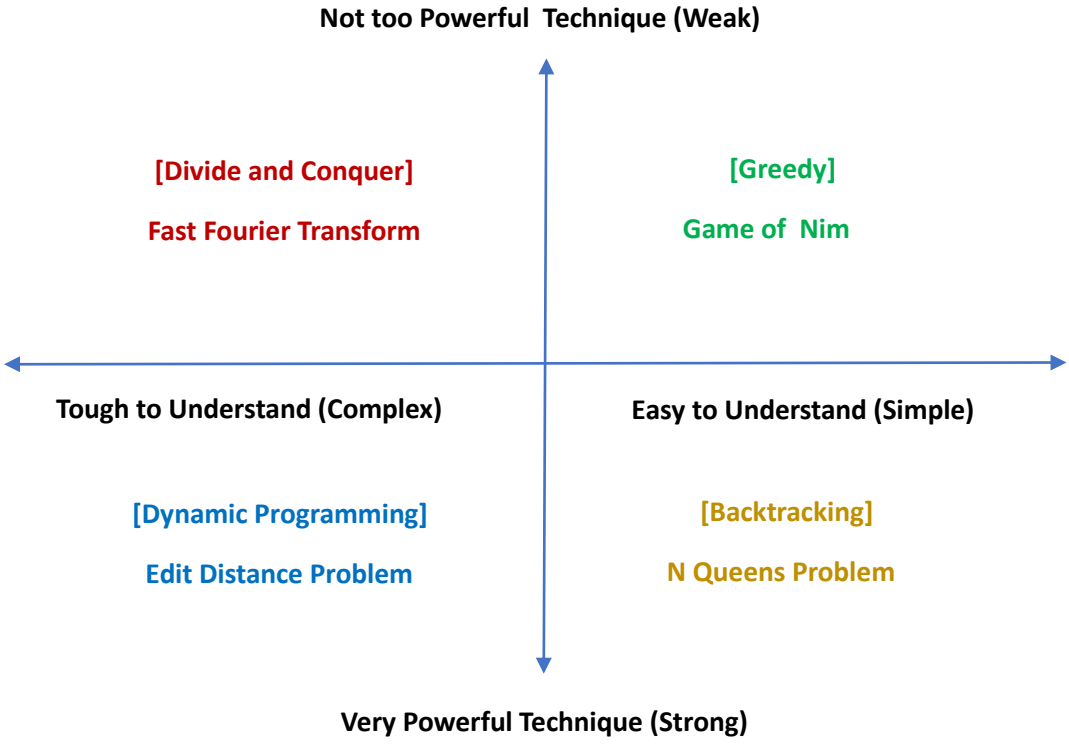
Date: 4th May 2023

The 4 Quadrants of Computing

"Abstraction is a killer. Pursue it and you will perish, let go and you will die!"



1. **Greedy Method: Essentially means that you select the most lucrative option at any instant of time. Notoriously called short-sighted or Myopic.**
   The class of Min-Max Problems fall into this category. Basically, Minimize the Opponent's Chances by assuming he will make the Most Optimal Maximum profitable move available to him at any given instant of time.
   - **Game of Nim – A very popular game which is deterministic in nature.**
     "There are 3 Heaps of 3,4,5 stones each. You can select any one pile and draw as many stones you wish. There are 2 players in the game. They take turns to pick stones from the Heaps. The player to pick the last stone from any pile loses the game. Given these as the rules of the game, is there a winning strategy to it?"

     Answer: Yes. In this case it is the player who starts second wins. Since, he gets a Nim-Sum of 0! Given that he maintains a Nim-Sum of 0 after each turn! Which essentially means an XOR with all the current sizes of the Heaps, to ascertain whether that number of stones can be picked from the pile or not!

     Here's some code that illustrates precisely that:

```
int xors = 0;
for (int i = 0; i < xs.Length; i++)
{
    xors ^= xs[i];
}
return xors;
```

```
public void Play(List<Player> players)
        {
            int turn = 0;
            do
            {
                var nimSum = Helper.Xors(Heaps.Select(h => h.CurSize).ToArray()); // Compute the Nim-Sum!
                if (nimSum == 0)
                {
                    Helper.RandomPick(Players[turn % Players.Count], Heaps); // Nim-Sum is 0, no matter what
you do you are bound to lose!
                    turn++;
                    continue;
                }
                for (int h = 0; h < Heaps.Count; h++)
                {
                    var heapNimSum = nimSum ^ Heaps[h].CurSize;
                    if (heapNimSum == 0)
                    {
                        Helper.RandomPick(Players[turn % Players.Count], Heaps); // Make any Random move
since you can never win now once the Nim-Sum is 0.
                        turn++;
                        break;
                    }
                    else
                    {
                        Helper.OptimalPick(Players[turn % Players.Count], Heaps, heapNimSum); // Make the
best move since you can win now once the Nim-Sum is 0.

                        turn++;
                        break;
                    }
                }
            } while (Heaps.Any(h => h.CurSize != 0));
        }
```

2. **<u>Divide and Conquer</u>: Talks about dividing the Problem into smaller identical spatial parts & then use that to parallelly compute all the possible solutions at any instant of time with tempo. Subsequently the smaller solutions are then merged to arrive at the final grand solution! This exploits parallelism to the fullest at the grandest scale! Merge Sort, Quick Sort, Closest points in a Plane & the FFT itself fall into this category or class of problems!**
   - **<u>FFT</u>: The Fast Fourier Transform is a popular technique to find the Discrete Fourier Transform of a signal sampled at discrete intervals decimated in time by exploiting the nature of the cyclic tendencies of the sine & cosine waveforms! This quintessentially means that the large set of points say 4096 for instance, may be computed in $O(N\log_2 N)$ time rather than $O(N^2)$.**
     If it takes 4 ms to compute DFT of 4096 points, then by FFT $T_1=4*2=8$ ms; by DFT $T_2=4*4=16$ ms. A speedup of 2x for the FFT is clearly remarkable!

$X_K = \sum x_n \times \cos(2\pi kn/N) - i \times \sum x_n \times \sin(2\pi kn/N)$

i.e.

$X_K = \sum x_n \times e^{(-2\pi ikn/N)}$

Now, consider $X_{k+N/2}$,

$X_{k+N/2} = \sum x_n \times e^{(-2\pi i(k+N/2)n/N)} = \sum x_n \times e^{(-2\pi i(k+N/2)n/N)}$

$= \sum x_n \times e^{(-2\pi i(k)n/N)} \times e^{(-2\pi i(N/2)/N)}$

$= \sum x_n \times e^{(-2\pi i(k)n/N)} \times e^{(-\pi i)}$

$= -\sum x_n \times e^{(-2\pi i(k)n/N)}$

Observe, that for $k \in [0, N/2]$,

$X_K = \sum x_n \times e^{(-2\pi ikn/N)} = x_0 \times e^{(-2\pi ik0/N)} + x_1 \times e^{(-2\pi ik1/N)} + x_2 \times e^{(-2\pi ik2/N)} + \ldots. x_{N/2-1} \times e^{(-2\pi ik(N/2-1)/N)}$

- $\Rightarrow X_K = x_0 + x_1 \times e^{(-2\pi ik1/N)} + x_2 \times e^{(-2\pi ik2/N)} + x_3 \times e^{(-2\pi ik3/N)} + \ldots. x_{N/2-1} \times e^{(-2\pi ik(N/2-1)/N)}$
- $\Rightarrow X_K = x_0 + x_2 \times e^{(-2\pi ik(2)/N)} + x_4 \times e^{(-2\pi ik4/N)} + \ldots. + x_1 \times e^{(-2\pi ik/N)} + x_3 \times e^{(-2\pi ik3/N)} + \ldots. x_{N/2-1} \times e^{(-2\pi ik(N/2-1)/N)}$
- $\Rightarrow X_K = [x_0 + x_2 \times e^{(-2\pi ik(2)/N)} + x_4 \times e^{(-2\pi ik4/N)} + \ldots. x_{N/2-2} \times e^{(-2\pi ik(N/2-2)/N)}] + e^{(-2\pi ik/N)} [x_1 + x_3 \times e^{(-2\pi ik2/N)} + \ldots. x_{N/2-1} \times e^{(-2\pi ik(N/2-2)/N)}]$

  By the clever trick as shown below, let $j = 2k$, so that j runs with increments of 1 =>

- $\Rightarrow X_j = [x_0 + x_1 \times e^{(-2\pi ij/N)} + x_2 \times e^{(-2\pi i2j/N)} + \ldots. x_{N/2-2} \times e^{(-2\pi i(j/2)(N/2-2)/N)}] + e^{(-2\pi ik/N)} [x_1 + x_2 \times e^{(-2\pi ij/N)} + \ldots. x_{N/2-1} \times e^{(-2\pi i(j/2)(N/2-2)/N)}]$
- $\Rightarrow X_j = X_{ke} + e^{(-2\pi ik/N)} X_{ko}$

Another neater trick is to pull out $e^{(-2\pi i(N/2)/N)} = e^{(-\pi i)}$, as the common factor from the progression,

$X_K = \sum x_n \times e^{(-2\pi ikn/N)} = -x_{N/2} \times e^{(-2\pi ik(N/2)/N)} - x_{N/2+1} \times e^{(-2\pi ik(N/2+1)/N)} - x_{N/2+2} \times e^{(-2\pi ik(N/2+2)/N)} - \ldots. x_N \times e^{(-2\pi ik(N)/N)}$

- $\Rightarrow X_j = -x_{N/2} \times e^{(-2\pi ik(N/2)/N)} - x_{N/2+1} \times e^{(-2\pi ik(N/2+1)/N)} - x_{N/2+2} \times e^{(-2\pi ik(N/2+2)/N)} - \ldots. x_N \times e^{(-2\pi ik(N)/N)}$
- $\Rightarrow X_j = e^{(-\pi i)} [-x_{N/2} \times e^{(-2\pi ik(0)/N)} - x_{N/2+1} \times e^{(-2\pi ik1/N)} - x_{N/2+2} \times e^{(-2\pi ik(2)/N)} - \ldots. x_N \times e^{(-2\pi ik(N/2)/N)}]$

Similarly, for $k \in [N/2, N]$, let $j = k - N/2$, so that j runs in increments of 1 & within the desired range =>

- $\Rightarrow X_j = e^{(-\pi i)} [-x_0 \times e^{(-2\pi ij(0)/N)} - x_1 \times e^{(-2\pi ij1)/N)} \times e^{(-\pi i)} - x_2 \times e^{(-2\pi ij(2)/N)} - x_3 \times e^{(-2\pi ij(3)/N)} \times e^{(-\pi i)} \ldots. x_N \times e^{(-2\pi ij(N/2)/N)}]$
- $\Rightarrow X_j = x_0 \times e^{(-2\pi ij(0)/N)} - x_1 \times e^{(-2\pi ij/N)} + x_2 \times e^{(-2\pi ij(2)/N)} - x_3 \times e^{(-2\pi ij(3)/N)} + \ldots. x_{N/2} \times e^{(-2\pi ik(N)/N)}$
- $\Rightarrow X_j = [x_0 + x_2 \times e^{(-2\pi ij(2)/N)} + \ldots. x_{N/2} \times e^{(-2\pi ij(N/2)/N)}] - [x_1 \times e^{(-2\pi ij/N)} + x_3 \times e^{(-2\pi ij(3)/N)} + \ldots. x_{N/2-1} \times e^{(-2\pi ik(N/2-1)/N)}]$

Now, let $j' = 2j$, so that the indices are now well & truly balanced =>

- $\Rightarrow X_{j'} = [x_0 + x_1 \times e^{(-2\pi ij'/N)} + \ldots. x_{N/2-2} \times e^{(-2\pi ij'/N)}] - e^{(-2\pi ij/N)} [x_1 + x_2 \times e^{(-2\pi ij(2)/N)} + \ldots. x_{N/2-1} \times e^{(-2\pi ij(N/2-1)/N)}]$

Observing that here j = k, since it gives us the complete series for all terms from 1 .. N =>

We conclude that,

- $\Rightarrow X_j = X_{ke} - e^{(-2\pi ik/N)} X_{ko}$

Hence, we get the important result, that,

$$X_k = \begin{cases} X_{even} + e^{(-2\pi ik/N)} X_{odd, \text{ for } k \in [0,N/2]} \\ X_{even} - e^{(-2\pi ik/N)} X_{odd, \text{ for } k \in [N/2,N]} \end{cases}$$

Which would be our basis for the Divide N Conquer strategy for the FFT!!

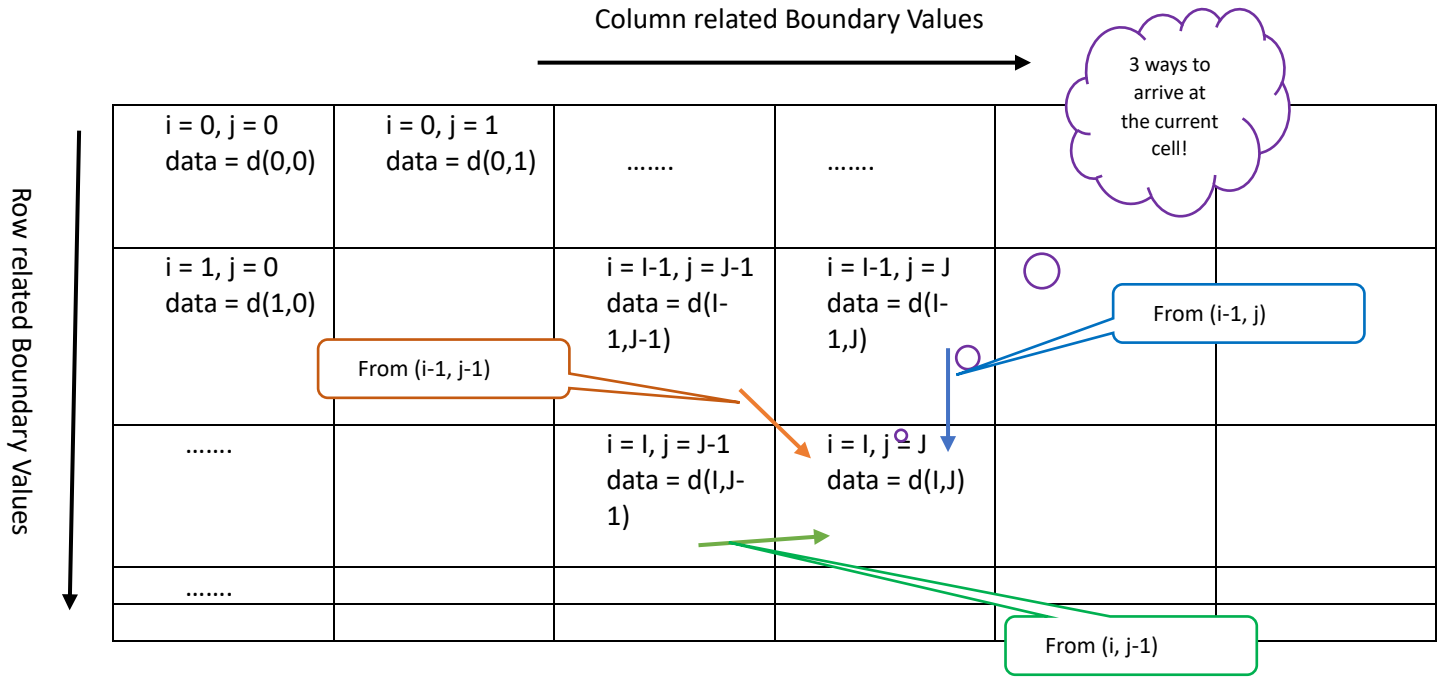
Here's some code to illustrate the technique precisely:

```
public List<Node> Fft(List<Node> x)
        {
            int N = x.Count; //fetch the number of samples

            if (N == 1)
            {
                return x; //base case :- one signal.
            }

            var Xe = Fft(Helper.Take(x, EvenOdd.Even)); //fetch the even terms :- the Divide phase
            var Xo = Fft(Helper.Take(x, EvenOdd.Odd));  //fetch the odd terms  :- the Divide phase

            var X = Helper.Take(x, EvenOdd.Both); //combine the odd + even terms :- the Conquer phase

            for (int k = 0; k < N / 2; k++)
            {
                Complex e = Xe[k].Value; //compute the even term
                Complex o = Helper.Exp(1, k, N) * Xo[k].Value; //compute the odd term
                X[k] = new Node(e + o); //Lower half range series is :- (even + odd) term
                X[k + N / 2] = new Node(e - o); //Upper half range series :- (even - odd) term
            }

            return X; //return the full range of signals
        }
```

3. **<u>Dynamic Programming</u> : The exceptional paradigm to exploit the recursive optimal subproblems concept. It involves memoization, which is a technique to fill a Memoized Table, and then pick the cell of concern, after taking into consideration boundary conditions, to arrive at the final answer!**
   Is a bit akin to solving the Differential Equations in College for numerous engineering problems.
   Some classic examples of this technique are Longest Common Subsequence, Max. Contiguous Sum in an Array, Knapsack Problem, Bin Packing problem and many more!
   - **<u>The Edit Distance Problem</u>: Find the least number of substitutions required to transform one word into another one!**

     Here's a sneak peek at the Memoization Method itself:

Memoization Table Filling Procedure :-



<u>Note</u>: The Other paths are not feasible due to the arrow & directionality of time! 😊

Here's a quick under the hood, bare metal implementation of the Edit Distance Problem's Build method for the Table Building:

Doesn't it remind you of the famous SQL Constructs? 😊

```
        public void Build(Table table)
        {
            for (int i = 1; i < table.i; i++)
            {
                for (int j = 1; j < table.j; j++)
                {
                    if (s[i - 1] == t[j - 1])
                    {
                        table.cells[i][j].row = i - 1;
                        table.cells[i][j].col = j - 1;
                        table.cells[i][j].opCounts = table.cells[i - 1][j - 1].opCounts; // From i-1,j-1 with no
changes or operations!!
                        if (t[j - 1] != null)
                            table.cells[i][j].operation = $"Copy {s[i - 1]} --> {t[j - 1]}";
                        else
                            table.cells[i][j].operation = string.Empty;
                        continue;
                    }

                    if (table.cells[i][j - 1].opCounts + 1 <= table.cells[i - 1][j - 1].opCounts &&
table.cells[i][j - 1].opCounts + 1 <= table.cells[i - 1][j].opCounts)
                    {
                        table.cells[i][j].row = i;
                        table.cells[i][j].col = j - 1;
                        table.cells[i][j].opCounts = table.cells[i][j - 1].opCounts + 1; // From i,j-1 with Create
as the operation!!
                        if (t[j - 1] != null)
                            table.cells[i][j].operation = $"Create {t[j - 1]}";
                        else
                            table.cells[i][j].operation = string.Empty;
                        continue;
                    }

                    if (table.cells[i - 1][j].opCounts + 1 <= table.cells[i - 1][j - 1].opCounts && table.cells[i -
1][j].opCounts + 1 <= table.cells[i][j - 1].opCounts)
                    {
                        table.cells[i][j].row = i - 1;
                        table.cells[i][j].col = j;
                        table.cells[i][j].opCounts = table.cells[i - 1][j].opCounts + 1; // From i-1,j with Drop as
the operation!!
                        if (s[i - 1] != null)
                            table.cells[i][j].operation = $"Drop {s[i - 1]}";
                        else
                            table.cells[i][j].operation = string.Empty;

                        continue;
                    }

                    if (table.cells[i - 1][j - 1].opCounts <= table.cells[i][j - 1].opCounts && table.cells[i -
1][j - 1].opCounts <= table.cells[i - 1][j].opCounts)
                    {
                        table.cells[i][j].row = i - 1;
                        table.cells[i][j].col = j - 1;
                        table.cells[i][j].opCounts = table.cells[i - 1][j - 1].opCounts; // From i-1,j-1 with
changes as Replace being the operation!!
                        if (t[j - 1] != null)
                            table.cells[i][j].operation = $"Replace {s[i - 1]} --> {t[j - 1]}";
                        else
                            table.cells[i][j].operation = string.Empty;
                        continue;
                    }
                }
            }
        }
```

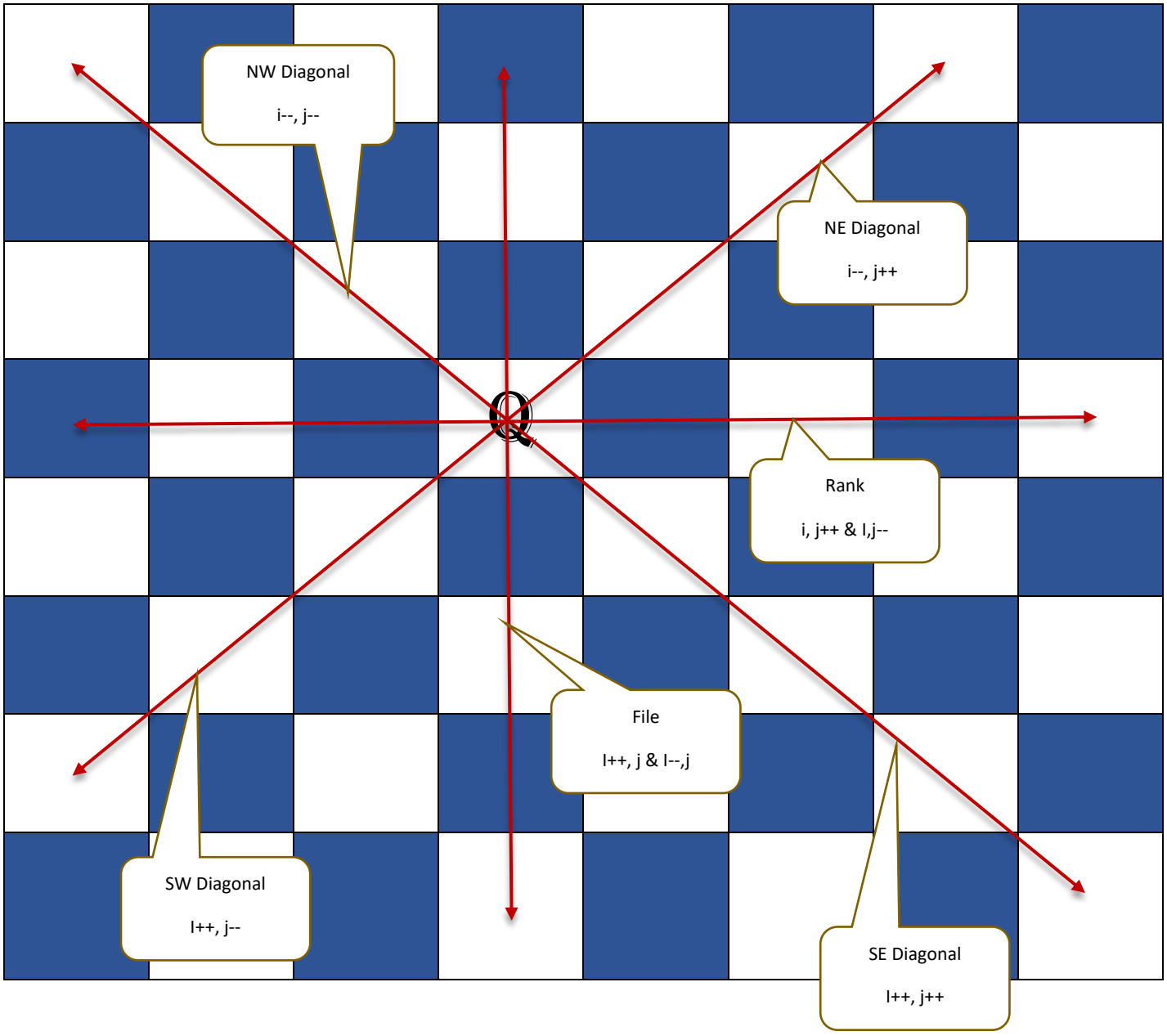
4. **<u>Backtracking</u>: The final Paradigm, being the simplest to get a grip on, but very perplexing to understand & wrap your head around! This essentially talks about exploring the options down a decision Tree, recursively, till you hit a dead-end, upon which you backtrack & unwind the stack and climb up the tree, to search for alternate solution(s) until you get one as per the requirements of the problem!**

This technique has been very popular with Maze problems (Yup, the one with the hamster or mouse in it!) & has been used heavily in Gaming! Also, they are formulated as Constraint Satisfaction Problems. Where, the constraints restrain the Tree of possibilities & the other paths or dead ends are pruned accordingly! There are Tree walking optimization techniques like BranchNBound which are very efficient computationally as well!!

- **<u>The N Queens Problem</u>: Place N Queens on an NxN Chess Board such that no Queen captures or attacks the others.**

Here's an illustration on the Constraints Composition itself!

An 8x8 Chess Board:



Notice that the above constraints are sufficient & necessary to enforce a set of solutions for the 8 Queens Problem.

Also, to me, this coincidently appears to bear a resemblance with the Union Jack itself! …. 😊

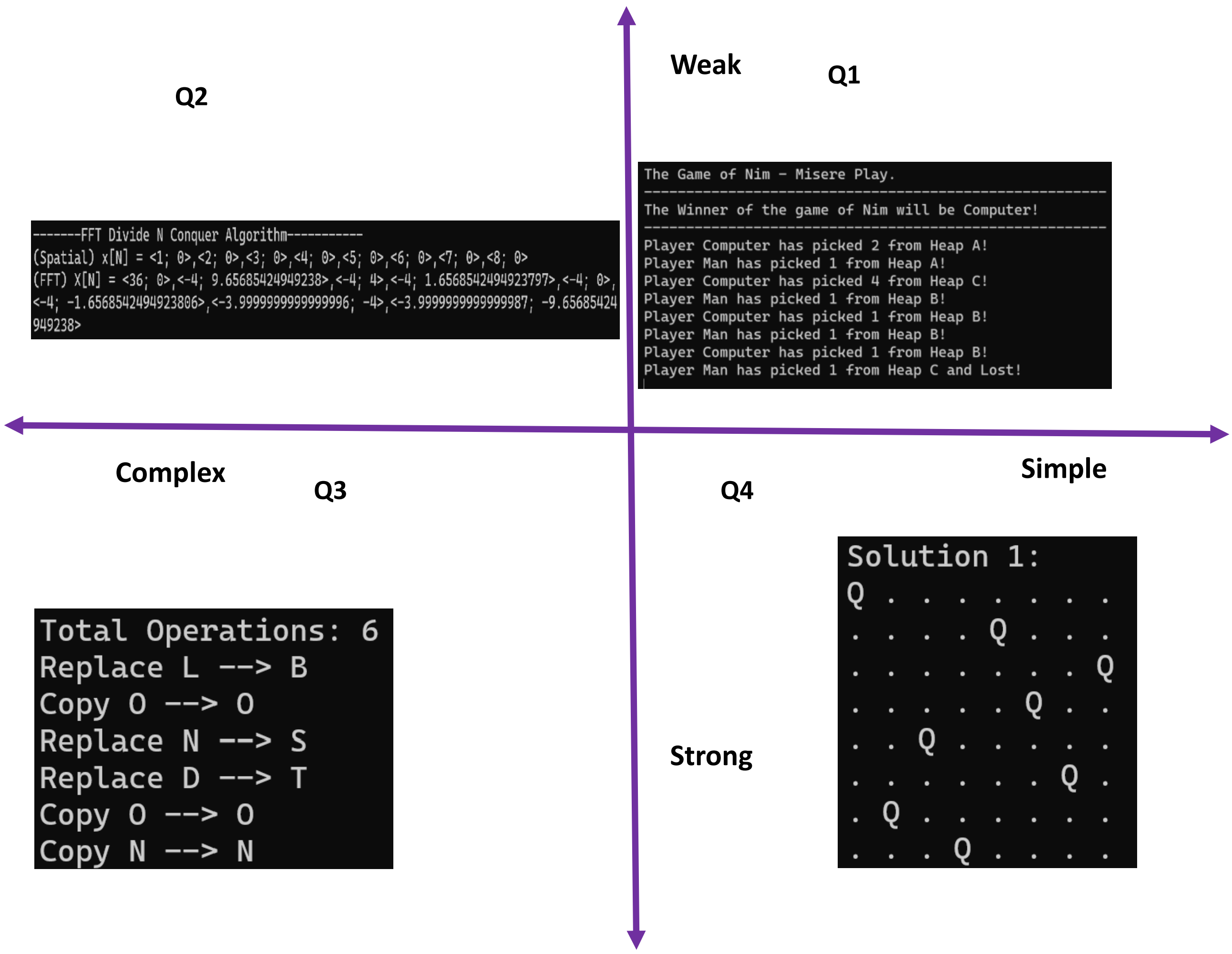
Here's some code to demonstrate the problem solution aesthetically:

```
public bool Solve(Square[][] board, int row, int turns)
        {
            if (turns == board.Length) //if the entire rows have been traversed
            {
                Board.Print(board); //print the solution
                return true; //true as the solution was found!
            }
            for (int c = 0; c < board.Length; c++) //iterate over the columns row-wise
(i.e. row-by-row)
            {
                //Check if all the Constraints are Satisfied or not?
                if (Piece.Place(board, row % board.Length, c)) //if the Queen can be
placed, then place it on the board!
                {
                    board[row % board.Length][c].Value = Piece.Name; //Place the Piece on
the Board!
                    if(Solve(board, row%board.Length + 1, turns + 1)) //recursively call
the solver to place the other Queens!
                        return true; //One Solution is found and Complete!
                    board[row % board.Length][c].Value = Board.VacantSquare; //If not
success, here, remove the Piece from the Board! (Mark it as vacant!)
                }
            }
            return false; //false if queens cannot be placed anymore! Hit a Dead-End!
        }
```

```
public override bool Place(Square[][] board, int row, int col)
        {
            //file wise
            for (int r = 0; r < board.Length; r++)
            {
                if (r == row)
                    continue;
                if (board[r][col].Value == Name)
                    return false;
            }
            //rank wise
            for (int c = 0; c < board.Length; c++)
            {
                if (c == col)
                    continue;
                if (board[row][c].Value == Name)
                    return false;
            }
            //NW Diagonal
            for (int r = row, c = col; r >= 0 && c >= 0; r--, c--)
            {
                if (!board.WithinBounds(r, c))
                    continue;
                if (board.WithinBounds(r, c) && board[r][c].Value == Name)
                    return false;
            }
            //SE Diagonal
            for (int r = row, c = col; r < board.Length && c < board.Length; r++, c++)
            {
                if (!board.WithinBounds(r, c))
                    continue;
                if (board.WithinBounds(r, c) && board[r][c].Value == Name)
                    return false;
            }
            //SW Diagonal
            for (int r = row, c = col; r < board.Length && c >= 0; r++, c--)
            {
                if (!board.WithinBounds(r, c))
                    return false;
                if (board.WithinBounds(r, c) && board[r][c].Value == Name)
                    return false;
            }
            //NE Diagonal
            for (int r = row, c = col; r >= 0 && c < board.Length; r--, c++)
            {
                if (!board.WithinBounds(r, c))
                    return false;
                if (board.WithinBounds(r, c) && board[r][c].Value == Name)
                    return false;
            }
            return true;
        }
```

<u>Conclusion</u>: The 4 Algorithmic Paradigms discussed here are critical to understanding how the Computing Machinery & Systems function! Computers are Ubiquitous these days, and to appreciate their beauty & significance we should always keep in mind what they were built for! It is undoubtedly true that Computers are the most Hardworking Problem Solvers on Earth these days!

Let's finally chart our Picture for the Fabric of Spacetime as we come to an end with the final impression of the 4 Quadrants of Computing as shown below:



"Computers have become Number Crunching Behemoths, but even the most Complex of Systems are built from the Simplest of Parts!! It's a bit fascinating to appreciate that even the most intricate fabric of Space-Time is but composed of the smallest of atoms or quarks as we all know!"

----- Bhaskar Rao.

~~~~~~
~~~~~~