# LeetCode Day 21 Simple Backtracking

Backtracking is an algorithm which is based on DFS to exhaust all the possible candidate and find the answer to the problem.

This time complexity for backtracking is O(n!) or O(e^n), n is the depth of search path, so you should use it very carefully.

## When to use Backtracking

The following are the typical patterns for a backtracking:

1. Fina all the answers, to do so, you need to exhaust all the candidates.
2. The total length of the path is less than 16, assume all the possible choice at each step is only 2 or 4, it is affordable.

## How it works

The best way to do the backtracking is to use a recursive function. Every time we call the recursive function, we go one step further in the search and every time we exit the function, we become one step back.

In the recursive function, the first thing we need to do is to check if we already find a complete answer, if yes, we will add to the result set and return. If the search is not complete, we will exhaust all the possible candidates, validate them for example, for example, check if this is a duplication, or invalid choice. After picking the possible candidate, we will mark them or add it to the accumulated sum and call the recursive function and go further step. Please notice that if we get back from the recursive function, regardless it is a successful search or not, we have to recover environment for the current candidate, for example, unmark it or deducted from the accumulated sum and try the next candidate. After all the candidates are exhaust in the current step, we exit the function and go one step back.

## Parameter of the recursive function

One interesting discussion here is that what you would like to pass to the recursive function, assuming the function is stateless. Obviously here are something you can consider:

1. Original data, such as the maze board or number set.
2. The partial search path which indicate where you are.
3. The visited cache which allow you to quickly prune the search.
4. Some complete flag which help you to finish the search.
5. The result set where you want to keep track the complete 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.

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.

### Analysis:

We can use DFS or BFS to get the answers, here we choose the DFS to search. Please notice that we should put the phone number mapping out of the recursive function.

/// <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;

}

## 22. Generate Parentheses

Medium

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

For example, given *n* = 3, a solution set is:

[

"((()))",

"(()())",

"(())()",

"()(())",

"()()()"

]

### Analysis:

We iterate on a string length of 2 \* n and need to keep the string valid during the process. To do so, we will keep track on the number of left brackets and right brackets.

/// <summary>

/// Leet code #22. Generate Parentheses

/// </summary>

void LeetCodeDFS::generateParenthesis(string &path, int n, int left,

int right, vector<string> &result)

{

if (path.size() == 2 \* n)

{

result.push\_back(path);

return;

}

for (size\_t i = 0; i < 2; i++)

{

if ((i == 0) && (left >= n)) continue;

if ((i == 1) && (left <= right)) continue;

if (i == 0)

{

left++;

path.push\_back('(');

}

else

{

right++;

path.push\_back(')');

}

generateParenthesis(path, n, left, right, result);

if (i == 0) left--;

else right--;

path.pop\_back();

}

}

/// <summary>

/// Leet code #22. Generate Parentheses

///

/// Given n pairs of parentheses, write a function to generate all

/// combinations of well-formed parentheses.

/// For example, given n = 3, a solution set is:

/// [

/// "((()))",

/// "(()())",

/// "(())()",

/// "()(())",

/// "()()()"

/// ]

/// </summary>

vector<string> LeetCodeDFS::generateParenthesis(int n)

{

vector<string> result;

string path;

int left = 0, right = 0;

generateParenthesis(path, n, left, right, result);

return result;

}

## 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 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]

]

### Analysis:

For combination, the key is to avoid duplication, so you need to sort the numbers, when you select numbers, always keep ascending order. Since there are duplication in the numbers, at any position, if number x is selected, next pick should be greater than x.

/// <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;

}

## 79. Word Search

Medium

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.

**Example:**

board =

[

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

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

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

]

Given word = "**ABCCED**", return **true**.

Given word = "**SEE**", return **true**.

Given word = "**ABCB**", return **false**.

### Analysis:

We start from any place in the board and walk through all the directions. We recursive call 4 directions, and if any direction matches the word, we return true. When you use sub-search condition combination, please be careful, say if it is an OR condition, when the first condition satisfied, it will not evaluate the second condition. But in this case it is exactly what we want.

/// <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;

bool found = 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))

{

found = true;

}

flag[x][y] = false;

return found;

}

/// <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;

}

## [131. Palindrome Partitioning](https://leetcode.com/problems/palindrome-partitioning/)

Medium

Given a string s, partition s such that every substring of the partition is a **palindrome**. Return all possible palindrome partitioning of s.

**Example 1:**

**Input:** s = "aab"

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

**Example 2:**

**Input:** s = "a"

**Output:** [["a"]]

**Constraints:**

* 1 <= s.length <= 16
* s contains only lowercase English letters.

/// <summary>

/// Leet code #131. Palindrome Partitioning

/// </summary>

void LeetCodeDFS::partition(string& s, vector<string> &arr, size\_t start, vector<vector<int>>

&dp, vector<vector<string>> &result)

{

if (start == s.size())

{

result.push\_back(arr);

}

for (size\_t end = start; end < s.size(); end++)

{

if (dp[start][end] == 0) continue;

arr.push\_back(s.substr(start, end - start + 1));

partition(s, arr, end + 1, dp, result);

arr.pop\_back();

}

}

/// <summary>

/// Leet Code 131. Palindrome Partitioning

///

/// Medium

///

/// Given a string s, partition s such that every substring of the

/// partition is a palindrome.

/// Return all possible palindrome partitioning of s.

///

/// Example 1:

/// Input: s = "aab"

/// Output: [["a","a","b"],["aa","b"]]

///

/// Example 2:

/// Input: s = "a"

/// Output: [["a"]]

///

/// Constraints:

/// 1. 1 <= s.length <= 16

/// 2. s contains only lowercase English letters.

/// </summary>

vector<vector<string>> LeetCodeDFS::partition(string s)

{

vector<vector<int>> dp(s.size(), vector<int>(s.size()));

for (size\_t i = 0; i < s.size(); i++)

{

dp[i][i] = 1;

for (size\_t j = 0; j < i; j++)

{

if (s[i] != s[j]) continue;

if ((i == j + 1) || (dp[j + 1][i - 1] == 1))

{

dp[j][i] = 1;

}

}

}

vector<vector<string>> result;

vector<string> path;

partition(s, path, 0, dp, 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**.

/// <summary>

/// Leet Code 139. Word Break

/// </summary>

bool LeetCodeDFS::wordBreakI(string s, size\_t pos, vector<int>& visited, size\_t min\_len, size\_t max\_len,

unordered\_set<string>& wordDict)

{

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

if (visited[pos] == 1) return false;

for (size\_t i = min\_len; i <= max\_len; i++)

{

if (pos + i > s.size()) break;

if (wordDict.count(s.substr(pos, i)) == 0) continue;

bool ret = wordBreakI(s, pos + i, visited, min\_len, max\_len, wordDict);

if (ret) return ret;

}

visited[pos] = 1;

return false;

}

/// <summary>

/// Leet code #139. Word Break

///

/// Medium

///

/// Given a non-empty string s and a dictionary wordDict containing a

/// list of non-empty words, determine if s can be segmented into a

/// space-separated sequence of one or more dictionary words.

///

/// Note:

///

/// The same word in the dictionary may be reused multiple times in

/// the segmentation.

/// You may assume the dictionary does not contain duplicate words.

///

/// 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

/// </summary>

bool LeetCodeDFS::wordBreakI(string s, vector<string>& wordDict)

{

unordered\_set<string> word\_set;

size\_t min\_len = INT\_MAX;

size\_t max\_len = 0;

for (string str : wordDict)

{

min\_len = min(min\_len, str.size());

max\_len = max(max\_len, str.size());

word\_set.insert(str);

}

vector<int> visited(s.size());

return wordBreakI(s, 0, visited, min\_len, max\_len, word\_set);

}

## [140. Word Break II](https://leetcode.com/problems/word-break-ii/)

Hard

Given a string s and a dictionary of strings wordDict, add spaces in s to construct a sentence where each word is a valid dictionary word. Return all such possible sentences in **any order**.

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

**Example 1:**

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

**Output:** ["cats and dog","cat sand dog"]

**Example 2:**

**Input:** s = "pineapplepenapple", wordDict = ["apple","pen","applepen","pine","pineapple"]

**Output:** ["pine apple pen apple","pineapple pen apple","pine applepen apple"]

**Explanation:** Note that you are allowed to reuse a dictionary word.

**Example 3:**

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

**Output:** []

**Constraints:**

* 1 <= s.length <= 20
* 1 <= wordDict.length <= 1000
* 1 <= wordDict[i].length <= 10
* s and wordDict[i] consist of only lowercase English letters.
* All the strings of wordDict are **unique**.
* Input is generated in a way that the length of the answer doesn't exceed 105.

/// <summary>

/// Leet code #140. Word Break II

/// </summary>

void LeetCodeDFS::wordBreakII(string& s, vector<string>& path, size\_t start,

vector<vector<bool>> &dp, vector<string>& result)

{

if (start == s.size())

{

string buff;

for (size\_t i = 0; i < path.size(); i++)

{

if (!buff.empty()) buff.push\_back(' ');

buff.append(path[i]);

}

result.push\_back(buff);

}

for (size\_t end = start; end < s.size(); end++)

{

if (dp[start][end] == true)

{

path.push\_back(s.substr(start, end - start + 1));

wordBreakII(s, path, end + 1, dp, result);

path.pop\_back();

}

}

}

/// <summary>

/// Leet Code 140. Word Break II

///

/// Hard

///

/// Given a string s and a dictionary of strings wordDict, add spaces in

/// s to construct a sentence where each word is a valid dictionary word.

/// Return all such possible sentences in any order.

/// Note that the same word in the dictionary may be reused multiple

/// times in the segmentation.

///

/// Example 1:

/// Input: s = "catsanddog", wordDict = ["cat","cats","and","sand","dog"]

/// Output: ["cats and dog","cat sand dog"]

///

/// Example 2:

/// Input: s = "pineapplepenapple", wordDict = ["apple","pen","applepen",

/// "pine","pineapple"]

/// Output: ["pine apple pen apple","pineapple pen apple","pine

/// applepen apple"]

/// Explanation: Note that you are allowed to reuse a dictionary word.

///

/// Example 3:

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

/// Output: []

///

/// Constraints:

/// 1. 1 <= s.length <= 20

/// 2. 1 <= wordDict.length <= 1000

/// 3. 1 <= wordDict[i].length <= 10

/// 4. s and wordDict[i] consist of only lowercase English letters.

/// 5. All the strings of wordDict are unique.

/// 6. Input is generated in a way that the length of the answer doesn't

/// exceed 105.

/// </summary>

vector<string> LeetCodeDFS::wordBreakII(string s, vector<string>& wordDict)

{

vector<vector<bool>> dp(s.size(), vector<bool>(s.size()));

unordered\_set<string> dict;

for (string str : wordDict)

{

dict.insert(str);

}

for (size\_t i = 0; i < s.size(); i++)

{

for (size\_t j = i; j < s.size(); j++)

{

string str = s.substr(i, j - i + 1);

if (dict.count(str) > 0)

{

dp[i][j] = true;

}

}

}

vector<string> result;

vector<string> path;

wordBreakII(s, path, 0, dp, result);

return result;

}

## 51. N-Queens

Hard

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

A screenshot of a game

Description automatically generated

Given an integer *n*, return all distinct solutions to the *n*-queens puzzle.

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:**

**Input:** 4

**Output:** [

[".Q..", // Solution 1

"...Q",

"Q...",

"..Q."],

["..Q.", // Solution 2

"Q...",

"...Q",

".Q.."]

]

**Explanation:** There exist two distinct solutions to the 4-queens puzzle as shown above.

### Analysis:

We can try to put the queen in each row and check the conflict, and keep the column, diagonal and back-diagonal line in an array, if we see the conflict we will skip the choice.

/// <summary>

/// Leet code # 51. N-Queens

/// </summary>

void LeetCodeDFS::solveNQueens(vector<string> &board, int row,

vector<int> &columns, vector<int>&diag, vector<vector<string>> &result)

{

size\_t n = board.size();

if (row == n)

{

result.push\_back(board);

return;

}

for (size\_t col = 0; col < n; col++)

{

if ((columns[col] == 1) || (diag[row + col] == 1) ||

(diag[row - col + 3 \* n] == 1))

{

continue;

}

board[row][col] = 'Q';

columns[col] = 1;

diag[row + col] = 1;

diag[row - col + 3 \* n] = 1;

solveNQueens(board, row + 1, columns, diag, result);

board[row][col] = '.';

columns[col] = 0;

diag[row + col] = 0;

diag[row - col + 3 \* n] = 0;

}

}

/// <summary>

/// Leet code # 51. N-Queens

///

/// The n-queens puzzle is the problem of placing n queens on an n×n

/// chessboard such that no two queens attack each other.

/// Given an integer n, return all distinct solutions to the n-queens puzzle.

///

/// Each solution contains a distinct board configuration of the n-queens'

/// placement, where 'Q' and '.'

/// both indicate a queen and an empty space respectively.

/// For example,

/// There exist two distinct solutions to the 4-queens puzzle:

/// [

/// [".Q..", // Solution 1

/// "...Q",

/// "Q...",

/// "..Q."],

///

/// ["..Q.", // Solution 2

/// "Q...",

/// "...Q",

/// ".Q.."]

/// ]

/// </summary>

vector<vector<string>> LeetCodeDFS::solveNQueens(int n)

{

vector<string> board(n, string(n, '.'));

vector<int> columns(n);

vector<int>diag(4 \* n);

vector<vector<string>> result;

solveNQueens(board, 0, columns, diag, result);

return result;

}

## 329. Longest Increasing Path in a Matrix

Hard

Given an integer matrix, find the length of the longest increasing path.

From each cell, you can either move to four directions: left, right, up or down. You may NOT move diagonally or move outside of the boundary (i.e. wrap-around is not allowed).

**Example 1:**

**Input:** nums =

[

[9,9,4],

[6,6,8],

[2,1,1]

]

**Output:** 4

**Explanation:** The longest increasing path is [1, 2, 6, 9].

**Example 2:**

**Input:** nums =

[

[3,4,5],

[3,2,6],

[2,2,1]

]

**Output:** 4

**Explanation:** The longest increasing path is [3, 4, 5, 6]. Moving diagonally is not allowed.

### Analysis:

We keep track on the longest path from each cell and reuse them as cache to build path.

/// <summary>

/// Leet code #329. Longest Increasing Path in a Matrix

/// </summary>

int LeetCodeDFS::longestIncreasingPath(vector<vector<int>>& matrix,

vector<vector<int>>& path, vector<vector<int>>& directions,

int i, int j)

{

if (path[i][j] > 0) return path[i][j];

path[i][j] = 1;

for (size\_t d = 0; d < directions.size(); d++)

{

int r = i + directions[d][0];

int c = j + directions[d][1];

if (r < 0 || r >= (int)matrix.size() || c < 0 || c >= (int)matrix[0].size())

{

continue;

}

if (matrix[r][c] > matrix[i][j])

{

path[i][j] = max(path[i][j], 1 + longestIncreasingPath(matrix, path, directions, r, c));

}

}

return path[i][j];

}

/// <summary>

/// Leet code #329. Longest Increasing Path in a Matrix

///

/// Hard

///

/// Given an integer matrix, find the length of the longest increasing

/// path.

/// From each cell, you can either move to four directions: left, right,

/// up or down. You may NOT move diagonally or move outside of the

/// boundary (i.e. wrap-around is not allowed).

/// Example 1:

/// Input: nums =

/// [

/// [9,9,4],

/// [6,6,8],

/// [2,1,1]

/// ]

/// Output: 4

/// Explanation: The longest increasing path is [1, 2, 6, 9].

///

/// Example 2:

/// Input: nums =

/// [

/// [3,4,5],

/// [3,2,6],

/// [2,2,1]

/// ]

/// Output: 4

/// Explanation: The longest increasing path is [3, 4, 5, 6].

/// Moving diagonally is not allowed.

/// </summary>

int LeetCodeDFS::longestIncreasingPath(vector<vector<int>>& matrix)

{

int result = 0;

if (matrix.empty()) return result;

vector<vector<int>> path(matrix.size(), vector<int>(matrix[0].size()));

vector<vector<int>> directions = { {-1, 0}, {1, 0}, {0, -1}, {0, 1} };

for (size\_t i = 0; i < matrix.size(); i++)

{

for (size\_t j = 0; j < matrix[i].size(); j++)

{

result = max(result,

longestIncreasingPath(matrix, path, directions, i, j));

}

}

return result;

}