Backtracking

Backtracking is similar to regular recursion in the sense that it **increments** one step at a time. The difference appears whenever you need to evalulate multiple alternatives, you evalutate **all** alternatives and choose the best.

---

51. N-Queens

[Leetcode]()
[Neetcode]()

Professor Solution:

```
Q = [1 .. r - 1]
n = size of table

def nqueen(Q, r):
	r = len(q)

	if r > n:
		print(Q)
	else:
		for j in range(n):
			legal = True

			for i in range(r):
				if Q[i] == j or Q[i] == j+r-1 or Q[i] == j-r+1:
					legal = False
					break

			if legal:
				Q.push(j)
				nqueen(Q, r+1)
				Q.pop()
```

Personal Backtracking Solution *(Leetcode)*

```
def solveNQueens(n: int) -> List[List[str]]:
	col = set()
	pDiag = set() #(row + column)
	nDiag = set() #(row - column)

	# Build empty board
	result = []
```

```python
        board = [["."] * n for _ in range(n)]

        # Backtracking function
        def nQueens(r):
                # Base case
                # If we've traversed whole board we add rows
                if r == n:
                        rowCopy = ["".join(row) for row in board]
                        result.append(rowCopy)
                        return

                for c in range(n):
                        # If the column is in a used row, diagonal or column skip
                        if c in col or (r + c) in pDiag or (r - c) in nDiag:
                                continue

                        # We add the current position to the sets
                        col.add(c)
                        pDiag.add(r+c)
                        nDiag.add(r-c)
                        board[r][c] = "Q"

                        # Call next row
                        nQueens(r+1)

                        # Reverse our backtracking
                        col.add(c)
                        pDiag.add(r+c)
                        nDiag.add(r-c)
                        board[r][c] = "Q"

        # Start backtracking at row 0
        nQueens(0)
        return result
```

The logic here is a queen piece can move **vertical**, **horizontal**, and **diagonal**. We can just simple check for pieces that aren't in any position a previous queen can visit.

We store all the previous piece's info in a set. We can check the row of the diagonal using a simple trick.

Positive Diagonals can be calculated by $Row + Column$
Negataive Diagonals can be calclulated by $Row - Column$

---

37. Sudoku Solver

[Leetcode]

Sudoku has 3 rules:

1. Each of 1-9 digits must occur exactly once in each row
2. Each of 1-9 digits must occur exactly once in each column
3. Each of 1-9 digits must occur exactly once in each of the 9x9 sub-boxes

Example Board:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Solved Board:

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Leetcode Solution:

```
def sudoku(Board: List[List[str]) -> None:
        # Sets for Sudoku's 3 rules
        row = [set() for _ in range(9)]
        col = [set() for _ in range(9)]
        boxes = [set() for _ in range(9)]

        # Create IDs for Sudoku rules that have items
        for i in range(9):
                for j in range(9):
                        if board[i][j] != '.':
                                num = int(board[i][j])
                                row[i].add(num)
                                col[j].add(num)
```

```
                                box_id = i // 3 * 3 + j // 3
                                boxes[box_id].add(num)

        # Recursive Function
        def sudoku(i, j):
                nonlocal solved
                # Travel down column then row
                # Set row
                new_i = i + (j+1) // 9
                # Set column
                new_j = (j+1) % 9

                # If at the last row and it's solved
                if i == 9:
                        solved = True
                        return

                # If the position is already filled skip
                if board[i][j] != '.':
                        sudoku(new_i, new_j)
                else:
                        # Try possible numbers
                        for num in range(1, 10):
                                # Check if the number is used following rules
                                box_id = i // 3 * 3 + j // 3
                                if num not in row[i] and num not in col[j] and num not
in boxes[box_id]:
                                        # If valid we add the item we created
                                        row[i].add(num)
                                        col[j].add(num)
                                        boxes[box_id].add(num)

                                        # Add the item to the board and move on
                                        board[i][j] = str(num)
                                        sudoku(new_i, new_j)

                                        # Backtrack if not solved
                                        if not solved:
                                                row[i].remove(num)
                                                col[j].remove(num)
                                                boxes[box_id].remove(num)
                                                board[i][j] = '.'

        # Call
        solved = False
        sudoku(0, 0)
```

HW2. Programming Question 1

Given a `n` & `k` and a chess board indicating a board of $n * n$ and k pieces to place. Output the different number of ways to place the piece on the chess board.

Pieces can be placed in `#` positions
Pieces cannot be placed in the same row or column as another

Example Input:
2 1
.#
#.

Output:
2

My Solution:

```
#!/usr/bin/env python

# Verify if the row and column are free
def isPositionFree(board: list, size: int, column: int) -> bool:
        # Navigate rows in the column
    for i in range(size):
        if board[i][column] == '*':
            return False

    return True


def getOptions(board: list, size: int, pieces: int, used: int, row: int) -> int:
        # If all pieces are used return 1
    if used == pieces:
        return 1
        # If traversed entire board stop
    if size == row:
        return 0

        # Options counter
    totalOptions = 0

        # Traverse column
    for column in range(size):
            # If the spot open and the position is free then place a piece
        if board[row][column] == '#' and isPositionFree(board, size, column):
            board[row][column] = '*'
            # Count the option and move onto the next row
            totalOptions += getOptions(board, size, pieces, used + 1, row + 1)
            # Backtrack the piece placed
            board[row][column] = '#'

        # TODO: Unsure verify this later
    totalOptions += getOptions(board, size, pieces, used, row + 1)
```

```python
    return totalOptions


def main():
    size, pieces = (int(x) for x in input().split())
    board = [list(input()) for _ in range(size)]

    print(getOptions(board, size, pieces, 0, 0))


if __name__ == "__main__":
    main()
```

1593. Split a String Into the Max Number of Unique Substrings

[Leetcode]()

Given string `s` find the max unique substrings.

Personal backtracking solution:

```python
def maxUniqueSplit(self, s: str) -> int:
        # Sets to store created substrings
        used = set()
        n = len(s)

        # Given index and current substring count
        def substring(i, count):
                nonlocal used
                ans = count

                # Base case
                if i >= n+1:
                        return 0

                # Loop through rest of string
                for j in range(i+1, n+1):
                        # If substring hasn't been created then continue
                        if s[i:j] not in used:
                                # Add substring to used
                                used.add(s[i:j])
                                # Call next option
                                ans = max(ans, substring(j, count+1))
                                # Backtrack
                                used.remove(s[i:j])

                return ans
```

```
        return substring(0, 0)
```