# 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.

## Pruning

If we consider the whole search process as a tree, then the key to improve the performance of backtracking is the pruning. We should cut the invalid choice as soon as possible and avoid going further steps to waste the time.

For example, if we want to get all the combination for some numbers, assume we have number A, B, C in the number set, we should say the order to pick the number does not matter, for example choose A,B,C or B,A,C will lead to the same answer, in this case we should avoid the duplicated search, two common tricks to do the pruning is either we sort the candidates or mark the numbers we already visited. The second trick will lead to a more interesting discussion on that how to mark the candidates and how to check it. The answer is normally a hash table with a bit map as the key.

## 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;

}

## 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.



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;

}

## 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;

}

## 464. Can I Win

Medium

In the "100 game," two players take turns adding, to a running total, any integer from 1..10. The player who first causes the running total to reach or exceed 100 wins.

What if we change the game so that players cannot re-use integers?

For example, two players might take turns drawing from a common pool of numbers of 1..15 without replacement until they reach a total >= 100.

Given an integer maxChoosableInteger and another integer desiredTotal, determine if the first player to move can force a win, assuming both players play optimally.

You can always assume that maxChoosableInteger will not be larger than 20 and desiredTotal will not be larger than 300.

**Example**

**Input:**

maxChoosableInteger = 10

desiredTotal = 11

**Output:**

false

**Explanation:**

No matter which integer the first player choose, the first player will lose.

The first player can choose an integer from 1 up to 10.

If the first player choose 1, the second player can only choose integers from 2 up to 10.

The second player will win by choosing 10 and get a total = 11, which is >= desiredTotal.

Same with other integers chosen by the first player, the second player will always win.

### Analysis:

We can use DFS to search any winnable solution, with any numbers picked up, if A pick any remaining number either he win immediately with total sum greater than desired total or the opponent B is not able to win with the remaining number then A win in this round. Here we use cache to track a known set of numbers are picked up, we do not care who pick them and in which order they are picked, we only care with the remaining number, can the first player win or not.

We use bit map to track the numbers are picked.

/// <summary>

/// Leet code #464. Can I Win

/// </summary>

bool LeetCodeDFS::canIWin(int maxChoosableInteger, int desiredTotal,

int signature, unordered\_map<int, bool> &game\_map)

{

if (game\_map.count(signature) > 0)

{

return game\_map[signature];

}

for (int i = maxChoosableInteger; i > 0; i--)

{

int bit = (1 << (i - 1));

if ((signature & bit) == 0)

{

desiredTotal = desiredTotal - i;

if ((desiredTotal <= 0) ||

!canIWin(maxChoosableInteger, desiredTotal, signature | bit, game\_map))

{

game\_map[signature] = true;

return true;

}

desiredTotal = desiredTotal + i;

}

}

game\_map[signature] = false;

return false;

}

/// <summary>

/// Leet code #464. Can I Win

///

/// In the "100 game," two players take turns adding, to a running total,

/// any integer from 1..10.

/// The player who first causes the running total to reach or exceed 100 wins.

/// What if we change the game so that players cannot re-use integers?

/// For example, two players might take turns drawing from a common pool of

/// numbers of 1..15 without replacement until they reach a total >= 100.

/// Given an integer maxChoosableInteger and another integer desiredTotal,

/// determine if the first player to move can force a win, assuming both

/// players play optimally.

/// You can always assume that maxChoosableInteger will not be larger than 20

/// and desiredTotal will not be larger than 300.

///

/// Example

/// Input:

/// maxChoosableInteger = 10

/// desiredTotal = 11

/// Output:

/// false

/// Explanation:

/// No matter which integer the first player choose, the first player will lose.

/// The first player can choose an integer from 1 up to 10.

/// If the first player choose 1, the second player can only choose integers

/// from 2 up to 10.

/// The second player will win by choosing 10 and get a total = 11, which

/// is >= desiredTotal.

/// Same with other integers chosen by the first player, the second player will

/// always win.

/// </summary>

bool LeetCodeDFS::canIWin(int maxChoosableInteger, int desiredTotal)

{

unordered\_map<int, bool> game\_map;

int signature = 0;

// No one can win

if ((1 + maxChoosableInteger) \* maxChoosableInteger < desiredTotal \* 2)

{

return false;

}

return canIWin(maxChoosableInteger, desiredTotal, signature, game\_map);

}

## 526. Beautiful Arrangement

Medium

Suppose you have **N** integers from 1 to N. We define a beautiful arrangement as an array that is constructed by these **N** numbers successfully if one of the following is true for the ith position (1 <= i <= N) in this array:

1. The number at the ith position is divisible by **i**.
2. **i** is divisible by the number at the ith position.

Now given N, how many beautiful arrangements can you construct?

**Example 1:**

**Input:** 2

**Output:** 2

**Explanation:**

The first beautiful arrangement is [1, 2]:

Number at the 1st position (i=1) is 1, and 1 is divisible by i (i=1).

Number at the 2nd position (i=2) is 2, and 2 is divisible by i (i=2).

The second beautiful arrangement is [2, 1]:

Number at the 1st position (i=1) is 2, and 2 is divisible by i (i=1).

Number at the 2nd position (i=2) is 1, and i (i=2) is divisible by 1.

**Note:**

1. **N** is a positive integer and will not exceed 15.

### Analysis:

We try to place the number in order from 1 to N, and keep the track on the numbers already placed by bit map.

/// <summary>

/// Leet code #526. Beautiful Arrangement

/// </summary>

int LeetCode::countArrangement(int N, int index, int visited, unordered\_map<int, int>& cache)

{

int result = 0;

if (index == N)

{

return 1;

}

if (cache.count(visited) > 0)

{

return cache[visited];

}

for (int i = 1; i <= N; i++)

{

int bit = (1 << (i - 1));

if (((index + 1) % i != 0) && (i % (index + 1) != 0)) continue;

if ((visited & bit) == 0)

{

result += countArrangement(N, index + 1, visited | bit, cache);

}

}

cache[visited] = result;

return result;

}

/// <summary>

/// Leet code #526. Beautiful Arrangement

///

/// Suppose you have N integers from 1 to N. We define a beautiful

/// arrangement as an array that is constructed by these N numbers

/// successfully if one of the following is true for the ith position

/// (1 ≤ i ≤ N) in this array:

/// 1.The number at the ith position is divisible by i.

/// 2.i is divisible by the number at the ith position.

///

/// Now given N, how many beautiful arrangements can you construct?

///

/// Example 1:

///

/// Input: 2

/// Output: 2

///

/// Explanation:

/// The first beautiful arrangement is [1, 2]:

/// Number at the 1st position (i=1) is 1, and 1 is divisible by i (i=1).

/// Number at the 2nd position (i=2) is 2, and 2 is divisible by i (i=2).

///

/// The second beautiful arrangement is [2, 1]:

/// Number at the 1st position (i=1) is 2, and 2 is divisible by i (i=1).

/// Number at the 2nd position (i=2) is 1, and i (i=2) is divisible by 1.

/// Note:

/// 1.N is a positive integer and will not exceed 15.

/// </summary>

int LeetCode::countArrangement(int N)

{

int visited = 0 ;

unordered\_map<int, int> cache;

return countArrangement(N, 0, visited, cache);

}