# Divide and Conquer

The third common pattern in Backtracking is divide and conquer. This is a top down resolution, first divide the big problem into small parts, then divide the smaller part further to smaller pieces, until the basic unit. We can use memorization for a range which is already searched.

The counter part of this solution is the Dynamic Programming which build from bottom to top. So for such problem normally either way works, depends on which one is more straightforward to you.

## 241. Different Ways to Add Parentheses

Medium

Given a string of numbers and operators, return all possible results from computing all the different possible ways to group numbers and operators. The valid operators are +, - and \*.

**Example 1:**

**Input:** "2-1-1"

**Output:** [0, 2]

**Explanation:**

((2-1)-1) = 0

(2-(1-1)) = 2

**Example 2:**

**Input:** "2\*3-4\*5"

**Output:** [-34, -14, -10, -10, 10]

**Explanation:**

(2\*(3-(4\*5))) = -34

((2\*3)-(4\*5)) = -14

((2\*(3-4))\*5) = -10

(2\*((3-4)\*5)) = -10

(((2\*3)-4)\*5) = 10

### Analysis:

When we see an operator, we can simply divide the expression as two parts, left and right, and the recursive call the function to get all the possible result for left and right.

/// <summary>

/// Leet code #241. Different Ways to Add Parentheses

///

/// Given a string of numbers and operators, return all possible

/// results from computing all the different possible ways to group

/// numbers and operators. The valid operators are +, - and \*.

///

/// Example 1

/// Input: "2-1-1".

/// ((2-1)-1) = 0

/// (2-(1-1)) = 2

/// Output: [0, 2]

///

/// Example 2

/// Input: "2\*3-4\*5"

/// (2\*(3-(4\*5))) = -34

/// ((2\*3)-(4\*5)) = -14

/// ((2\*(3-4))\*5) = -10

/// (2\*((3-4)\*5)) = -10

/// (((2\*3)-4)\*5) = 10

/// Output: [-34, -14, -10, -10, 10]

/// </summary>

vector<int> LeetCodeDFS::diffWaysToCompute(string input)

{

vector<int> result;

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

{

if ((input[i] == '+') || (input[i] == '-') || (input[i] == '\*'))

{

vector<int> left = diffWaysToCompute(input.substr(0, i));

vector<int> right = diffWaysToCompute(input.substr(i + 1));

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

{

for (size\_t k = 0; k < right.size(); k++)

{

if (input[i] == '+') result.push\_back(left[j] + right[k]);

else if (input[i] == '-') result.push\_back(left[j] - right[k]);

else if (input[i] == '\*') result.push\_back(left[j] \* right[k]);

}

}

}

}

if (result.empty())

{

result.push\_back(atoi(input.c\_str()));

}

return result;

}

**1140. Stone Game II**

Medium

Alice and Bob continue their games with piles of stones.  There are a number of piles **arranged in a row**, and each pile has a positive integer number of stones piles[i].  The objective of the game is to end with the most stones.

Alice and Bob take turns, with Alice starting first.  Initially, M = 1.

On each player's turn, that player can take **all the stones** in the **first** X remaining piles, where 1 <= X <= 2M.  Then, we set M = max(M, X).

The game continues until all the stones have been taken.

Assuming Alice and Bob play optimally, return the maximum number of stones Alice can get.

**Example 1:**

**Input:** piles = [2,7,9,4,4]

**Output:** 10

**Explanation:** If Alice takes one pile at the beginning, Bob takes two piles, then Alice takes 2 piles again. Alice can get 2 + 4 + 4 = 10 piles in total. If Alice takes two piles at the beginning, then Bob can take all three piles left. In this case, Alice get 2 + 7 = 9 piles in total. So we return 10 since it's larger.

**Example 2:**

**Input:** piles = [1,2,3,4,5,100]

**Output:** 104

**Constraints:**

* 1 <= piles.length <= 100
* 1 <= piles[i] <= 104

### Analysis:

For every selection on current step, it will determine the range in next step. The top down solution is more clean than bottom up.

/// <summary>

/// Leet code #1140. Stone Game II

/// </summary>

int LeetCodeDFS::stoneGameII(vector<int>& sum, int start, int M, vector<vector<int>>& memo)

{

int result = 0;

if (start + 2 \* M >= (int)sum.size() - 1)

{

result = sum[sum.size() - 1] - sum[start];

}

else if (memo[start][M] > 0)

{

result = memo[start][M];

}

else

{

for (int i = start; i < start + 2 \* M; i++)

{

result = max(result, sum[i+1] - sum[start] + sum[sum.size() - 1] - sum[i+1] -

stoneGameII(sum, i + 1, max(i - start + 1, M), memo));

}

}

memo[start][M] = result;

return result;

}

/// <summary>

/// Leet code #1140. Stone Game II

///

/// Alex and Lee continue their games with piles of stones. There are a

/// number of piles arranged in a row, and each pile has a positive

/// integer number of stones piles[i]. The objective of the game is to

/// end with the most stones.

///

/// Alex and Lee take turns, with Alex starting first. Initially, M = 1.

///

/// On each player's turn, that player can take all the stones in the

/// first X remaining piles, where 1 <= X <= 2M. Then, we set M = max(M, X).

///

/// The game continues until all the stones have been taken.

///

/// Assuming Alex and Lee play optimally, return the maximum number of

/// stones Alex can get.

///

/// Example 1:

/// Input: piles = [2,7,9,4,4]

/// Output: 10

/// Explanation: If Alex takes one pile at the beginning, Lee takes two

/// piles, then Alex takes 2 piles again. Alex can get 2 + 4 + 4 = 10 piles

/// in total. If Alex takes two piles at the beginning, then Lee can take

/// all three piles left. In this case, Alex get 2 + 7 = 9 piles in total.

/// So we return 10 since it's larger.

///

/// Constraints:

/// 1. 1 <= piles.length <= 100

/// 2. 1 <= piles[i] <= 10 ^ 4

/// </summary>

int LeetCodeDFS::stoneGameII(vector<int>& piles)

{

vector<int> sum;

sum.push\_back(0);

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

{

sum.push\_back(sum.back() + piles[i]);

}

vector<vector<int>> memo(piles.size() + 1, vector<int>(piles.size()));

return stoneGameII(sum, 0, 1, memo);

}