LeetCode Training Day 25 Backtracking

Backtracking is a DFS pattern, which you start with all possible starting points, running in recursive function, search all possible position in the next steps, check the validity, and return if you already find the answer or not possible to find the answer.

On the search you should be aware of two things, one is to skip the duplication, another is to break out earlier if you find such choice may be invalid, both will improve performance. To skip the duplication, we will use visited flag same as we traverse graph, but unlike in graph, the visited flag should be reset when we are back from the search in deep levels. Specially if we do not care the sequence of a selection, we use bit map to track the visited item.

In many case backing tracking problem ask you to return all the valid answers, so in the recursion, we need to prepare a bag to collect the result.

## 17. 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 digit to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.

A picture containing electronics, calculator, hand, orange

Description automatically generated

**Example 1:**

**Input:** digits = "23"

**Output:** ["ad","ae","af","bd","be","bf","cd","ce","cf"]

**Example 2:**

**Input:** digits = ""

**Output:** []

**Example 3:**

**Input:** digits = "2"

**Output:** ["a","b","c"]

**Constraints:**

* 0 <= digits.length <= 4
* digits[i] is a digit in the range ['2', '9'].

### Analysis:

This is simplest form of backtracking, we start with number and iterate all letter in the number then come to next number. We track what is the string currently in search and when digits finish, we simply store current string into bag of result.

/// <summary>

/// Leet code #17. Letter Combinations of a Phone Number

/// </summary>

void LeetCodeDFS::letterCombinations(string& digits, string &path,

unordered\_map<char, string>& phone\_keyboard, vector<string> &result)

{

if (path.size() == digits.size())

{

if (!path.empty()) result.push\_back(path);

return;

}

char digit = digits[path.size()];

string target\_str = phone\_keyboard[digit];

for (char ch : target\_str)

{

path.push\_back(ch);

letterCombinations(digits, path, phone\_keyboard, result);

path.pop\_back();

}

}

/// <summary>

/// Leet code #17. 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.

///

/// A mapping of digit to letters (just like on the telephone buttons)

/// is given below. Note that 1 does not map to any letters.

///

/// Example:

///

/// Input: "23"

/// Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

/// Note:

/// Although the above answer is in lexicographical order, your answer

/// could be in any order you want.

/// </summary>

vector<string> LeetCodeDFS::letterCombinations(string digits)

{

unordered\_map<char, string> phone\_keyboard =

{

{ '2', "abc" },{ '3', "def" },{ '4', "ghi" },{ '5', "jkl" },

{ '6', "mno" },{ '7', "pqrs" },{ '8', "tuv" },{ '9', "wxyz" },

{ '\*', "+" }

};

string path;

vector<string> result;

letterCombinations(digits, path, phone\_keyboard, result);

return result;

}

## 79. 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:**

Calendar

Description automatically generated

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

**Output:** true

**Example 2:**

Calendar

Description automatically generated

**Input:** board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "SEE"

**Output:** true

**Example 3:**

A picture containing text, crossword puzzle

Description automatically generated

**Input:** board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCB"

**Output:** false

**Constraints:**

* m == board.length
* n = board[i].length
* 1 <= m, n <= 6
* 1 <= word.length <= 15
* board and word consists of only lowercase and uppercase English letters.

**Follow up:** Could you use search pruning to make your solution faster with a larger board?

### Analysis:

We search from the cell, which is starting of word, and move to next position for the next character in the word, if we cannot find such character, we return to last level. We mark the cell during the search and release it when we return the recursive function.

/// <summary>

/// Leet code #79. Word Search

/// </summary>

bool LeetCodeDFS::wordSearch(vector<vector<char>>& board,

vector<vector<bool>>& flag, string word, int x, int y, int pos)

{

if (pos == word.size()) return true;

if ((x < 0) || (x == board.size()) || (y < 0) || (y == board[0].size()))

{

return false;

}

if (board[x][y] != word[pos])

{

return false;

}

if (flag[x][y] == true) return false;

flag[x][y] = true;

if (wordSearch(board, flag, word, x - 1, y, pos + 1) ||

wordSearch(board, flag, word, x + 1, y, pos + 1) ||

wordSearch(board, flag, word, x, y - 1, pos + 1) ||

wordSearch(board, flag, word, x, y + 1, pos + 1))

{

return true;

}

flag[x][y] = false;

return false;

}

/// <summary>

/// Leet code #79. Word Search

///

/// Given a 2D board and a word, find if the word exists in the grid.

/// The word can be constructed from letters of sequentially adjacent

/// cell, where "adjacent" cells are those

/// horizontally or vertically neighboring. The same letter cell may not

/// be used more than once.

/// For example,

/// Given board =

/// [

/// ['A','B','C','E'],

/// ['S','F','C','S'],

/// ['A','D','E','E']

/// ]

/// word = "ABCCED", -> returns true,

/// word = "SEE", -> returns true,

/// word = "ABCB", -> returns false.

/// </summary>

bool LeetCodeDFS::wordSearch(vector<vector<char>>& board, string word)

{

vector<vector<bool>> flag(board.size(), vector<bool>(board[0].size()));

for (size\_t x = 0; x < board.size(); x++)

{

for (size\_t y = 0; y < board[0].size(); y++)

{

if (wordSearch(board, flag, word, x, y, 0))

{

return true;

}

}

}

return false;

}

## 39. 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.

**Example 2:**

**Input:** candidates = [2,3,5], target = 8

**Output:** [[2,2,2,2],[2,3,3],[3,5]]

**Example 3:**

**Input:** candidates = [2], target = 1

**Output:** []

**Constraints:**

* 1 <= candidates.length <= 30
* 1 <= candidates[i] <= 200
* All elements of candidates are **distinct**.
* 1 <= target <= 500

### Analysis:

First, we want combination, for example 2, 3 and 3, 2 are same, we sort the numbers first, this is to avoid duplication. On every level of search, we move from the number same as last round to right, which means keep search number incremental. Second we keep on adding sum, when we reach or exceed target we stop search further.

/// <summary>

/// Leet code #39. Combination Sum

/// </summary>

void LeetCodeDFS::combinationSum(vector<int>& candidates, int target, int index,

vector<int>& path, vector<vector<int>>&result)

{

if (target == 0)

{

if (!path.empty()) result.push\_back(path);

return;

}

for (size\_t i = index; i < candidates.size(); i++)

{

if (candidates[i] > target) break;

target -= candidates[i];

path.push\_back(candidates[i]);

combinationSum(candidates, target, i, path, result);

target += candidates[i];

path.pop\_back();

}

}

/// <summary>

/// Leet code #39. Combination Sum

/// Given a set of candidate numbers (C) and a target number (T), find all

/// unique combinations in C where the candidate numbers sums to T.

/// The same repeated number may be chosen from C unlimited number of times.

/// Note:

/// All numbers (including target) will be positive integers.

/// The solution set must not contain duplicate combinations.

/// For example, given candidate set [2, 3, 6, 7] and target 7,

/// A solution set is:

/// [

/// [7],

/// [2, 2, 3]

/// ]

/// </summary>

vector<vector<int>> LeetCodeDFS::combinationSum(vector<int>& candidates, int target)

{

vector<int> path;

vector<vector<int>> result;

sort(candidates.begin(), candidates.end());

combinationSum(candidates, target, 0, path, result);

return result;

}

## 40. Combination Sum II

Medium

Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sum to target.

Each number in candidates may only be used **once** in the combination.

**Note:** The solution set must not contain duplicate combinations.

**Example 1:**

**Input:** candidates = [10,1,2,7,6,1,5], target = 8

**Output:**

[

[1,1,6],

[1,2,5],

[1,7],

[2,6]

]

**Example 2:**

**Input:** candidates = [2,5,2,1,2], target = 5

**Output:**

[

[1,2,2],

[5]

]

**Constraints:**

* 1 <= candidates.length <= 100
* 1 <= candidates[i] <= 50
* 1 <= target <= 30

### Analysis:

We sort the number, and on each step we search from a number with index + 1 in last level, and because we only want to use the number once and the number set contains duplicate, we should skip duplicate number in each round.

/// <summary>

/// Leet code #40. Combination Sum II

/// </summary>

void LeetCodeDFS::combinationSum2(vector<int>& candidates, int target, int index,

vector<int>& path, vector<vector<int>>&result)

{

if (target == 0)

{

if (!path.empty()) result.push\_back(path);

return;

}

for (size\_t i = index; i < candidates.size(); i++)

{

if (candidates[i] > target) break;

if ((i > (size\_t)index) && (candidates[i] == candidates[i - 1])) continue;

target -= candidates[i];

path.push\_back(candidates[i]);

combinationSum2(candidates, target, i + 1, path, result);

target += candidates[i];

path.pop\_back();

}

}

/// <summary>

/// Leet code #40. Combination Sum II

///

/// Medium

///

/// Given a collection of candidate numbers (candidates) and a target

/// number (target), find all unique combinations in candidates where

/// the candidate numbers sums to target.

///

/// Each number in candidates may only be used once in the combination.

///

/// Note:

///

/// All numbers (including target) will be positive integers.

/// The solution set must not contain duplicate combinations.

///

/// Example 1:

/// Input: candidates = [10,1,2,7,6,1,5], target = 8,

/// A solution set is:

/// [

/// [1, 7],

/// [1, 2, 5],

/// [2, 6],

/// [1, 1, 6]

/// ]

///

/// Example 2:

///

/// Input: candidates = [2,5,2,1,2], target = 5,

/// A solution set is:

/// [

/// [1,2,2],

/// [5]

/// ]

/// </summary>

vector<vector<int>> LeetCodeDFS::combinationSum2(vector<int>& candidates,

int target)

{

vector<int> path;

vector<vector<int>> result;

sort(candidates.begin(), candidates.end());

combinationSum2(candidates, target, 0, path, result);

return result;

}

**139. Word Break**

Medium

Given a string s and a dictionary of strings wordDict, return true if s can be segmented into a space-separated sequence of one or more dictionary words.

**Note** that the same word in the dictionary may be reused multiple times in the segmentation.

**Example 1:**

**Input:** s = "leetcode", wordDict = ["leet","code"]

**Output:** true

**Explanation:** Return true because "leetcode" can be segmented as "leet code".

**Example 2:**

**Input:** s = "applepenapple", wordDict = ["apple","pen"]

**Output:** true

**Explanation:** Return true because "applepenapple" can be segmented as "apple pen apple".

Note that you are allowed to reuse a dictionary word.

**Example 3:**

**Input:** s = "catsandog", wordDict = ["cats","dog","sand","and","cat"]

**Output:** false

**Constraints:**

* 1 <= s.length <= 300
* 1 <= wordDict.length <= 1000
* 1 <= wordDict[i].length <= 20
* s and wordDict[i] consist of only lowercase English letters.
* All the strings of wordDict are **unique**.