Backtracking

Advanced Programming

Spring 2024

Introduction

- Write a method permute that accepts a string as a parameter and outputs all possible rearrangements of the letters in that string. The arrangements may be output in any order.
- Example: permute("MARTY") outputs the following sequence of lines:

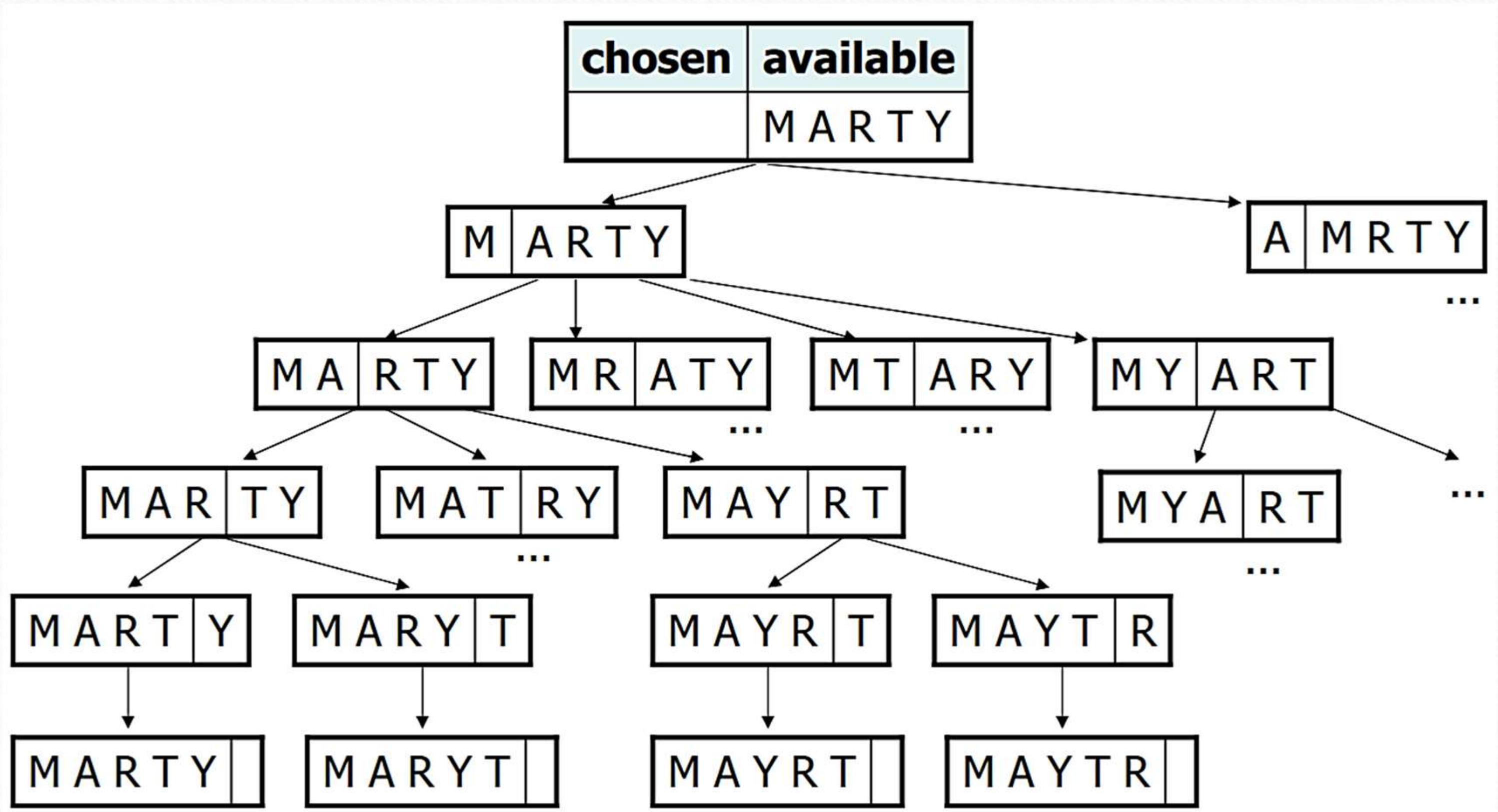
MARTY	MYRAT	ATYMR	RTMAY	TARMY	YMTAR
MARYT	MYRTA	ATYRM	RTMYA	TARYM	YMTRA
MATRY	MYTAR	AYMRT	RTAMY	TAYMR	YAMRT
MATYR	MYTRA	AYMTR	RTAYM	TAYRM	YAMTR
MAYRT	AMRTY	AYRMT	RTYMA	TRMAY	YARMT
MAYTR	AMRYT	AYRTM	RTYAM	TRMYA	YARTM
MRATY	AMTRY	AYTMR	RYMAT	TRAMY	YATMR
MRAYT	AMTYR	AYTRM	RYMTA	TRAYM	YATRM
MRTAY	AMYRT	RMATY	RYAMT	TRYMA	YRMAT
MRTYA	AMYTR	RMAYT	RYATM	TRYAM	YRMTA
MRYAT	ARMTY	RMTAY	RYTMA	TYMAR	YRAMT
MRYTA	ARMYT	RMTYA	RYTAM	TYMRA	YRATM
MTARY	ARTMY	RMYAT	TMARY	TYAMR	YRTMA
MTAYR	ARTYM	RMYTA	TMAYR	TYARM	YRTAM
MTRAY	ARYMT	RAMTY	TMRAY	TYRMA	YTMAR
MTRYA	ARYTM	RAMYT	TMRYA	TYRAM	YTMRA
MTYAR	ATMRY	RATMY	TMYAR	YMART	YTAMR
MTYRA	ATMYR	RATYM	TMYRA	YMATR	YTARM
MYART	ATRMY	RAYMT	TAMRY	YMRAT	YTRMA
MYATR	ATRYM	RAYTM	TAMYR	YMRTA	YTRAM

- Think of each permutation as a set of choices or decisions:
    - – Which character do I want to place first?
    - – Which character do I want to place second? – ...
    - – solution space: set of all possible sets of decisions to explore
- We want to generate all possible sequences of decisions.

```
for (each possible first letter):
    for (each possible second letter):
        for (each possible third letter):
            ...
            print!
```

- – This is called a depth-first search

Decision Trees



Backtracking

- Backtracking is a problem-solving algorithmic technique that involves finding a solution incrementally by trying different options and undoing them if they lead to a dead end.
- It is commonly used in situations where you need to explore multiple possibilities to solve a problem, like searching for a path in a maze or solving puzzles like Sudoku.
- When a dead end is reached, the algorithm backtracks to the previous decision point and explores a different path until a solution is found or all possibilities have been exhausted.
- Backtracking can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.

Backtracking steps

1. **Start with a problem**: Backtracking is often used for problems where you need to find all possible solutions, such as puzzles or optimization problems.
2. **Make a choice**: You make a choice at each step, trying out one of the options available.
3. **Explore**: Once you've made a choice, you explore further to see if it leads to a solution.
4. **Backtrack**: If the choice doesn't lead to a solution, you backtrack and try a different choice.
5. **Repeat**: You keep repeating this process until you find a solution or exhaust all possibilities.

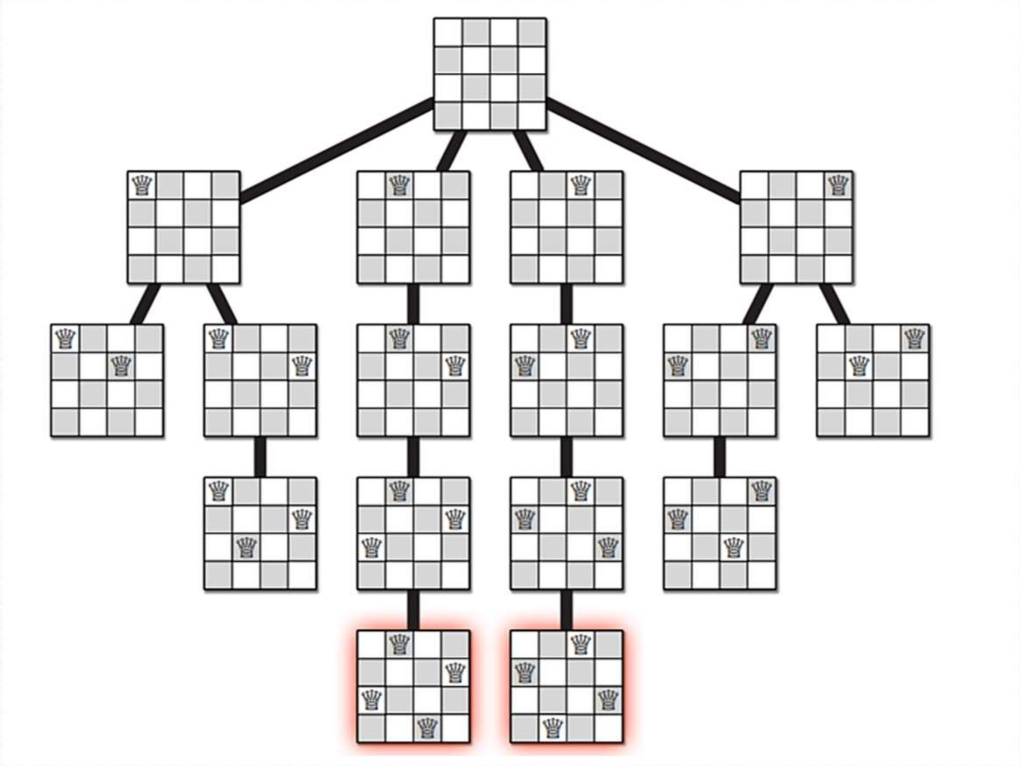
Solution to permutation problem

```
def permutations(a, idx =0, solutions = []):
    if idx == len(a)-1:
        solutions.append(''.join(a))
    for i in range(idx,len(a)):
        a[idx], a[i] = a[i], a[idx]
        permutations(a, idx+1, solutions) #generate all current possible solutions
        a[idx], a[i] = a[i], a[idx] #backtrack
    return solutions

print(permutations(list("ABC")))
```

Example: N Queens

- Put n queens on an $n \times n$ chessboard, so that no two queens are attacking each other.



N-Queen in python

```
N = 8 # (size of the chessboard)

def solveNQueens(board, row):
    if row == N:
        print(board)
        return True
    for col in range(N):
        if isSafe(board, row, col):
            board[row][col] = 1
            if solveNQueens(board, row + 1):
                return True
            board[row][col] = 0
    return False

def isSafe(board, row, col):
    for x in range(row):
        if board[x][col] == 1:
            return False
    for x, y in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[x][y] == 1:
            return False
    for x, y in zip(range(row, -1, -1), range(col, 1, N)):
        if board[x][y] == 1:
            return False
    return True

board = [[0 for x in range(N)] for y in range(N)]
if not solveNQueens(board, 0):
    print("No solution found")
```

Step 1: Did we reach to a solution?

Step 2: Generate all <u>current</u> possible choices

Step 3: Check if constraints are satisfied

Step 4: Add to solution path

Step 5: recurse

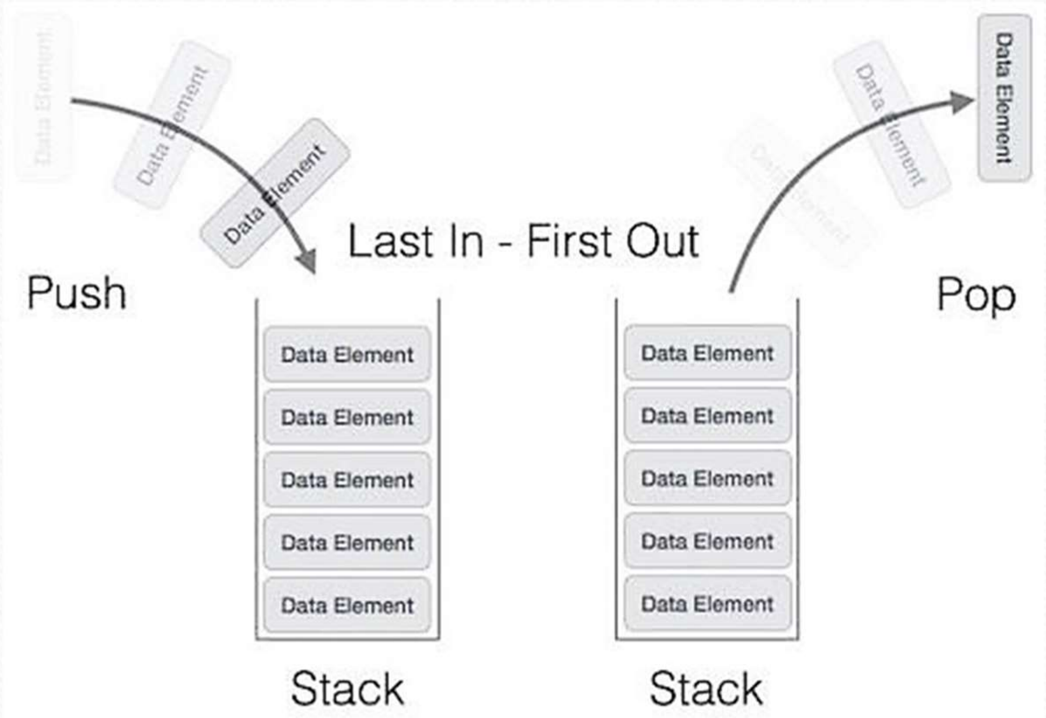
Step 6: Backtrack if didn't lead to solution

Checking constraints for current partial solution

Not in same column in previous rows

Not threatened diagonally

Brief introduction to Stack



- We can create it very easily!!!!
  - list1 = [ ]
  - list1.append(1)
  - list1.append(2)
  - list1.pop( )
  - list1.append(3)
  - list1.pop( )

Escape Maze using backtracking & stack

High level algorithm

1. Choose the initial cell, mark it as visited and push it to the stack
2. While the stack is not empty
    1. Pop a cell from the stack and make it a current cell
    2. If the cell is the destination cell

        return
    3. If the current cell has any neighbors which have not been visited
        1. Push the current cell to the stack
        2. Choose one of the unvisited neighbors
        3. Mark the chosen cell as visited and push it to the stack