Backtracking

Yiwei Gu

Content

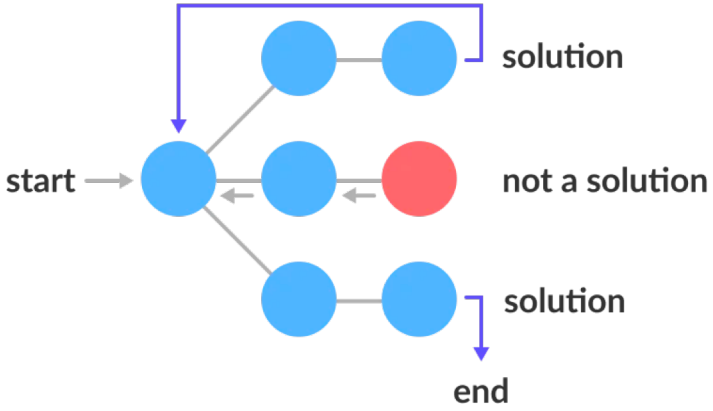
- Basic Concepts
- Backtracking Logic Pattern
- Example Questions

What is Backtracking?

Wiki:

Backtracking can be defined as a general algorithmic technique that considers **searching every possible combination** in order to solve a computational problem.

- A Brute Force approach
- State-Space Tree
- Recursion (DFS)



Problems in Backtracking

Generally, every **constraint satisfaction problem** satisfies two conditions below can be solved by Backtracking.

1. Incrementally build candidate to the solution
2. Candidate can be abandoned if it cannot possibly reach a valid solution

Three types of problem in backtracking

1. Enumeration Problem – in this, we find **all feasible** solutions.
2. Decision Problem – in this, we search for **a feasible** solution
3. Optimization Problem – in this, we search for **the best** solution

Backtracking Logic Pattern

Logic Key Components:

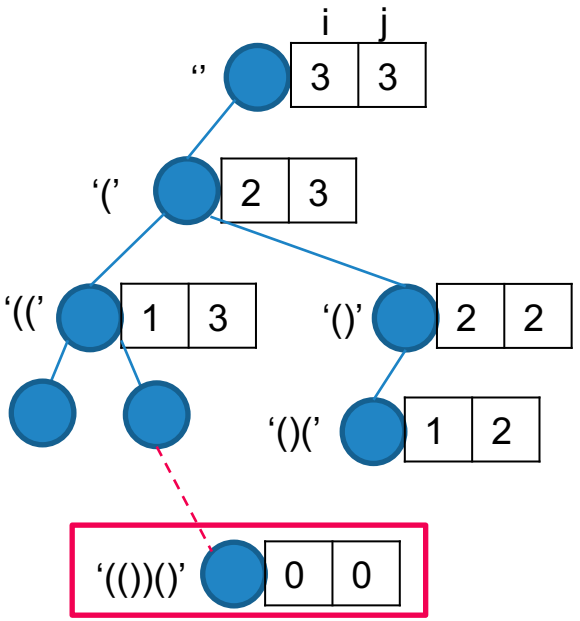
1. Choices - Decisions to change a state of sub-problems
2. Constraints – Rules necessary to validate choices
3. Goals – Define success of the sub-problems

Example: LC 22 Generate Parentheses

Given n pairs of parentheses (), write a function to *generate all combinations of well-formed parentheses*.

Input: n = 3 **Output:** ["((()))","(()())","(())()","()(())","()()()"]

Choice: use 1 left out i ; use 1 right out j
Constraint: could add right if i < j
Goal: a string of n well-formed parentheses



LC22 – Generate Parentheses (Medium)

Given `n` pairs of parentheses, write a function to *generate all combinations of well-formed parentheses*.

Example 1:

```
Input: n = 3
Output: ["((()))","(()())","(())()","()(())","()()()"]
```

Example 2:

```
Input: n = 1
Output: ["()"]
```

```python
def generateParenthesis(self, n: int) -> List[str]:

    def helper(curr_pl, i, j, res):
        if i == 0 and j == 0:
            res.append(''.join(curr_pl))
            return
        ## add left parenthesis
        if i > 0:
            curr_pl.append('(')
            helper(curr_pl, i-1,j, res)
            curr_pl.pop()
        ## add right parenthesis
        if i < j:
            curr_pl.append(')')
            helper(curr_pl,i,j-1,res)
            curr_pl.pop()
    res = []
    helper([], n, n, res)

    return res
```

Time Complexity: O(2^(2N))
Space Complexity: O((num of valid solution)*N)

LC17 – Letter Combinations of a Phone Number (Medium)

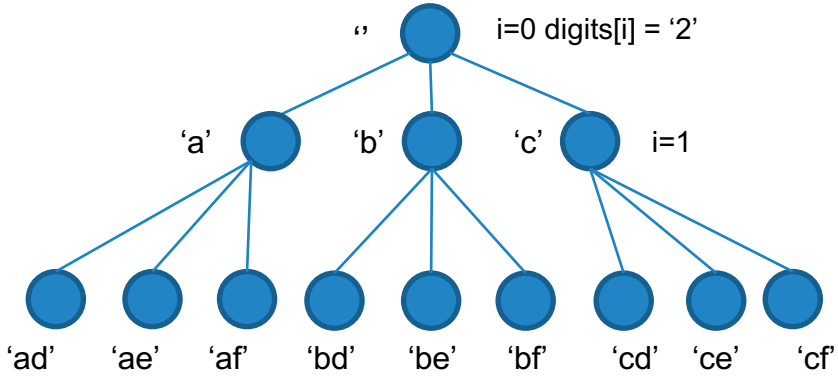
Given a string containing digits from `2-9` inclusive, return all possible letter combinations that the number could represent. Return the answer in **any order**.

A mapping of digits to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.

Example 1:

```
Input: digits = "23"
Output: ["ad","ae","af","bd","be","bf","cd","ce","cf"]
```

Choice: possible letter to add
Constraint: N/A
Goal: a string of possible letter combination



LC17 – Letter Combinations of a Phone Number (Medium)

Given a string containing digits from `2-9` inclusive, return all possible letter combinations that the number could represent. Return the answer in **any order**.

A mapping of digits to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.

Example 1:

```
Input: digits = "23"
Output: ["ad","ae","af","bd","be","bf","cd","ce","cf"]
```

```python
def letterCombinations(self, digits: str) -> List[str]:

    letter_set = {'2':'abc','3':'def','4':'ghi','5':'jkl','6':'mno'\
                  ,'7':'pqrs', '8':'tuv', '9':'wxyz'}

    def helper(digits, i, curr_lc, res):
        if len(digits) == 0:
            return
        elif i == len(digits):
            res.append(curr_lc[:])
        else:
            curr_digit = digits[i]
            curr_can = letter_set[curr_digit]
            for ch in curr_can:
                curr_lc = curr_lc+ch
                helper(digits, i+1, curr_lc, res)
                curr_lc = curr_lc[:-1]
    res = []
    helper(digits,0,'',res)
    return res
```

Time Complexity: O(3^N) ~ exponential
Space Complexity for stack: O(3^N*N)

LC39 – Combination Sum (Medium)

Given an array of **distinct** integers `candidates` and a target integer `target`, return *a list of all **unique combinations** of* `candidates` *where the chosen numbers sum to* `target`. You may return the combinations in **any order**.

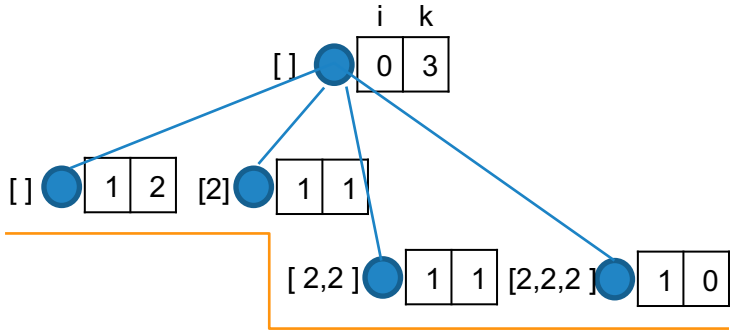
The **same** number may be chosen from `candidates` an **unlimited number of times**. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

It is **guaranteed** that the number of unique combinations that sum up to `target` is less than `150` combinations for the given input.

Example 1:

```
Input: candidates = [2,3,6,7], target = 7
Output: [[2,2,3],[7]]
Explanation:
2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used
multiple times.
7 is a candidate, and 7 = 7.
These are the only two combinations.
```

Choice: add candidate[i] k times
Constraint: sum of the combination <= target
Goal: a combinations satisfied conditions



LC39 – Combination Sum (Medium)

Given an array of **distinct** integers `candidates` and a target integer `target`, return *a list of all **unique combinations** of* `candidates` *where the chosen numbers sum to* `target`. You may return the combinations in **any order**.

The **same** number may be chosen from `candidates` an **unlimited number of times**. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

It is **guaranteed** that the number of unique combinations that sum up to `target` is less than `150` combinations for the given input.

Example 1:

```
Input: candidates = [2,3,6,7], target = 7
Output: [[2,2,3],[7]]
Explanation:
2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used
multiple times.
7 is a candidate, and 7 = 7.
These are the only two combinations.
```

```python
def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:

    def helper(cand_list, i, curr_list, curr_sum, target, res):
        if curr_sum == target:
            res.append(curr_list[:])
        elif curr_sum > target:
            return
        elif i >= len(cand_list):
            return
        else:
            residue = target - curr_sum
            k = residue // cand_list[i]
            for j in range(k+1):
                curr_list.extend([cand_list[i]]*j)
                curr_sum = curr_sum + cand_list[i]*j
                helper(cand_list, i+1 , curr_list, curr_sum, target, res)
                for m in range(j):
                    curr_list.pop()
                curr_sum = curr_sum - cand_list[i]*j


    res = []
    helper(candidates, 0, [], 0, target, res)
    return res
```

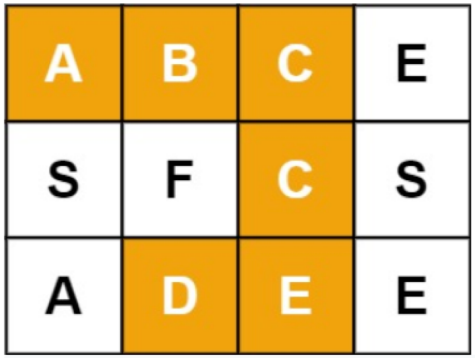
Time Complexity: O(k^N) ~ exponential
Space Complexity: O((num of valid solution)*N)

LC79 – Word Search (Medium)

Given an `m x n` grid of characters `board` and a string `word`, return `true` *if* `word` *exists in the grid*.

The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

Example 1:



Input: board = [["A","B","C","E"],["S","F","C","S"], ["A","D","E","E"]], word = "ABCCED"
Output: true

Choice: left, right, up, down
Constraint:
1. boundary restriction
2. no back path

Goal: match letter of “word” in sequence

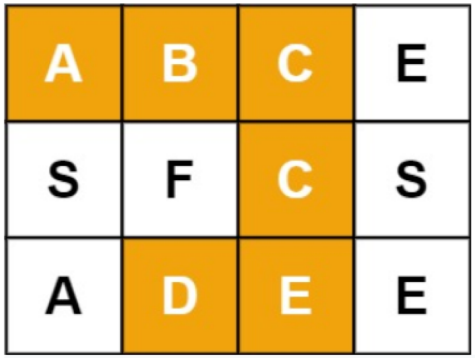
```
def helper(i,j,k):
    if i < 0 or i >= ROW:
        return False
    elif j < 0 or j >= COL:
        return False
    elif (i,j) in visited_set:
        return False
    elif board[i][j] != word[k]:
        return False
    elif k == len(word)-1 and board[i][j] == word[k]:
        return True
    else:
        visited_set.add((i,j))
        k+=1
        up_res = helper(i-1,j,k)
        down_res = helper(i+1, j,k)
        left_res = helper(i,j-1,k)
        right_res = helper(i, j+1,k)
        visited_set.remove((i,j))
        k-=1
        return up_res or down_res or left_res or right_res
```

LC79 – Word Search (Medium)

Given an `m x n` grid of characters `board` and a string `word`, return `true` *if* `word` *exists in the grid*.

The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

Example 1:



Input: board = [["A","B","C","E"],["S","F","C","S"], ["A","D","E","E"]], word = "ABCCED"
Output: true

```python
def exist(self, board: List[List[str]], word: str) -> bool:
    ROW = len(board)
    COL = len(board[0])
    visited_set = set()
    def helper(i,j,k):
          ...

    for i in range(ROW):
        for j in range(COL):
            chk_res = helper(i,j,0)
            if chk_res == True:
                return True

    return False
```

Time Complexity: O(m*n*3^L)
Space Complexity: O(L)

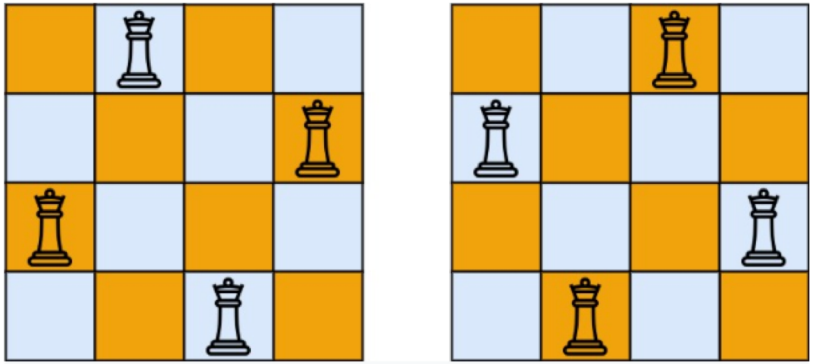
LC51 – N Queens (Hard)

The **n-queens** puzzle is the problem of placing `n` queens on an `n x n` chessboard such that no two queens attack each other.

Given an integer `n`, return *all distinct solutions to the **n-queens puzzle***. You may return the answer in **any order**.

Each solution contains a distinct board configuration of the n-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space, respectively.

Example 1:

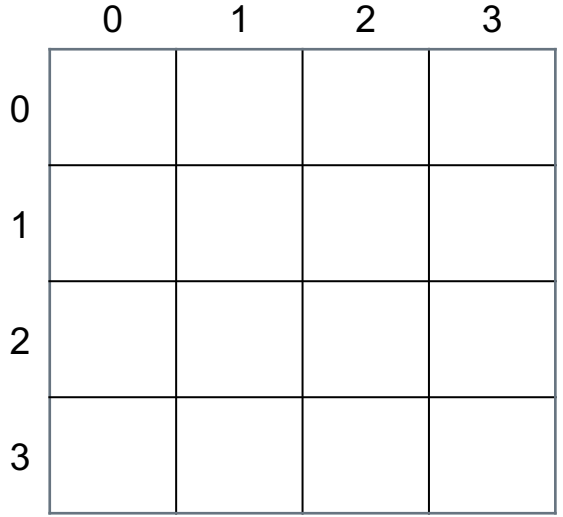


```
Input: n = 4
Output: [[".Q..","...Q","Q...","..Q."],
["..Q.","Q...","...Q",".Q.."]]
```

Choice: add a queen in i's row

Constraint: no attach from previous added queens

1. horizontal i
2. vertical j
3. positive diagonal i+j
4. negative diagonal i-j



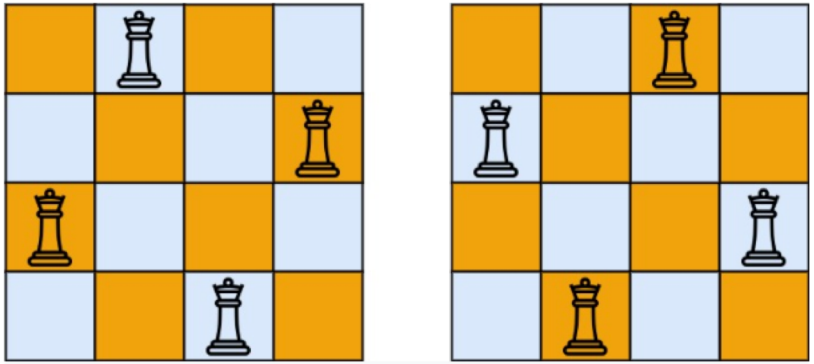
LC51 – N Queens (Hard)

The **n-queens** puzzle is the problem of placing `n` queens on an `n x n` chessboard such that no two queens attack each other.

Given an integer `n`, return *all distinct solutions to the **n-queens puzzle***. You may return the answer in **any order**.

Each solution contains a distinct board configuration of the n-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space, respectively.

Example 1:



Input: n = 4
Output: [[".Q..","...Q","Q...","..Q."],
["..Q.","Q...","...Q",".Q.."]]

```
def helper(n,i,col_used, posd_used, negd_used, exist_qs, res):
    if i == n:
        res.append(exist_qs[:])
    else:
        for j in range(n):
            #col
            if j in col_used:
                pass
            #posDiag
            elif i+j in posd_used:
                pass
            #negDiag
            elif i-j in negd_used:
                pass
            else:
                col_used.add(j)
                posd_used.add(i+j)
                negd_used.add(i-j)
                exist_qs.append(label_queen(n,j))
                helper(n,i+1, col_used, posd_used, negd_used,exist_qs,res)
                col_used.remove(j)
                posd_used.remove(i+j)
                negd_used.remove(i-j)
                exist_qs.pop()
```

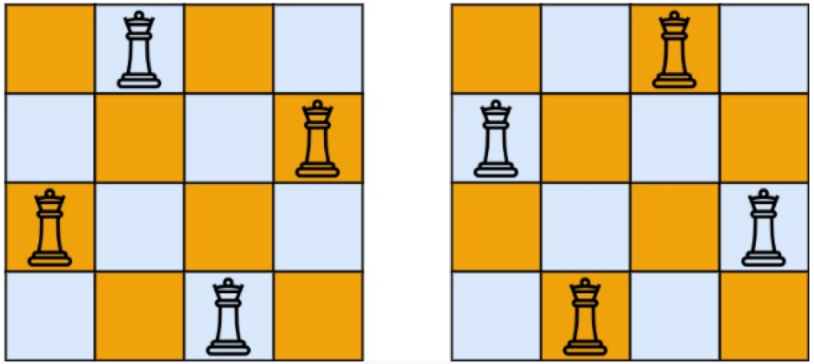
LC51 – N Queens (Hard)

The **n-queens** puzzle is the problem of placing `n` queens on an `n x n` chessboard such that no two queens attack each other.

Given an integer `n`, return *all distinct solutions to the **n-queens puzzle***. You may return the answer in **any order**.

Each solution contains a distinct board configuration of the n-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space, respectively.

Example 1:



Input: n = 4
Output: [[".Q..","...Q","Q...","..Q."],
["..Q.","Q...","...Q",".Q.."]]

```
def solveNQueens(self, n: int) -> List[List[str]]:

    def label_queen(n,j):
        """
        generate string of dots and label col of positioned queen
        """
        dots = ['.' for k in range(n)]
        dots[j] = 'Q'
        return ''.join(dots)

    def helper(n,i,col_used, posd_used, negd_used, exist_qs, res):
            ...

    res = []
    exist_qs = []
    col_used = set()
    posd_used = set()
    negd_used = set()
    helper(n,0, col_used, posd_used, negd_used, exist_qs,res)

    return res
```

Time Complexity:
lower bound - O(N*(N-3)*(N-6)…) > exponential

Space Complexity: O(N^2)

Homework

- LC90 Subsets II
- LC40 Combination Sum II
- LC131 Palindrome Partitioning
- LC93 Restore IP Addresses
- LC401 Binary Watch
- LC37 Sudoku Solver (optional)