
5. X2 != X3
6. X2 != X4
7. X3 != X4
8. X3 != X5
9. X4 != X6
10. X5 != X6
11. X0 + X1 == 3
12. X0 + X2 == 5
13. X1 + X3 + X5 == 3
14. X2 + X3 + X4 == 5
15. X4 + X6 == 1
16. X5 + X6 == 1

The constraints can be divided into 2 types.

- inequalityConstraints (2 variables can’t have same values)
  [(0, 1), (0, 2), (1, 3), (1, 5), (2, 3), (2, 4), (3, 4), (3, 5), (4, 6), (5, 6)]
- sumEqualityConstraints (variables must sum up to a given value)
  [([0,1], 3), ([0,2], 5), ([1,3,5], 3), ([2,3,4], 5), ([4,6], 1), ([5,6], 1)]

For sumEqualityConstraint:

1. We get all the possible sum options based of the sumEqualityConstriant.

```
// Gets all the possible sum options for a given constraint
function SumOptions(constriant: SEC): Int[][] {
    mutable result = [];
    let variables = constriant::variables;
    let sumValue = constriant::sum;
    if Length(variables) == 2 {
        for i in 0 .. 3 {
            if sumValue - i < 4 and sumValue - i > -1 {
                if i != sumValue - i {
                    set result += [[i, sumValue - i]];
                }
            }
        }
    } elif Length(variables) == 3 {
        for i in 0 .. 3 {
            if sumValue - i < 4 and sumValue - i > -1 {
                let options = SumOptions(SEC([0,1], i));
                for o in options {
                    if sumValue - i != o[0] and sumValue - i != o[1] and o[0] != o[1] {
                        set result += [[sumValue-i, o[0], o[1]]];
                        set result += [[o[0], sumValue-i, o[1]]];
                        set result += [[o[0], o[1], sumValue-i]];
                    }
                }
            }
        }
    }
    return result;
}
```

2. **From the sum options we can get sumConstraints (i.e. numbers a particular variable can't be)**

   [(0, 0),(0, 1),(1, 3),(2, 0),(2, 1),(3, 1),(3, 3),(4, 1),(4, 2),(4, 3),(5, 2),(5, 3),(6, 2),(6, 3)]

   X0 can't be [0,1]

   X1 can't be [3]

   X2 can't be [0,1] ……..

```
// Find the puzzle empty variables and there sums
// Then gets the sum constraints for each empty variable
function FindSumConstraints(sumEqualityConstraints: SEC[], nVariables: Int): (Int,Int)[] {
    mutable result = [];
    let limitations = [0, 1, 2, 3];
    for i in 0 .. nVariables - 1 {
        mutable x = 0;
        mutable options1 = [];
        mutable options2 = [];
        mutable finalOptions = [];
        for constriaint in sumEqualityConstraints {
            let variables = constriaint::variables;
            for j in 0 .. Length(variables) - 1 {
                if variables[j] == i {
                    if x == 0 {
                        for option in SumOptions(constriaint) {
                            set options1 += [option[j]];
                        }
                    } elif x == 1 {
                        for option in SumOptions(constriaint) {
                            set options2 += [option[j]];
                        }
                        set finalOptions = RemoveDuplicates(FindCommonElementsInArrays(options1, options2));
                    }
                    set x += 1;
                }
            }
        }
        Message($"Possible value(s) for X{i}: {finalOptions}");
        let LimitedOptions = GetLimitationConstraints(finalOptions, limitations);

        for option in LimitedOptions {
            set result += [(i, option)];
        }
    }

    Message($"Sum constraints: {result}");
    return result;
}
```

3. Use the sumConstraints we get to prepare an equal superposition that satisfy it

```
// Encodes sum constraints into amplitudes
function AllowedAmplitudes(nVariables: Int, valueQubits: Int, sumConstraints: (Int, Int)[]) : Double[][] {
    mutable amplitudes = [[1.0, size=1 <<< valueQubits], size=nVariables];
    for (cell, value) in sumConstraints {
        set amplitudes w/= cell <- (amplitudes[cell] w/ value <- 0.0);
    }
    return amplitudes;
}

// Prepare an equal superposition of all basis states that satisfy the constraints
operation PrepareSearchStatesSuperposition(nVariables: Int, valueQubits: Int,
    sumConstraints: (Int, Int)[], register : Qubit[]
): Unit is Adj + Ctl {
    // Split the given register into nVariables chunks of size bits per value
    let valueRegister = Chunks(valueQubits, register);
    // For each variable, create an array of possible states we're looking at
    let amplitudes = AllowedAmplitudes(nVariables, valueQubits, sumConstraints);
    // For each variable, prepare a superposition of its possible states on the chunk storing its value
    for (amps, chunk) in Zipped(amplitudes, valueRegister) {
        PrepareArbitraryStateD(amps, LittleEndian(chunk));
    }
}
```

For inequalityConstraint:

1. We check if 2 variable values are the same

```
// Checks if the value of two variables are the same
operation ApplyValueEqualityOracle(vqb0: Qubit[], vqb1: Qubit[], target: Qubit): Unit is Adj + Ctl {
    within {
        // Iterates over pair of (vqb0, vqb1)
        // Then computes XOR of bits vqb0[i] and vqb1[i] in place (storing it in vqb1[i])
        ApplyToEachCA(CNOT, Zipped(vqb0, vqb1));
    } apply {
        // If all computed XORs are 0, then the values are equal, flip the target qubit
        ControlledOnInt(0, X)(vqb1, target);
    }
}
```

2. We check if the variable value is valid (there are no conflicts)

```
// Checks if the variable values satisfy the inequality constraints
operation ApplyVariableValuesOracles (nVariables :Int, valueQubits: Int,
    inequalityConstraints: (Int, Int)[],
    valueRegister: Qubit[], target: Qubit
): Unit is Adj + Ctl {
    let nInequalityConstraints = Length(inequalityConstraints);
    use inequalityConflictQubits = Qubit[nInequalityConstraints];
    within {
        for ((start, end), conflictQubit) in Zipped(inequalityConstraints, inequalityConflictQubits) {
            // if the values are the same the result will be 1, indicating a conflict
            ApplyValueEqualityOracle(
                valueRegister[start * valueQubits .. (start + 1) * valueQubits - 1],
                valueRegister[end * valueQubits .. (end + 1) * valueQubits - 1],
                conflictQubit
            );
        }
    } apply {
        // If no conflicts (all qubits are in 0 state), the variable is valid
        ControlledOnInt(0, X)(inequalityConflictQubits, target);
    }
}
```

3. Convert the marking oracle to phase oracle

```
// Convert the marking oracle to a phase oracle
operation ApplyPhaseOracle (oracle : ((Qubit[], Qubit) => Unit is Adj), register : Qubit[]): Unit is Adj {
    use target = Qubit();
    within {
        // Put the target into the |-⟩ state.
        X(target);
        H(target);
    } apply {
        // Apply the marking oracle; since the target is in the |-⟩ state,
        // flipping the target if the register satisfies the oracle condition
        // will apply a -1 factor to the state.
        oracle(register, target);
    // We put the target back into |0⟩ so we can return it.
    }
}
```

Get Solution:

Run Grover's search algorithm

```
// Unitary implementing Grover's search algorithm
operation ApplyGroversIteration(register: Qubit[],
    oracle: ((Qubit[], Qubit) => Unit is Adj),
    statePrep: (Qubit[] => Unit is Adj), iterations: Int
): Unit {
    let applyPhaseOracle = ApplyPhaseOracle(oracle, _);
    statePrep(register);

    for _ in 1 .. iterations {
        applyPhaseOracle(register);
        within {
            Adjoint statePrep(register);
            ApplyToEachA(X, register);
        } apply {
            Controlled Z(Most(register), Tail(register));
        }
    }
}

// Grover search to find variable valid values
operation FindValuesWithGrover(nVariables: Int, valueQubits: Int, nIterations: Int,
    oracle:((Qubit[], Qubit) => Unit is Adj),
    statePrep: (Qubit[] => Unit is Adj)
): Int[] {
    // The value register has bits per value for each variable
    use register = Qubit[valueQubits * nVariables];
    Message($"Trying search with {nIterations} iterations...");
    if (nIterations > 75) {
        Message($"Warning: This might take a while");
    }
    ApplyGroversIteration(register, oracle, statePrep, nIterations);
    return MeasureAllValues(valueQubits, register);
}
```

Solution:

When we run the project, we get

```
Solving Kakuro Puzzle using Grover's Algorithm
Possible value(s) for X0: [2,3]
Possible value(s) for X1: [2,0,1]
Possible value(s) for X2: [3,2]
Possible value(s) for X3: [0,2]
Possible value(s) for X4: [0]
Possible value(s) for X5: [1,0]
Possible value(s) for X6: [1,0]
Sum constraints: [(0, 0),(0, 1),(1, 3),(2, 0),(2, 1),(3, 1),(3, 3),(4, 1),(4, 2),(4, 3),(5, 2),(5, 3),(6, 2),(6, 3)]
Running Quantum test with number of variables = 7
   Bits per value = 2
   Inequality constraints = [(0, 1),(0, 2),(1, 3),(1, 5),(2, 3),(2, 4),(3, 4),(3, 5),(4, 6),(5, 6)]
   Sum Equality Constraints = [SEC(([0,2], 5)),SEC(([1,3,5], 3)),SEC(([4,6], 1)),SEC(([2,3,4], 5)),SEC(([0,1], 3)),SEC(([5,6], 1))]
   Estimated number iterations needed = 7
   Size of kakuro grid = 4x4
   Search space size = 96
Trying search with 7 iterations...
Got Sudoku solution: [2,1,3,2,0,0,1]
Got valid Sudoku solution: [2,1,3,2,0,0,1]
X0 = 2
X1 = 1
X2 = 3
X3 = 2
X4 = 0
X5 = 0
X6 = 1
```

Which matches the solution to the problem

	5	3	1
3	2	1	
5	3	2	0
1		0	1

Test Solution is Valid:

Kakuro problems have only one valid solution we test the solution given against the constraints.

```
// Checks if the value found for each empty variable is corrrect and satisfies all constraints
function IsKakuroSolutionValid(inequalityConstraints: (Int,Int)[],
    sumEqualityConstraints: SEC[], values: Int[]
) : Bool {
    if (Any(GreaterThanOrEqualI(_, 4), values)) {
        return false;
    }
    if (Any(EqualI, inequalityConstraints)) {
        return false;
    }

    for constraint in sumEqualityConstraints {
        let variables = constraint::variables;
        let expectedSum = constraint::sum;
        mutable valuesSum = 0;
        for variable in variables {
            set valuesSum += values[variable];
        }
        if valuesSum != expectedSum {
            return false;
        }
    }

    return true;
}
```

Full-State Simulator vs Sparse Simulator:

- Full-state simulator
  Takes on average 1min 24s to run on a full-state simulator

```
%%timeit
Start.simulate()
```

✓ 11 min 28 sec

```
   Inequality constraints = [(0, 1),(0, 2),(1, 3),(1, 5),(2, 3),(2, 4),(3, 4),(3, 5),(4, 6),(5, 6)]
   Sum Equality Constraints = [SEC(([0,2], 5)),SEC(([1,3,5], 3)),SEC(([4,6], 1)),SEC(([2,3,4], 5)),SEC(([0,1], 3)),SEC(([5,6], 1))]
   Estimated number iterations needed = 7
   Size of kakuro grid = 3x3
   Search space size = 96
Trying search with 7 iterations...
Got Sudoku solution: [2,1,3,2,0,0,1]
Got valid Sudoku solution: [2,1,3,2,0,0,1]
X0 = 2
X1 = 1
X2 = 3
X3 = 2
X4 = 0
X5 = 0
X6 = 1
1min 24s ± 1.53 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

- Sparse simulator
  Takes on average 68.5 ms to run on a sparse simulator (It's significantly faster)

```
%%timeit
Start.simulate_sparse()
```

✓ <1 sec

```
   Inequality constraints = [(0, 1),(0, 2),(1, 3),(1, 5),(2, 3),(2, 4),(3, 4),(3, 5),(4, 6),(5, 6)]
   Sum Equality Constraints = [SEC(([0,2], 5)),SEC(([1,3,5], 3)),SEC(([4,6], 1)),SEC(([2,3,4], 5)),SEC(([0,1], 3)),SEC(([5,6], 1))]
   Estimated number iterations needed = 7
   Size of kakuro grid = 3x3
   Search space size = 96
Trying search with 7 iterations...
Got Sudoku solution: [2,1,3,2,0,0,1]
Got valid Sudoku solution: [2,1,3,2,0,0,1]
X0 = 2
X1 = 1
X2 = 3
X3 = 2
X4 = 0
X5 = 0
X6 = 1
68.5 ms ± 3.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Resource estimation:

```
resource_estimaion_result = qsharp.azure.output("5e98e1f8-3649-4e56-a45a-3430d170480e")
resource_estimaion_result
```

✓ <1 sec

▼ **Physical resource estimates**

Physical qubits	240600	**Number of physical qubits** This value represents the total number of physical qubits, which is the sum of 37800 physical qubits to implement the algorithm logic, and 202800 physical qubits to execute the T factories that are responsible to produce the T states that are consumed by the algorithm.
Runtime	18ms 168us	**Total runtime** This is a runtime estimate (in nanosecond precision) for the execution time of the algorithm. In general, the execution time corresponds to the duration of one logical cycle (6us) multiplied by the 3028 logical cycles to run the algorithm. If however the duration of a single T factory (here: 67us 600ns) is larger than the algorithm runtime, we extend the number of logical cycles artificially in order to exceed the runtime of a single T factory.

▶ **Resource estimates breakdown**
▶ **Logical qubit parameters**
▶ **T factory parameters**
▶ **Pre-layout logical resources**
▶ **Assumed error budget**
▶ **Physical qubit parameters**
▶ **Assumptions**

Solving for smaller size 3 X 3 kakuro puzzle:

number of missing cells = 4

These are commented in the project code.

```
Solving Kakuro Puzzle using Grover's Algorithm
Possible value(s) for X0: [3]
Possible value(s) for X1: [1]
Possible value(s) for X2: [2]
Possible value(s) for X3: [0]
Sum constraints: [(0, 0),(0, 1),(0, 2),(1, 0),(1, 2),(1, 3),(2, 0),(2, 1),(2, 3),(3, 1),(3, 2),(3, 3)]
Running Quantum test with number of variables = 4
   Bits per value = 2
   Inequality constraints = [(0, 1),(0, 2),(1, 3),(2, 3)]
   Sum Equality Constraints = [SEC(([0,1], 4)),SEC(([0,2], 5)),SEC(([1,3], 1)),SEC(([2,3], 2))]
   Estimated number iterations needed = 0
   Size of kakuro grid = 3x3
   Search space size = 1
Trying search with 0 iterations...
Got Sudoku solution: [3,1,2,0]
Got valid Sudoku solution: [3,1,2,0]
X0 = 3
X1 = 1
X2 = 2
X3 = 0
```

Unit Test:

dotnet test --filter="Target = QuantumSimulator"

```
Running Quantum test with number of variables = 7
   Bits per value = 2
   Inequality constraints = [(0, 1),(0, 2),(1, 3),(1, 5),(2, 3),(2, 4),(3, 4),(3, 5),(4, 6),(5, 6)]
   Sum Equality Constraints = [SEC(([0,1], 3)),SEC(([0,2], 5)),SEC(([1,3,5], 3)),SEC(([2,3,4], 5)),SEC(([4,6], 1)),SEC(([5,6], 1))]
   Estimated number iterations needed = 7
   Size of kakuro grid = 4x4
   Search space size = 96
Trying search with 7 iterations...
Got Sudoku solution: [2,1,3,2,0,0,1]
Got valid Sudoku solution: [2,1,3,2,0,0,1]

Passed!  - Failed:     0, Passed:     5, Skipped:     0, Total:     5, Duration: 1 m 34 s - /Users/olasunkanmi/Documents/neu/spring_23/i
0/StandaloneProject.dll (net6.0)
```

Running on Hardware:

Unable to successfully run on quantum hardware, got errors I wasn't able to debug

```
qsharp.azure.target("ionq.qpu.aria-1")
qsharp.azure.submit(Start, jobName="Sudoku solver on ionq simulator")

11 sec - AzureError: {'error_code': 1011, 'error_name': 'JobSubmissionFailed', 'error_description': 'Failed to submit the job to the Azure Quantum workspace.'}

Loading package Microsoft.Quantum.Providers.IonQ and dependencies...
Active target is now ionq.qpu.aria-1
Submitting Start to target ionq.qpu.aria-1...
fail: Microsoft.Quantum.IQSharp.AzureClient.AzureClient[0]
      Failed to submit Q# operation Start for execution.
      System.AggregateException: One or more errors occurred. (Exception has been thrown by the target of an invocation.)
       ---> System.Reflection.TargetInvocationException: Exception has been thrown by the target of an invocation.
       ---> Microsoft.Quantum.Providers.Core.Processor.CannotCompareMeasurementResultException: Measurement results cannot be compared to Zero or One on the
target architecture
```