# Pruning with 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);

}

## 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;

}