# LeetCode Day 22 Backtracking Memorization

One of the Backtracking is pruning. This is to say we can cut the search process in the early stage if we know the result already.

## 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.

One common technique we use is to memorize the previous search result. This is called memorization.

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. One common way is a hash table with a bit map as the key.

## 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);

}

## 473. Matchsticks to Square

Medium

68862Add to ListShare

Remember the story of Little Match Girl? By now, you know exactly what matchsticks the little match girl has, please find out a way you can make one square by using up all those matchsticks. You should not break any stick, but you can link them up, and each matchstick must be used **exactly** one time.

Your input will be several matchsticks the girl has, represented with their stick length. Your output will either be true or false, to represent whether you could make one square using all the matchsticks the little match girl has.

**Example 1:**

**Input:** [1,1,2,2,2]

**Output:** true

**Explanation:** You can form a square with length 2, one side of the square came two sticks with length 1.

**Example 2:**

**Input:** [3,3,3,3,4]

**Output:** false

**Explanation:** You cannot find a way to form a square with all the matchsticks.

**Note:**

1. The length sum of the given matchsticks is in the range of 0 to 10^9.
2. The length of the given matchstick array will not exceed 15.

### Analysis:

First keep stick increasing, then use bitmap to remember which match sticks are selected and verified as false.

/// <summary>

/// Leet code #473. Matchsticks to Square

/// </summary>

bool LeetCodeDFS::makesquare(vector<int>& nums, vector<int>& path, int bit\_mask, int sum,

int side\_length, unordered\_set<int>& cache)

{

if (path.size() == nums.size()) return true;

if (cache.count(bit\_mask) > 0) return false;

int start = 0;

sum %= side\_length;

if (sum > 0)

{

start = path.back() + 1;

}

int n = nums.size();

bool result = false;

for (int i = start; i < n; i++)

{

if (sum + nums[i] > side\_length) break;

int bit = 1 << i;

if ((bit\_mask & bit) != 0) continue;

path.push\_back(i);

result = makesquare(nums, path, bit\_mask | bit, sum + nums[i], side\_length, cache);

path.pop\_back();

if (result == true) break;

}

if (result == false) cache.insert(bit\_mask);

return result;

}

/// <summary>

/// Leet code #473. Matchsticks to Square

///

/// Medium

///

/// Remember the story of Little Match Girl? By now, you know exactly what

/// matchsticks the little match girl has, please find out a way you can

/// make one square by using up all those matchsticks. You should not

/// break any stick, but you can link them up, and each matchstick must be

/// used exactly one time.

///

/// Your input will be several matchsticks the girl has, represented with

/// their stick length. Your output will either be true or false, to

/// represent whether you could make one square using all the matchsticks

/// the little match girl has.

///

/// Example 1:

/// Input: [1,1,2,2,2]

/// Output: true

///

/// Explanation: You can form a square with length 2, one side of the

/// square came two sticks with length 1.

///

/// Example 2:

/// Input: [3,3,3,3,4]

/// Output: false

/// Explanation: You cannot find a way to form a square with all the

/// matchsticks.

///

/// Note:

/// The length sum of the given matchsticks is in the range of 0 to 10^9.

/// The length of the given matchstick array will not exceed 15.

/// </summary>

bool LeetCodeDFS::makesquare(vector<int>& nums)

{

unordered\_set<int> cache;

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

vector<int> path;

int sum = 0;

for (size\_t i = 0; i < nums.size(); i++) sum += nums[i];

if ((sum == 0) || (sum % 4 != 0)) return false;

int side\_length = sum / 4;

return makesquare(nums, path, 0, 0, side\_length, cache);

}

## [1066. Campus Bikes II](https://leetcode.com/problems/campus-bikes-ii/)

Medium

On a campus represented as a 2D grid, there are n workers and m bikes, with n <= m. Each worker and bike is a 2D coordinate on this grid.

We assign one unique bike to each worker so that the sum of the **Manhattan distances** between each worker and their assigned bike is minimized.

Return the minimum possible sum of Manhattan distances between each worker and their assigned bike.

The **Manhattan distance** between two points p1 and p2 is Manhattan(p1, p2) = |p1.x - p2.x| + |p1.y - p2.y|.

**Example 1:**

A graph of a bicycle and a person

Description automatically generated

**Input:** workers = [[0,0],[2,1]], bikes = [[1,2],[3,3]]

**Output:** 6

**Explanation:**

We assign bike 0 to worker 0, bike 1 to worker 1. The Manhattan distance of both assignments is 3, so the output is 6.

**Example 2:**

A diagram of a person and a bicycle

Description automatically generated

**Input:** workers = [[0,0],[1,1],[2,0]], bikes = [[1,0],[2,2],[2,1]]

**Output:** 4

**Explanation:**

We first assign bike 0 to worker 0, then assign bike 1 to worker 1 or worker 2, bike 2 to worker 2 or worker 1. Both assignments lead to sum of the Manhattan distances as 4.

**Example 3:**

**Input:** workers = [[0,0],[1,0],[2,0],[3,0],[4,0]], bikes = [[0,999],[1,999],[2,999],[3,999],[4,999]]

**Output:** 4995

**Constraints:**

* n == workers.length
* m == bikes.length
* 1 <= n <= m <= 10
* workers[i].length == 2
* bikes[i].length == 2
* 0 <= workers[i][0], workers[i][1], bikes[i][0], bikes[i][1] < 1000
* All the workers and the bikes locations are **unique**.

/// <summary>

/// Leet Code 1066. Campus Bikes II

/// </summary>

int LeetCodeDFS::assignBikesII(vector<vector<int>>& workers, vector<vector<int>>& bikes,

int worker, int bit\_mask, vector<int>& dp)

{

if (worker == workers.size())

{

return 0;

}

if (dp[bit\_mask] != INT\_MAX) return dp[bit\_mask];

int result = INT\_MAX;

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

{

if ((bit\_mask & (1 << i)) != 0) continue;

int distance = std::abs(workers[worker][0] - bikes[i][0]) + std::abs(workers[worker][1] - bikes[i][1]);

result = min(result, distance + assignBikesII(workers, bikes, worker + 1, bit\_mask | (1 << i), dp));

}

dp[bit\_mask] = result;

return result;

}

/// <summary>

/// Leet code #1066. Campus Bikes II

///

/// On a campus represented as a 2D grid, there are N workers and M bikes,

/// with N <= M. Each worker and bike is a 2D coordinate on this grid.

///

/// We assign one unique bike to each worker so that the sum of the

/// Manhattan distances between each worker and their assigned bike is

/// minimized.

///

/// The Manhattan distance between two points p1 and p2 is

/// Manhattan(p1, p2) = |p1.x - p2.x| + |p1.y - p2.y|.

///

/// Return the minimum possible sum of Manhattan distances between each

/// worker and their assigned bike.

///

/// Example 1:

/// Input: workers = [[0,0],[2,1]], bikes = [[1,2],[3,3]]

/// Output: 6

/// Explanation:

/// We assign bike 0 to worker 0, bike 1 to worker 1. The Manhattan

/// distance of both assignments is 3, so the output is 6.

///

/// Example 2:

/// Input: workers = [[0,0],[1,1],[2,0]], bikes = [[1,0],[2,2],[2,1]]

/// Output: 4

/// Explanation:

/// We first assign bike 0 to worker 0, then assign bike 1 to worker 1

/// or worker 2, bike 2 to worker 2 or worker 1. Both assignments lead

/// to sum of the Manhattan distances as 4.

///

/// Note:

/// 1. 0 <= workers[i][0], workers[i][1], bikes[i][0], bikes[i][1] < 1000

/// 2. All worker and bike locations are distinct.

/// 3. 1 <= workers.length <= bikes.length <= 10

/// </summary>

int LeetCodeDFS::assignBikesII(vector<vector<int>>& workers, vector<vector<int>>& bikes)

{

vector<int> dp(1 << bikes.size(), INT\_MAX);

int result = assignBikesII(workers, bikes, 0, 0, dp);

return result;

}

## [2992. Number of Self-Divisible Permutations](https://leetcode.com/problems/number-of-self-divisible-permutations/)

Medium

Given an integer n, return the number of ***permutations*** of the ***1-indexed*** array nums = [1, 2, ..., n], such that it's ***self-divisible***.

A **1-indexed** array a of length n is **self-divisible** if for every 1 <= i <= n,

gcd

(a[i], i) == 1.

A **permutation** of an array is a rearrangement of the elements of that array, for example here are all of the permutations of the array [1, 2, 3]:

* [1, 2, 3]
* [1, 3, 2]
* [2, 1, 3]
* [2, 3, 1]
* [3, 1, 2]
* [3, 2, 1]

**Example 1:**

**Input:** n = 1

**Output:** 1

**Explanation:** The array [1] has only 1 permutation which is self-divisible.

**Example 2:**

**Input:** n = 2

**Output:** 1

**Explanation:** The array [1,2] has 2 permutations and only one of them is self-divisible:

nums = [1,2]: This is not self-divisible since gcd(nums[2], 2) != 1.

nums = [2,1]: This is self-divisible since gcd(nums[1], 1) == 1 and gcd(nums[2], 2) == 1.

**Example 3:**

**Input:** n = 3

**Output:** 3

**Explanation:** The array [1,2,3] has 3 self-divisble permutations: [1,3,2], [3,1,2], [2,3,1].

It can be shown that the other 3 permutations are not self-divisible. Hence the answer is 3.

**Constraints:**

* 1 <= n <= 12

/// <summary>

/// Leet Code 2992. Number of Self-Divisible Permutations

/// </summary>

int LeetCodeDFS::selfDivisiblePermutationCount(int i, int n, int bit\_mask, unordered\_map<int, int>& cache)

{

if (i > n) return 1;

int count = 0;

if (cache.count(bit\_mask) > 0) return cache[bit\_mask];

for (int k = 1; k <= n; k++)

{

// not good

if (std::gcd(i, k) != 1) continue;

// already occupied

if ((bit\_mask & (2 << (k - 1))) != 0)

{

continue;

}

count += selfDivisiblePermutationCount(i + 1, n, bit\_mask | (2 << (k - 1)), cache);

}

cache[bit\_mask] = count;

return count;

}

/// <summary>

/// Leet Code 2992. Number of Self-Divisible Permutations

///

/// Medium

///

/// Given an integer n, return the number of permutations of the 1-indexed

/// array nums = [1, 2, ..., n], such that it's self-divisible.

///

/// Array nums is self-divisible if for every 1 <= i <= n, at least one of

/// the following conditions holds:

///

/// nums[i] % i == 0

/// i % nums[i] == 0

/// A permutation of an array is a rearrangement of the elements of that

/// array, for example here are all of the permutations of the array

/// [1, 2, 3]:

/// [1, 2, 3]

/// [1, 3, 2]

/// [2, 1, 3]

/// [2, 3, 1]

/// [3, 1, 2]

/// [3, 2, 1]

///

/// Example 1:

/// Input: n = 1

/// Output: 1

/// Explanation: The array [1] has only 1 permutation which is

/// self-divisible.

///

/// Example 2:

/// Input: n = 2

/// Output: 2

/// Explanation: The array [1,2] has 2 permutations both of which are

/// self-divisible:

/// nums = [1,2]: This is self-divisible since nums[1] % 1 == 0 and

/// nums[2] % 2 == 0.

/// nums = [2,1]: This is self-divisible since nums[1] % 1 == 0 and

/// 2 % nums[2] == 0.

///

/// Example 3:

/// Input: n = 3

/// Output: 3

/// Explanation: The array [1,2,3] has 3 self-divisble permutations:

/// [1,2,3], [2,1,3], [3,2,1].

/// It can be shown that the other 3 permutations are not self-divisible.

/// Hence the answer is 3.

///

/// Constraints:

/// 1. 1 <= n <= 15

/// </summary>

int LeetCodeDFS::selfDivisiblePermutationCount(int n)

{

unordered\_map<int, int> cache;

return selfDivisiblePermutationCount(1, n, 0, cache);

}

## [2741. Special Permutations](https://leetcode.com/problems/special-permutations/)

Medium

You are given a **0-indexed** integer array nums containing n **distinct** positive integers. A permutation of nums is called special if:

* For all indexes 0 <= i < n - 1, either nums[i] % nums[i+1] == 0 or nums[i+1] % nums[i] == 0.

Return the total number of special permutations. As the answer could be large, return it **modulo**109+ 7.

**Example 1:**

**Input:** nums = [2,3,6]

**Output:** 2

**Explanation:** [3,6,2] and [2,6,3] are the two special permutations of nums.

**Example 2:**

**Input:** nums = [1,4,3]

**Output:** 2

**Explanation:** [3,1,4] and [4,1,3] are the two special permutations of nums.

**Constraints:**

* 2 <= nums.length <= 14
* 1 <= nums[i] <= 109

/// <summary>

/// Leet Code 2741. Special Permutations

/// </summary>

int LeetCodeDFS::specialPerm(int k, int bits, vector<vector<int>>& neighbors, vector<vector<int>>& dp)

{

int M = 1000000007;

int n = neighbors.size();

bits |= (1 << k);

// if all nodes are visited

if (bits == (1 << n) - 1) return 1;

int result = 0;

for (size\_t i = 0; i < neighbors[k].size(); i++)

{

int next = neighbors[k][i];

// skip if the next node is already visited

if ((bits & (1 << next)) != 0) continue;

// if the next node is already calculated

if (dp[bits][next] != -1)

{

result += dp[bits][next];

result %= M;

continue;

}

result += specialPerm(next, bits, neighbors, dp);

result %= M;

}

bits ^= (1 << k);

dp[bits][k] = result;

return result;

}

/// <summary>

/// Leet Code 2741. Special Permutations

///

/// Medium

///

/// You are given a 0-indexed integer array nums containing n distinct

/// positive integers. A permutation of nums is called special if:

///

/// For all indexes 0 <= i < n - 1, either nums[i] % nums[i+1] == 0 or

/// nums[i+1] % nums[i] == 0.

/// Return the total number of special permutations. As the answer could

/// be large, return it modulo 10^9 + 7.

///

/// Example 1:

/// Input: nums = [2,3,6]

/// Output: 2

/// Explanation: [3,6,2] and [2,6,3] are the two special permutations of

/// nums.

///

/// Example 2:

/// Input: nums = [1,4,3]

/// Output: 2

/// Explanation: [3,1,4] and [4,1,3] are the two special permutations of

/// nums.

///

/// Constraints:

/// 1. 2 <= nums.length <= 14

/// 2. 1 <= nums[i] <= 10^9

/// </summary>

int LeetCodeDFS::specialPerm(vector<int>& nums)

{

int M = 1000000007;

int n = nums.size();

vector<vector<int>> dp(1 << n,vector<int>(n, - 1));

vector<vector<int>> neighbors(n);

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

if (i == j) continue;

if (nums[i] % nums[j] == 0 || nums[j] % nums[i] == 0)

{

neighbors[i].push\_back(j);

}

}

}

int result = 0;

for (int k = 0; k < n; k++)

{

result += specialPerm(k, 0, neighbors, dp);

result %= M;

}

return result;

}