# Two pointers

The two pointers problem is that you may use two pointers, either they are two indexes, or two iterators for the container, they can iterate either from front to end, the last pointer is always ahead of the first pointer or they traverse from both end of the array until they meet.

### **42. Trapping Rain Water**

Given *n* non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

For example,   
Given [0,1,0,2,1,0,1,3,2,1,2,1], return 6.



The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped. **Thanks Marcos** for contributing this image!

Hint: the idea is to have two pointers pointing both ends and move the shorter side to taller side, because the water trapped depends on the short side, if we saw a new bar is lower than the previous bar, so we know there is water trapped, we collect water and fill the lower bar to the same height as the previous bar and continue the move, until the two pointers meet.

/// <summary>

/// Leet code #42. Trapping Rain Water

/// Given n non-negative integers representing an elevation map where the

/// width of each bar is 1,

/// compute how much water it is able to trap after raining.

/// For example,

/// Given [0,1,0,2,1,0,1,3,2,1,2,1], return 6.

/// The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1].

/// In this case, 6 units of rain water (blue section) are being trapped.

/// </summary>

int LeetCode::trapWater(vector<int>& height)

{

int sum = 0;

int left\_index = 0;

int right\_index = height.size() - 1;

int left\_value = height[left\_index];

int right\_value = height[right\_index];

while (left\_index < right\_index)

{

if (left\_value < right\_value)

{

left\_index++;

// if left side become lower, take the water and fill it

if (height[left\_index] < left\_value)

{

sum = sum + left\_value - height[left\_index];

}

else

{

left\_value = height[left\_index];

}

}

else

{

right\_index--;

// if right side become lower, take the water and fill it

if (height[right\_index] < right\_value)

{

sum = sum + right\_value - height[right\_index];

}

else

{

right\_value = height[right\_index];

}

}

}

return sum;

}

Exercise

1. Please complete the following problems, no pay user please skip locked problem

beginner: 26, 27, 125, 209,658, 670

advanced: 11, 42, 680, 713

Locked and Optional: 159, 487

# Array

When you handle the array problem, you may need to know some common techniques

1. How to calculate the partial sum in a 1-D array or a 2-D array without duplicated effort.
2. How to calculate rotated index, starting from 0 to N then to 0.
3. How to move from a cell in a 2D array (4 direction).
4. How to traverse a 2D array from outer edge to the center.
5. How to find out the missing item to the index.

### **53. Maximum Subarray**

Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

For example, given the array [-2,1,-3,4,-1,2,1,-5,4],  
the contiguous subarray [4,-1,2,1] has the largest sum = 6.

/// <summary>

/// Leet code #53. Maximum Subarray

/// Find the contiguous subarray within an array (containing at least

/// one number) which has the largest sum.

/// For example, given the array [-2,1,-3,4,-1,2,1,-5,4],

/// the contiguous subarray [4,-1,2,1] has the largest sum = 6.

/// </summary>

int LeetCode::maxSubArray(vector<int>& nums)

{

int min\_sum = 0;

int max\_sum = INT\_MIN;

int sum = 0;

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

{

sum += nums[i];

// calculate max\_sum first because min\_sum is for previous ones.

max\_sum = max(max\_sum, sum - min\_sum);

min\_sum = min(min\_sum, sum);

}

return max\_sum;

}

### **169. Majority Element**

Given an array of size *n*, find the majority element. The majority element is the element that appears **more than** ⌊ n/2 ⌋ times.

You may assume that the array is non-empty and the majority element always exist in the array.

/// <summary>

/// Leet code #169. Majority Element

/// Given an array of size n, find the majority element. The majority element

/// is the element that appears more than ⌊ n/2 ⌋ times.

/// You may assume that the array is non-empty and the majority element always

/// exist in the array.

/// </summary>

int LeetCode::majorityElement(vector<int>& nums)

{

int count = 0;

int major\_number = 0;

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

{

if (count == 0)

{

major\_number = nums[i];

count++;

}

else

{

if (major\_number == nums[i])

{

count++;

}

else

{

count--;

}

}

}

return major\_number;

}

### **41. First Missing Positive**

Given an unsorted integer array, find the first missing positive integer.

For example,  
Given [1,2,0] return 3,  
and [3,4,-1,1] return 2.

Your algorithm should run in *O*(*n*) time and uses constant space.

/// <summary>

/// Leet code #41. First Missing Positive

/// Given an unsorted integer array, find the first missing positive integer.

/// For example,

/// Given [1,2,0] return 3,

/// and [3,4,-1,1] return 2.

/// Your algorithm should run in O(n) time and uses constant space.

/// </summary>

int LeetCode::firstMissingPositive(vector<int>& nums)

{

if (nums.size() == 0)

{

return 1;

}

size\_t index = 0;

while (index < nums.size())

{

// non-positive or out of range, skip it.

if ((nums[index] <= 0) || (nums[index] >= (int)nums.size()))

{

index++;

}

// already in order, skip it

else if (nums[index] == index + 1)

{

index++;

}

// already same data so no need to swap

else if (nums[index] == nums[nums[index] - 1])

{

index++;

}

else

{

swap(nums[index], nums[nums[index] - 1]);

}

}

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

{

if (nums[i] != i + 1)

{

return i + 1;

}

}

return nums.size() + 1;

}

### **54. Spiral Matrix**

Given a matrix of *m* x *n* elements (*m* rows, *n* columns), return all elements of the matrix in spiral order.

For example,  
Given the following matrix:

[

[ 1, 2, 3 ],

[ 4, 5, 6 ],

[ 7, 8, 9 ]

]

You should return [1,2,3,6,9,8,7,4,5].

Hint: remember the left top, right and bottom position as the wall and move the cursor to traverse the array, turn when hit the wall.

/// <summary>

/// Leet code #54. Spiral Matrix

/// Given a matrix of m x n elements (m rows, n columns), return all elements

/// of the matrix in spiral order.

/// For example,

/// Given the following matrix:

/// [

/// [ 1, 2, 3 ],

/// [ 4, 5, 6 ],

/// [ 7, 8, 9 ]

/// ]

/// You should return [1,2,3,6,9,8,7,4,5].

/// </summary>

vector<int> LeetCode::spiralOrder(vector<vector<int>>& matrix)

{

vector<int> result;

if (matrix.empty() || matrix[0].empty()) return result;

int begin\_row = 0;

int end\_row = matrix.size() - 1;

int begin\_col = 0;

int end\_col = matrix[0].size() - 1;

int direction = 0;

while ((begin\_row <= end\_row) && (begin\_col <= end\_col))

{

switch (direction)

{

case 0:

for (int i = begin\_col; i <= end\_col; i++)

{

result.push\_back(matrix[begin\_row][i]);

}

begin\_row++;

break;

case 1:

for (int i = begin\_row; i <= end\_row; i++)

{

result.push\_back(matrix[i][end\_col]);

}

end\_col--;

break;

case 2:

for (int i = end\_col; i >= begin\_col; i--)

{

result.push\_back(matrix[end\_row][i]);

}

end\_row--;

break;

case 3:

for (int i = end\_row; i >= begin\_row; i--)

{

result.push\_back(matrix[i][begin\_col]);

}

begin\_col++;

break;

}

direction = (direction + 1) % 4;

}

return result;

}

### **134. Gas Station**

There are *N* gas stations along a circular route, where the amount of gas at station *i* is gas[i].

You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from station *i* to its next station (*i*+1). You begin the journey with an empty tank at one of the gas stations.

Return the starting gas station's index if you can travel around the circuit once, otherwise return -1.

**Note:**  
The solution is guaranteed to be unique.

hint: start from the worst case scenario point and do the travel.

/// <summary>

/// Leet code #134. Gas Station

///

/// There are N gas stations along a circular route, where the amount of gas

/// at station i is gas[i].

/// You have a car with an unlimited gas tank and it costs cost[i] of gas to

/// travel from station i

/// to its next station (i+1). You begin the journey with an empty tank at

/// one of the gas stations.

///

/// Return the starting gas station's index if you can travel around the

/// circuit once, otherwise return -1.

/// Note:

/// The solution is guaranteed to be unique.

/// </summary>

int LeetCode::canCompleteCircuit(vector<int>& gas, vector<int>& cost)

{

vector<int> sum(gas.size());

int start\_index = -1;

int min\_sum = INT\_MAX;

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

{

if (i == 0)

{

sum[i] = gas[i] - cost[i];

}

else

{

sum[i] = sum[i - 1] + gas[i] - cost[i];

}

if (sum[i] < min\_sum)

{

min\_sum = sum[i];

start\_index = (i + 1 == gas.size()) ? 0 : i + 1;

}

}

if (sum[gas.size() - 1] >= 0)

{

return start\_index;

}

else

{

return -1;

}

}

**Advanced User only**

### **287. Find the Duplicate Number**

Given an array *nums* containing *n* + 1 integers where each integer is between 1 and *n* (inclusive), prove that at least one duplicate number must exist. Assume that there is only one duplicate number, find the duplicate one.

**Note:**

1. You **must not** modify the array (assume the array is read only).
2. You must use only constant, *O*(1) extra space.
3. Your runtime complexity should be less than O(n2).
4. There is only one duplicate number in the array, but it could be repeated more than once.

Hint: It is based on tortoise and hair method, It is exactly same as how you

find the entry point of a cycled linked list. consider the array as a

loop linked list each cell point to the index of another cell (which is

1-n), so if you have duplicate (two cells point to same index),

it must be a loop. You start from 0 where you will not be there again

because no value in array is 0, then you enter the loop and leet

fast (2X) and slow (1X) pointers meet, the point enter the loop is the

duplicate number, assume from 0 to duplicate point is X, loop cycle is

C and meet point is Y, you have 2\*(X+Y) = X+Y+C X = C-Y, so after the slow

and fast pointer meet, you place the fast point back to start (0), and let

it move in normal speed, when the slow complete the travel on the

remaining cycle which is C-Y, the fast travel X and they meet again, now it

is the duplicate number also enter point to cycle

/// <summary>

/// Leet code #287. Find the Duplicate Number

/// Given an array nums containing n + 1 integers where each integer is

/// between 1 and n (inclusive),

/// prove that at least one duplicate number must exist. Assume that there

/// is only one duplicate number, find the duplicate one.

/// Note:

/// 1.You must not modify the array (assume the array is read only).

/// 2.You must use only constant, O(1) extra space.

/// 3.Your runtime complexity should be less than O(n^2).

/// 4.There is only one duplicate number in the array, but it could be

/// repeated more than once.

/// </summary>

int LeetCode::findDuplicate(vector<int>& nums)

{

// protect empty array

if (nums.size() == 0) return -1;

// tortoise

int slow = 0;

int fast = 0;

while (true)

{

slow = nums[slow];

fast = nums[nums[fast]];

if (slow == fast) break;

}

fast = 0;

while (true)

{

slow = nums[slow];

fast = nums[fast];

if (slow == fast) break;

}

return slow;

}

Exercise

1. Please complete the following problems, no pay user please skip locked problem

beginner: 53,54, 66,67,88, 134, 168, 169, 171, 189,268,283, 349, 350, 448, 455,539, 544, 566,575,581,594,598,599,643,661,665,686,696,724,747

advanced: 41, 48, 59, 73, 152,229,238, 284, 303, 304, 396, 413, 419, 454, 498, 525,565,649,659, 667,678,750. (red is special and optional)

Locked and Optional: 6, 36,157,158, 243,251,259,281,287, 289, 301, 307, 308, 318, 363, 364, 370, 465, 475,529, 531, 533, 551, 624, 644,723

# Hashtable

The hash table is the most popular data structure in the real world. It can be used in the following scenarios.

1. used as a sparse array and you are not sure the index range.
2. find out repeat pattern.
3. count the items.
4. remember position
5. check the sum (or difference) of any of two items (or a range) in an array as k
6. index and revert index

### **325. Maximum Size Subarray Sum Equals k**

Given an array *nums* and a target value *k*, find the maximum length of a subarray that sums to *k*. If there isn't one, return 0 instead.

**Note:**  
The sum of the entire *nums* array is guaranteed to fit within the 32-bit signed integer range.

**Example 1:**

Given *nums* = [1, -1, 5, -2, 3], *k* = 3,  
return 4. (because the subarray [1, -1, 5, -2] sums to 3 and is the longest)

**Example 2:**

Given *nums* = [-2, -1, 2, 1], *k* = 1,  
return 2. (because the subarray [-1, 2] sums to 1 and is the longest)

**Follow Up:**  
Can you do it in O(*n*) time?

Hint: if two numbers add up is S, then if you have number as X, you can find out another number if exist, by looking up S-X in the hashtable.

int LeetCode::maxSubArrayLen(vector<int>& nums, int k)

{

unordered\_map<int, int> sum\_map;

int sum = 0;

int max\_length = 0;

sum\_map[0] = -1;

for (int i = 0; i < (int)nums.size(); i++)

{

sum += nums[i];

if (sum\_map.find(sum - k) != sum\_map.end())

{

max\_length = max(max\_length, i - sum\_map[sum-k]);

}

if (sum\_map.find(sum) == sum\_map.end())

{

sum\_map[sum] = i;

}

}

return max\_length;

}

### **314. Binary Tree Vertical Order Traversal**

Given a binary tree, return the *vertical order* traversal of its nodes' values. (ie, from top to bottom, column by column).

If two nodes are in the same row and column, the order should be from **left to right**.

**Examples:**

1. Given binary tree [3,9,20,null,null,15,7],

3

/\

/ \

9 20

/\

/ \

15 7

return its vertical order traversal as:

[

[9],

[3,15],

[20],

[7]

]

1. Given binary tree [3,9,8,4,0,1,7],

3

/\

/ \

9 8

/\ /\

/ \/ \

4 01 7

return its vertical order traversal as:

[

[4],

[9],

[3,0,1],

[8],

[7]

]

1. Given binary tree [3,9,8,4,0,1,7,null,null,null,2,5] (0's right child is 2 and 1's left child is 5),

3

/\

/ \

9 8

/\ /\

/ \/ \

4 01 7

/\

/ \

5 2

return its vertical order traversal as:

[

[4],

[9,5],

[3,0,1],

[8,2],

[7]

]

/// <summary>

/// Leet code #314. Binary Tree Vertical Order Traversal

///

/// Given a binary tree, return the vertical order traversal of its nodes' values.

/// (ie, from top to bottom, column by column).

/// If two nodes are in the same row and column, the order should be from left

/// to right.

/// Examples:

/// 1.Given binary tree [3,9,20,null,null,15,7],

/// 3

/// / \

/// 9 20

/// / \

/// 15 7

/// return its vertical order traversal as:

///

/// [

/// [9],

/// [3,15],

/// [20],

/// [7]

/// ]

///

/// 2.Given binary tree [3,9,8,4,0,1,7],

/// 3

/// / \

/// 9 8

/// / \ / \

/// 4 01 7

/// return its vertical order traversal as:

/// [

/// [4],

/// [9],

/// [3,0,1],

/// [8],

/// [7]

/// ]

///

/// 3.Given binary tree [3,9,8,4,0,1,7,null,null,null,2,5] (0's right child is 2 and 1's left child is 5),

/// 3

/// / \

/// 9 8

/// / \ / \

/// 4 01 7

/// / \

/// 5 2

/// return its vertical order traversal as:

/// [

/// [4],

/// [9,5],

/// [3,0,1],

/// [8,2],

/// [7]

/// ]

/// </summary>

vector<vector<int>> LeetCode::verticalOrder(TreeNode\* root)

{

vector<vector<int>> result;

// since index can be negative, use map to remember it

map<int, vector<int>> node\_map;

queue<pair<TreeNode\*, int>> process\_queue;

process\_queue.push(make\_pair(root, 0));

// BFS to traverse tree

while (!process\_queue.empty())

{

pair<TreeNode\*, int> node\_info = process\_queue.front();

process\_queue.pop();

TreeNode \* node = node\_info.first;

int index = node\_info.second;

if (node == nullptr) continue;

else node\_map[index].push\_back(node->val);

process\_queue.push(make\_pair(node->left, index - 1));

process\_queue.push(make\_pair(node->right, index + 1));

}

for (map<int, vector<int>>::iterator itr = node\_map.begin(); itr != node\_map.end(); ++itr)

{

result.push\_back(itr->second);

}

return result;

}

### **128. Longest Consecutive Sequence**

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

For example,  
Given [100, 4, 200, 1, 3, 2],  
The longest consecutive elements sequence is [1, 2, 3, 4]. Return its length: 4.

Your algorithm should run in O(*n*) complexity.

Hint: remember dividend in hashtable,

/// <summary>

/// Leet code #128. Longest Consecutive Sequence

/// Given an unsorted array of integers, find the length of the longest

/// consecutive elements sequence.

/// For example,

/// Given [100, 4, 200, 1, 3, 2],

/// The longest consecutive elements sequence is [1, 2, 3, 4].

/// Return its length: 4.

/// Your algorithm should run in O(n) complexity.

/// </summary>

int LeetCode::longestConsecutive(vector<int>& nums)

{

unordered\_set<int> set;

int max\_length = 0;

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

{

set.insert(nums[i]);

}

while (!set.empty())

{

int length = 1;

int number = \*set.begin();

set.erase(number);

int index = -1;

while (set.find(number + index) != set.end())

{

set.erase(number + index);

length++;

index--;

}

index = 1;

while (set.find(number + index) != set.end())

{

set.erase(number + index);

length++;

index++;

}

max\_length = max(max\_length, length);

}

return max\_length;

}

### **166. Fraction to Recurring Decimal**

Given two integers representing the numerator and denominator of a fraction, return the fraction in string format.

If the fractional part is repeating, enclose the repeating part in parentheses.

For example,

* Given numerator = 1, denominator = 2, return "0.5".
* Given numerator = 2, denominator = 1, return "2".
* Given numerator = 2, denominator = 3, return "0.(6)".

Hint: keep the remainder in hashtable with position, when repeated, quotient digits are repeated.

string LeetCode::fractionToDecimal(int numerator, int denominator)

{

string result;

vector<long long> decimals;

unordered\_map<long long, int> map;

if (denominator == 0) return "overflow";

if (numerator == 0) return "0";

int sign = ((numerator > 0) ^ (denominator > 0)) ? -1 : 1;

if (sign < 0) result = "-";

long long long\_numerator = abs((long long)numerator);

long long long\_denominator = abs((long long)denominator);

long long dividend = long\_numerator / long\_denominator;

long long reminder = long\_numerator % long\_denominator;

result.append(to\_string(dividend));

if (reminder != 0) result.append(".");

int pos = 0;

int repeat\_pos = -1;

while (reminder != 0)

{

if (map.find(reminder) != map.end())

{

repeat\_pos = map[reminder];

break;

}

else

{

map[reminder] = pos;

}

reminder = reminder \* 10;

dividend = reminder / long\_denominator;

decimals.push\_back(dividend);

reminder = reminder % long\_denominator;

pos++;

}

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

{

if (i == repeat\_pos)

{

result.append("(");

}

result.append(to\_string(decimals[i]));

}

if (repeat\_pos != -1) result.append(")");

return result;

}

Exercise

1. Please complete the following problems, no pay user please skip locked problem

beginner: 1, 3, 49, 202, 205, 242, 290, 347, 387,389, 442,451,500,506,645,692,697,734

advanced: 18, 128, 159, 166, 220, 438, 447,523,560,748

Locked and Optional: 15, 80, 170, 217, 219, 244, 245, 246, 266, 299, 314, 325,383,409,548,609,657,676,745

# Union Find

Union find is to group many items into one group as long as it has some relationship with another item in that group. Such relationship may be transitive, for example, if A and B are in same group and B and C are in same group then A and C must be in same group.

A typical solution for union find is easy, have a hash table which map the item to its id or value. By default, each item point to itself, this is to say each item are in its own silo, when we have two items are in same group, we simply point one item to another. But here is a catch, after some grouping, the item may not point to itself, but to someone else. So we should change the above statement to that we point the root of one item to the root of another. What is the root? The item pointing to itself is a root. So the common pattern for union find is below:

// find the root of the first word

while (similar\_words[first] != first) first = similar\_words[first];

// find the root of the second word

while (similar\_words[second] != second) second = similar\_words[second];

// point the second word to the first

similar\_words[second] = first;

### **547. Friend Circles**

There are **N** students in a class. Some of them are friends, while some are not. Their friendship is transitive in nature. For example, if A is a **direct** friend of B, and B is a **direct** friend of C, then A is an **indirect** friend of C. And we defined a friend circle is a group of students who are direct or indirect friends.

Given a **N\*N** matrix **M** representing the friend relationship between students in the class. If M[i][j] = 1, then the ith and jth students are **direct** friends with each other, otherwise not. And you have to output the total number of friend circles among all the students.

**Example 1:**

**Input:**

[[1,1,0],

[1,1,0],

[0,0,1]]

**Output:** 2

**Explanation:**The 0th and 1st students are direct friends, so they are in a friend circle.   
The 2nd student himself is in a friend circle. So return 2.

**Example 2:**

**Input:**

[[1,1,0],

[1,1,1],

[0,1,1]]

**Output:** 1

**Explanation:**The 0th and 1st students are direct friends, the 1st and 2nd students are direct friends,   
so the 0th and 2nd students are indirect friends. All of them are in the same friend circle, so return 1.

**Note:**

1. N is in range [1,200].
2. M[i][i] = 1 for all students.
3. If M[i][j] = 1, then M[j][i] = 1.

/// <summary>

/// Leet code #547. Friend Circles

///

/// There are N students in a class. Some of them are friends, while some

/// are not. Their friendship is transitive in nature. For example, if A

/// is a direct friend of B, and B is a direct friend of C, then A is an

/// indirect friend of C. And we defined a friend circle is a group of

/// students who are direct or indirect friends.

/// Given a N\*N matrix M representing the friend relationship between

/// students in the class. If M[i][j] = 1, then the ith and jth students

/// are direct friends with each other, otherwise not. And you have to

/// output the total number of friend circles among all the students.

/// Example 1:

/// Input:

/// [[1,1,0],

/// [1,1,0],

/// [0,0,1]]

/// Output: 2

/// Explanation:The 0th and 1st students are direct friends, so they are

/// in a friend circle.

/// The 2nd student himself is in a friend circle. So return 2.

/// Example 2:

/// Input:

/// [[1,1,0],

/// [1,1,1],

/// [0,1,1]]

/// Output: 1

/// Explanation:The 0th and 1st students are direct friends, the 1st and

/// 2nd students are direct friends,

/// so the 0th and 2nd students are indirect friends. All of them are in

/// the same friend circle, so return 1.

///

/// Note:

/// N is in range [1,200].

/// M[i][i] = 1 for all students.

/// If M[i][j] = 1, then M[j][i] = 1.

/// </summary>

int LeetCode::findCircleNum(vector<vector<int>>& M)

{

vector<int> circle\_map(M.size());

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

{

circle\_map[i] = i;

}

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

{

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

{

if (i == j) continue;

if (M[i][j] == 1)

{

// fine the root of both source and target and union them by

// pointing target to the source

int source = i;

int target = j;

while (circle\_map[source] != source) source = circle\_map[source];

while (circle\_map[target] != target) target = circle\_map[target];

circle\_map[target] = source;

}

}

}

int count = 0;

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

{

if (circle\_map[i] == i) count++;

}

return count;

}

### **737. Sentence Similarity II**

Given two sentences words1, words2 (each represented as an array of strings), and a list of similar word pairs pairs, determine if two sentences are similar.

For example, words1 = ["great", "acting", "skills"] and words2 = ["fine", "drama", "talent"] are similar, if the similar word pairs are pairs = [["great", "good"], ["fine", "good"], ["acting","drama"], ["skills","talent"]].

Note that the similarity relation **is** transitive. For example, if "great" and "good" are similar, and "fine" and "good" are similar, then "great" and "fine" **are similar**.

Similarity is also symmetric. For example, "great" and "fine" being similar is the same as "fine" and "great" being similar.

Also, a word is always similar with itself. For example, the sentences words1 = ["great"], words2 = ["great"], pairs = [] are similar, even though there are no specified similar word pairs.

Finally, sentences can only be similar if they have the same number of words. So a sentence like words1 = ["great"] can never be similar to words2 = ["doubleplus","good"].

**Note:**

 The length of words1 and words2 will not exceed 1000.

 The length of pairs will not exceed 2000.

 The length of each pairs[i] will be 2.

 The length of each words[i] and pairs[i][j] will be in the range [1, 20].

/// <summary>

/// Leet code #737. Sentence Similarity II

///

/// Given two sentences words1, words2 (each represented as an array of

/// strings), and a list of similar word pairs pairs, determine if two

/// sentences are similar.

///

/// For example, words1 = ["great", "acting", "skills"] and words2 =

/// ["fine", "drama", "talent"] are similar, if the similar word pairs

/// are pairs = [["great", "good"], ["fine", "good"], ["acting","drama"],

/// ["skills","talent"]].

///

/// Note that the similarity relation is transitive. For example, if

/// "great" and "good" are similar, and "fine" and "good" are similar,

/// then "great" and "fine" are similar.

///

/// Similarity is also symmetric. For example, "great" and "fine" being

/// similar is the same as "fine" and "great" being similar.

///

/// Also, a word is always similar with itself. For example, the sentences

/// words1 = ["great"], words2 = ["great"], pairs = [] are similar, even

/// though there are no specified similar word pairs.

///

/// Finally, sentences can only be similar if they have the same number of

/// words. So a sentence like words1 = ["great"] can never be similar to

/// words2 = ["doubleplus","good"].

///

/// Note:

///

/// The length of words1 and words2 will not exceed 1000.

/// The length of pairs will not exceed 2000.

/// The length of each pairs[i] will be 2.

/// The length of each words[i] and pairs[i][j] will be in the range [1, 20].

/// </summary>

bool LeetCode::areSentencesSimilarTwo(vector<string>& words1, vector<string>& words2,

vector<pair<string, string>> pairs)

{

if (words1.size() != words2.size()) return false;

unordered\_map<string, string> similar\_words;

for (auto itr : pairs)

{

string first = itr.first;

// insert the first word if not exist

if (similar\_words.count(first) == 0) similar\_words[first] = first;

// find the root of the first word

while (similar\_words[first] != first) first = similar\_words[first];

string second = itr.second;

// insert the second word if not exist

if (similar\_words.count(second) == 0) similar\_words[second] = second;

// find the root of the second word

while (similar\_words[second] != second) second = similar\_words[second];

// point the second word to the first

similar\_words[second] = first;

}

for (size\_t i = 0; i < words1.size(); i++)

{

string first = words1[i];

// find the root of first word

while (similar\_words[first] != first) first = similar\_words[first];

string second = words2[i];

// find the root of second word

while (similar\_words[second] != second) second = similar\_words[second];

if (first != second) return false;

}

return true;

}

Exercise

1. Please complete the following problems, no pay user please skip locked problem

advanced: 305,547,721,737

# Interval and Schedule

In my last lecture, the interval count problem is part of a greedy problem. But it looks that there are quite some similar problems in leetcode and it deserve to have it discussed separately.

To solve such problem, the first question to answer is that how do you want to store the intervals? The answer is that there are two ways of storing the interval. The first way is straight forward, you have the interval stored as pair, and you store the pairs in an ordered way (C++ implement the comparator for pair, but no hash functions), normally it is a BST. But you need to split the interval into the ones which do not have overlap, this will make sure that you know which existing intervals in the BST tree may have overlap with the new input one.

Let’s look at the following example.

The key search is to find a range the starting of which is equal or less than the input range starting point and work from it. The lower\_bound function can be used for this purpose, although the lower\_bound is designed to find thing which is greater or equal than the input.

Please notice there are 3 cases here,

* The starting range is complete fall ahead of the input range, in this case the starting range is igored.
* The starting range is overlapped with the input range, in this case the second half of the starting range is cut.
* The starting range fully cover the input range, in this case the starting range will be cut into 3 pieces, the middle one is replaced by the input range.

### **715. Range Module**

A Range Module is a module that tracks ranges of numbers. Your task is to design and implement the following interfaces in an efficient manner.

 addRange(int left, int right) Adds the half-open interval [left, right), tracking every real number in that interval. Adding an interval that partially overlaps with currently tracked numbers should add any numbers in the interval [left, right) that are not already tracked.

 queryRange(int left, int right) Returns true if and only if every real number in the interval [left, right) is currently being tracked.

 removeRange(int left, int right) Stops tracking every real number currently being tracked in the interval [left, right).

**Example 1:**

**addRange(10, 20)**: null

**removeRange(14, 16)**: null

**queryRange(10, 14)**: true (Every number in [10, 14) is being tracked)

**queryRange(13, 15)**: false (Numbers like 14, 14.03, 14.17 in [13, 15) are not being tracked)

**queryRange(16, 17)**: true (The number 16 in [16, 17) is still being tracked, despite the remove operation)

**Note:**

 A half open interval [left, right) denotes all real numbers left <= x < right.

 0 < left < right < 10^9 in all calls to addRange, queryRange, removeRange.

 The total number of calls to addRange in a single test case is at most 1000.

 The total number of calls to queryRange in a single test case is at most 5000.

 The total number of calls to removeRange in a single test case is at most 1000.

/// <summary>

/// Leet code #715. Range Module

///

/// A Range Module is a module that tracks ranges of numbers. Your task is

/// to design and implement the following interfaces in an efficient

/// manner.

///

/// addRange(int left, int right) Adds the half-open interval

/// [left, right), tracking every real number in that interval. Adding

/// an interval that partially overlaps with currently tracked numbers

/// should add any numbers in the interval [left, right) that are not

/// already tracked.

/// queryRange(int left, int right) Returns true if and only if every real

/// number in the interval [left, right) is currently being tracked.

/// removeRange(int left, int right) Stops tracking every real number

/// currently being tracked in the interval [left, right).

///

/// Example 1:

/// addRange(10, 20): null

/// removeRange(14, 16): null

/// queryRange(10, 14): true (Every number in [10, 14) is being tracked)

/// queryRange(13, 15): false (Numbers like 14, 14.03, 14.17 in [13, 15)

/// are not being tracked)

/// queryRange(16, 17): true (The number 16 in [16, 17) is still being

/// tracked, despite the remove operation)

/// Note:

///

/// A half open interval [left, right) denotes all real numbers

/// left <= x < right.

/// 0 < left < right < 10^9 in all calls to addRange, queryRange,

/// removeRange.

/// The total number of calls to addRange in a single test case is at most

/// 1000.

/// The total number of calls to queryRange in a single test case is at

/// most 5000.

/// The total number of calls to removeRange in a single test case is at

/// most 1000.

/// </summary>

class RangeModule

{

private:

set<pair<int, int>> m\_Range;

public:

RangeModule()

{

}

void addRange(int left, int right)

{

pair<int, int> range = make\_pair(left, right);

if (m\_Range.empty())

{

m\_Range.insert(range);

return;

}

auto itr = m\_Range.lower\_bound(range);

// the previous range may need to be adjusted.

if (itr != m\_Range.begin()) itr--;

// loop from small to large range

while (itr != m\_Range.end() && itr->first <= range.second)

{

auto temp = itr;

itr++;

if (temp->second >= range.first)

{

range.first = min(temp->first, range.first);

range.second = max(temp->second, range.second);

m\_Range.erase(temp);

}

}

m\_Range.insert(range);

}

bool queryRange(int left, int right)

{

pair<int, int> range = make\_pair(left, right);

if (m\_Range.empty())

{

return false;

}

auto itr = m\_Range.lower\_bound(range);

// the previous range may need to be adjusted.

if (itr != m\_Range.begin()) itr--;

// loop from small to large range

while (itr != m\_Range.end() && itr->first < range.second)

{

if (itr->second <= range.first)

{

itr++;

}

else if ((itr->first <= range.first) && (itr->second >= range.second))

{

return true;

}

// if (itr->first > range.first) || (itr->second < range.second)

else

{

return false;

}

}

return false;

}

void removeRange(int left, int right)

{

pair<int, int> range = make\_pair(left, right);

if (m\_Range.empty())

{

return;

}

auto itr = m\_Range.lower\_bound(range);

// the previous range may need to be adjusted.

if (itr != m\_Range.begin()) itr--;

// loop from small to large range

while (itr != m\_Range.end() && itr->first < range.second)

{

auto temp = itr;

itr++;

if (temp->second <= range.first)

{

continue;

}

else if (temp->first < range.first)

{

pair<int, int> prev = make\_pair(temp->first, range.first);

pair<int, int> next = make\_pair(range.second, temp->second);

m\_Range.erase(temp);

m\_Range.insert(prev);

if (next.second > next.first) m\_Range.insert(next);

}

else if (temp->second <= range.second)

{

m\_Range.erase(temp);

}

// if (temp->first < range.second) && (temp->second > range.second)

else

{

pair<int, int> next = make\_pair(range.second, temp->second);

m\_Range.erase(temp);

m\_Range.insert(next);

}

}

}

};

However, there is another way to store the interval and make it easy to resolve. We store the start and end point only in a 1-D array. Assume we want to represent an interval between X and Y with the value as A, and after Y it is zero, we only need to say array[X] = A and array[Y] = 0. If we look for a point in X-axis with the position of Z, where X<=Z<Y, we just search the position in the X-axis to find any position which is less than or equal to Z, and the value of that position is the value which Z should have.

Please look at the following example:

### **732. My Calendar III**

Implement a MyCalendarThree class to store your events. A new event can **always** be added.

Your class will have one method, book(int start, int end). Formally, this represents a booking on the half open interval [start, end), the range of real numbers x such that start <= x < end.

A *K-booking* happens when **K** events have some non-empty intersection (ie., there is some time that is common to all K events.)

For each call to the method MyCalendar.book, return an integer K representing the largest integer such that there exists a K-booking in the calendar.

Your class will be called like this: MyCalendarThree cal = new MyCalendarThree(); MyCalendarThree.book(start, end)

**Example 1:**

MyCalendarThree();

MyCalendarThree.book(10, 20); // returns 1

MyCalendarThree.book(50, 60); // returns 1

MyCalendarThree.book(10, 40); // returns 2

MyCalendarThree.book(5, 15); // returns 3

MyCalendarThree.book(5, 10); // returns 3

MyCalendarThree.book(25, 55); // returns 3

**Explanation:**

The first two events can be booked and are disjoint, so the maximum K-booking is a 1-booking.

The third event [10, 40) intersects the first event, and the maximum K-booking is a 2-booking.

The remaining events cause the maximum K-booking to be only a 3-booking.

Note that the last event locally causes a 2-booking, but the answer is still 3 because

eg. [10, 20), [10, 40), and [5, 15) are still triple booked.

**Note:**

 The number of calls to MyCalendarThree.book per test case will be at most 400.

 In calls to MyCalendarThree.book(start, end), start and end are integers in the range [0, 10^9].

/// <summary>

/// Leet code #732. My Calendar III

///

/// Implement a MyCalendarThree class to store your events. A new event can

/// always be added.

///

/// Your class will have one method, book(int start, int end). Formally,

/// this represents a booking on the half open interval [start, end), the

/// range of real numbers x such that start <= x < end.

///

/// A K-booking happens when K events have some non-empty intersection

/// (ie., there is some time that is common to all K events.)

///

/// For each call to the method MyCalendar.book, return an integer K

/// representing the largest integer such that there exists a K-booking

/// in the calendar.

///

/// Your class will be called like this: MyCalendarThree cal = new

/// MyCalendarThree(); MyCalendarThree.book(start, end)

///

/// Example 1:

/// MyCalendarThree();

/// MyCalendarThree.book(10, 20); // returns 1

/// MyCalendarThree.book(50, 60); // returns 1

/// MyCalendarThree.book(10, 40); // returns 2

/// MyCalendarThree.book(5, 15); // returns 3

/// MyCalendarThree.book(5, 10); // returns 3

/// MyCalendarThree.book(25, 55); // returns 3

/// Explanation:

/// The first two events can be booked and are disjoint, so the maximum

/// K-booking is a 1-booking.

/// The third event [10, 40) intersects the first event, and the maximum

/// K-booking is a 2-booking.

/// The remaining events cause the maximum K-booking to be only a

/// 3-booking.

/// Note that the last event locally causes a 2-booking, but the answer

/// is still 3 because eg. [10, 20), [10, 40), and [5, 15) are still

/// triple booked.

/// Note:

///

/// The number of calls to MyCalendarThree.book per test case will be at

/// most 400.

/// In calls to MyCalendarThree.book(start, end), start and end are

/// integers in the range [0, 10^9].

/// </summary>

class MyCalendarThree

{

private:

int m\_Book;

map<int, int> m\_TimeMap;

map<int, int>::iterator getLocation(int time\_stamp)

{

auto itr = m\_TimeMap.lower\_bound(time\_stamp);

if (itr == m\_TimeMap.end() || time\_stamp < itr->first)

{

itr--;

}

return itr;

};

public:

MyCalendarThree()

{

m\_TimeMap[0] = 0;

m\_Book = 0;

}

int book(int start, int end)

{

auto itr = getLocation(end);

m\_TimeMap[end] = itr->second;

itr = getLocation(start);

m\_TimeMap[start] = itr->second + 1;

m\_Book = max(m\_Book, m\_TimeMap[start]);

itr = m\_TimeMap.find(start);

// int prev\_value = itr->second;

itr++;

while (itr != m\_TimeMap.end() && itr->first < end)

{

auto temp = itr;

itr->second++;

m\_Book = max(m\_Book, itr->second);

itr++;

/\* clean up duplication is optional

if (temp->second == prev\_value)

{

m\_TimeMap.erase(temp);

}

else

{

prev\_value = temp->second;

}

\*/

}

return m\_Book;

}

};

# Parsing Expression

Parsing expression is usually considered as difficult level, however if you do it properly it may not be that difficult. When you solve such problem, please think what you learn in the course of compiler, in a normal compiler implementation, we will do it in two phases, the lexical analyzing and the syntax parsing.

So the first phase is to read each token, think what are the possible delimiter, for example the variables, operators, numeric constants can be delimited by space or brackets, such as "(" and ")".

As a traditional processing you can also do it by using two stacks, the operand stack and an operator stack. The first solution below is using this method.

### **772. Basic Calculator III**

Implement a basic calculator to evaluate a simple expression string.

The expression string may contain open ( and closing parentheses ), the plus + or minus sign -, **non-negative** integers and empty spaces .

The expression string contains only non-negative integers, +, -, \*, / operators , open ( and closing parentheses ) and empty spaces . The integer division should truncate toward zero.

You may assume that the given expression is always valid. All intermediate results will be in the range of [-2147483648, 2147483647].

Some examples:

"1 + 1" = 2

" 6-4 / 2 " = 4

"2\*(5+5\*2)/3+(6/2+8)" = 21

"(2+6\* 3+5- (3\*14/7+2)\*5)+3"=-12

**Note:** **Do not** use the eval built-in library function.

My first solution is based on two stacks

/// <summary>

/// Leet code #772. Basic Calculator III

/// </summary>

string LeetCode::parseExpressionToken(string s, int& index)

{

string result;

while (index < (int)s.size())

{

if (isalnum(s[index]))

{

result.push\_back(s[index]);

index++;

}

else if (isspace(s[index]))

{

index++;

}

else

{

if (!result.empty()) break;

else

{

result = s[index];

index++;

break;

}

}

}

return result;

}

/// <summary>

/// Leet code #772. Basic Calculator III

/// </summary>

void LeetCode::calculateIntegerExpression(stack<int>& operands, stack<string>& operators)

{

string op = operators.top();

operators.pop();

int number2 = operands.top();

operands.pop();

int number1 = operands.top();

operands.pop();

int result = 0;

if (op == "+")

{

result = number1 + number2;

}

else if (op == "-")

{

result = number1 - number2;

}

else if (op == "\*")

{

result = number1 \* number2;

}

else if (op == "/")

{

result = number1 / number2;

}

operands.push(result);

}

/// <summary>

/// Leet code #772. Basic Calculator III

/// </summary>

int LeetCode::parseIntegerExpression(string& s, int& index)

{

stack<int> operands;

stack<string> operators;

string token;

while (true)

{

token = parseExpressionToken(s, index);

if ((token == ")") || (token.empty()))

{

break;

}

else if (token == "(")

{

int operand = parseIntegerExpression(s, index);

operands.push(operand);

}

else if (isalnum(token[0]))

{

int operand = atoi(token.c\_str());

operands.push(operand);

}

else

{

if ((token == "+") || (token == "-"))

{

while (!operators.empty())

{

calculateIntegerExpression(operands, operators);

}

}

else if ((token == "\*") || (token == "/"))

{

while ((!operators.empty()) && (operators.top() == "\*" || operators.top() == "/"))

{

calculateIntegerExpression(operands, operators);

}

}

operators.push(token);

}

}

while (!operators.empty())

{

calculateIntegerExpression(operands, operators);

}

return operands.top();

}

/// <summary>

/// Leet code #772. Basic Calculator III

///

/// Implement a basic calculator to evaluate a simple expression string.

/// The expression string may contain open ( and closing parentheses ),

/// the plus + or minus sign -, non-negative integers and empty spaces.

///

/// The expression string contains only non-negative integers, +, -, \*, /

/// operators , open ( and closing parentheses ) and empty spaces .

/// The integer division should truncate toward zero.

///

/// You may assume that the given expression is always valid. All

/// intermediate results will be in the range of [-2147483648, 2147483647].

///

/// Some examples:

///

/// "1 + 1" = 2

/// " 6-4 / 2 " = 4

/// "2\*(5+5\*2)/3+(6/2+8)" = 21

/// "(2+6\* 3+5- (3\*14/7+2)\*5)+3"=-12

/// </summary>

int LeetCode::calculateIII(string s)

{

int index = 0;

return parseIntegerExpression(s, index);

}

# Deterministic finite automaton

From the tokens, after you know what it is, you can build the syntax analyzer by some recusive function call, for example, in an arithmetic expression, you may have

1. Term = <digits> | "(" Expr ")"
2. Factor = Term( ( "\*" | "/") Term)\*
3. Expr = Factor (("+" | "-") Factor)\*

You expect an expression first, then an expression can be made of multiple factor, a factor can be made of multiple terms. A term, if you see "(", will recursively enter the expression parsing.

If you use this method, please remember you may need to have a function to look up ahead, which means you just look at what is the next operator, but not fetch it.

/// <summary>

/// Leet code #772. Basic Calculator III

/// </summary>

string LeetCode::parseExpressionToken(string s, int& index)

{

string result;

while (index < (int)s.size())

{

if (isalnum(s[index]))

{

result.push\_back(s[index]);

index++;

}

else if (isspace(s[index]))

{

index++;

}

else

{

if (!result.empty()) break;

else

{

result = s[index];

index++;

break;

}

}

}

return result;

}

/// <summary>

/// Leet code #772. Basic Calculator III

/// </summary>

int LeetCode::parseIntegerTerm(vector<string>& tokens, int& index)

{

int result;

if (isdigit(tokens[index][0]))

{

result = atoi(tokens[index].c\_str());

}

else if (tokens[index] == "(")

{

index++;

result = parseIntegerExpression(tokens, index);

// when return with ")" left there

}

index++;

return result;

}

/// <summary>

/// Leet code #772. Basic Calculator III

/// </summary>

int LeetCode::parseIntegerFactor(vector<string>& tokens, int& index)

{

int result = parseIntegerTerm(tokens, index);

while (index < (int)tokens.size())

{

if (tokens[index] == "\*")

{

index++;

result \*= parseIntegerTerm(tokens, index);

}

else if (tokens[index] == "/")

{

index++;

result /= parseIntegerTerm(tokens, index);

}

else

{

break;

}

}

return result;

}

/// <summary>

/// Leet code #772. Basic Calculator III

/// </summary>

int LeetCode::parseIntegerExpression(vector<string>& tokens, int& index)

{

int result = parseIntegerFactor(tokens, index);

while (index < (int)tokens.size())

{

if (tokens[index] == "+")

{

index++;

result += parseIntegerFactor(tokens, index);

}

else if (tokens[index] == "-")

{

index++;

result -= parseIntegerFactor(tokens, index);

}

else

{

break;

}

}

return result;

}

/// <summary>

/// Leet code #772. Basic Calculator III

///

/// Implement a basic calculator to evaluate a simple expression string.

/// The expression string may contain open ( and closing parentheses ),

/// the plus + or minus sign -, non-negative integers and empty spaces.

///

/// The expression string contains only non-negative integers, +, -, \*, /

/// operators , open ( and closing parentheses ) and empty spaces .

/// The integer division should truncate toward zero.

///

/// You may assume that the given expression is always valid. All

/// intermediate results will be in the range of [-2147483648, 2147483647].

///

/// Some examples:

///

/// "1 + 1" = 2

/// " 6-4 / 2 " = 4

/// "2\*(5+5\*2)/3+(6/2+8)" = 21

/// "(2+6\* 3+5- (3\*14/7+2)\*5)+3"=-12

/// </summary>

int LeetCode::calculateIII\_V2(string s)

{

vector<string> tokens;

int index = 0;

while (index < (int)s.size())

{

tokens.push\_back(parseExpressionToken(s, index));

}

index = 0;

int result = parseIntegerExpression(tokens, index);

return result;

}

### **736. Parse Lisp Expression**

You are given a string expression representing a Lisp-like expression to return the integer value of.

The syntax for these expressions is given as follows.

 An expression is either an integer, a let-expression, an add-expression, a mult-expression, or an assigned variable. Expressions always evaluate to a single integer.

 (An integer could be positive or negative.)

 A let-expression takes the form (let v1 e1 v2 e2 ... vn en expr), where let is always the string "let", then there are 1 or more pairs of alternating variables and expressions, meaning that the first variable v1 is assigned the value of the expression e1, the second variable v2 is assigned the value of the expression e2, and so on **sequentially**; and then the value of this let-expression is the value of the expression expr.

 An add-expression takes the form (add e1 e2) where add is always the string "add", there are always two expressions e1, e2, and this expression evaluates to the addition of the evaluation of e1 and the evaluation of e2.

 A mult-expression takes the form (mult e1 e2) where mult is always the string "mult", there are always two expressions e1, e2, and this expression evaluates to the multiplication of the evaluation of e1 and the evaluation of e2.

 For the purposes of this question, we will use a smaller subset of variable names. A variable starts with a lowercase letter, then zero or more lowercase letters or digits. Additionally for your convenience, the names "add", "let", or "mult" are protected and will never be used as variable names.

 Finally, there is the concept of scope. When an expression of a variable name is evaluated, **within the context of that evaluation**, the innermost scope (in terms of parentheses) is checked first for the value of that variable, and then outer scopes are checked sequentially. It is guaranteed that every expression is legal. Please see the examples for more details on scope.

**Evaluation Examples:**

**Input:** (add 1 2)

**Output:** 3

**Input:** (mult 3 (add 2 3))

**Output:** 15

**Input:** (let x 2 (mult x 5))

**Output:** 10

**Input:** (let x 2 (mult x (let x 3 y 4 (add x y))))

**Output:** 14

**Explanation:** In the expression (add x y), when checking for the value of the variable x,

we check from the innermost scope to the outermost in the context of the variable we are trying to evaluate.

Since x = 3 is found first, the value of x is 3.

**Input:** (let x 3 x 2 x)

**Output:** 2

**Explanation:** Assignment in let statements is processed sequentially.

**Input:** (let x 1 y 2 x (add x y) (add x y))

**Output:** 5

**Explanation:** The first (add x y) evaluates as 3, and is assigned to x.

The second (add x y) evaluates as 3+2 = 5.

**Input:** (let x 2 (add (let x 3 (let x 4 x)) x))

**Output:** 6

**Explanation:** Even though (let x 4 x) has a deeper scope, it is outside the context

of the final x in the add-expression. That final x will equal 2.

**Input:** (let a1 3 b2 (add a1 1) b2)

**Output** 4

**Explanation:** Variable names can contain digits after the first character.

**Note:**

 The given string expression is well formatted: There are no leading or trailing spaces, there is only a single space separating different components of the string, and no space between adjacent parentheses. The expression is guaranteed to be legal and evaluate to an integer.

 The length of expression is at most 2000. (It is also non-empty, as that would not be a legal expression.)

 The answer and all intermediate calculations of that answer are guaranteed to fit in a 32-bit integer.

/// <summary>

/// Leet code #736. Parse Lisp Expression

/// </summary>

string LeetCode::parseLispToken(string& expression, int& index)

{

string result;

while (index < (int)expression.size())

{

if (isspace(expression[index]))

{

index++;

if (!result.empty()) break;

}

else if (expression[index] == '(')

{

if (!result.empty()) break;

else

{

result = "(";

index++;

break;

}

}

else if (expression[index] == ')')

{

if (!result.empty()) break;

else

{

result = ")";

index++;

break;

}

}

else

{

result.push\_back(expression[index]);

index++;

}

}

return result;

}

/// <summary>

/// Leet code #736. Parse Lisp Expression

/// </summary>

int LeetCode::calculateLispCommand(string& command, vector<string>& parameters, unordered\_map<string, int>&variables)

{

int result = 0;

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

{

if (islower(parameters[i][0]))

{

parameters[i] = to\_string(variables[parameters[i]]);

}

}

if (command == "add")

{

result = atoi(parameters[0].c\_str()) + atoi(parameters[1].c\_str());

}

else if (command == "mult")

{

result = atoi(parameters[0].c\_str()) \* atoi(parameters[1].c\_str());

}

else if (command == "let")

{

result = atoi(parameters[0].c\_str());

}

return result;

}

/// <summary>

/// Leet code #736. Parse Lisp Expression

/// </summary>

int LeetCode::processLispCommand(string& expression, int& index, unordered\_map<string, int> variables)

{

string command;

vector<string> parameters;

while (index < (int)expression.size())

{

string token = parseLispToken(expression, index);

if (token == ")")

{

if (!command.empty()) break;

}

else if (token == "(")

{

if (!command.empty())

{

int parameter = processLispCommand(expression, index, variables);

parameters.push\_back(to\_string(parameter));

}

}

else if (command.empty())

{

command = token;

}

else

{

parameters.push\_back(token);

}

if ((command == "let") && (parameters.size() == 2))

{

if (islower(parameters[1][0]))

{

variables[parameters[0]] = variables[parameters[1]];

}

else

{

variables[parameters[0]] = atoi(parameters[1].c\_str());

}

parameters.clear();

}

}

return calculateLispCommand(command, parameters, variables);

}

/// <summary>

/// Leet code #736. Parse Lisp Expression

///

/// You are given a string expression representing a Lisp-like expression to

/// return the integer value of.

///

/// The syntax for these expressions is given as follows.

///

/// An expression is either an integer, a let-expression, an add-expression,

/// a mult-expression, or an assigned variable. Expressions always evaluate to

/// a single integer.

/// (An integer could be positive or negative.)

/// A let-expression takes the form (let v1 e1 v2 e2 ... vn en expr), where

/// let is always the string "let", then there are 1 or more pairs of

/// alternating variables and expressions, meaning that the first variable v1

/// is assigned the value of the expression e1, the second variable v2 is

/// assigned the value of the expression e2, and so on sequentially; and then

/// the value of this let-expression is the value of the expression expr.

///

/// An add-expression takes the form (add e1 e2) where add is always the string

/// "add", there are always two expressions e1, e2, and this expression

/// evaluates to the addition of the evaluation of e1 and the evaluation of e2.

///

/// A mult-expression takes the form (mult e1 e2) where mult is always the

/// string "mult", there are always two expressions e1, e2, and this expression

/// evaluates to the multiplication of the evaluation of e1 and the evaluation

/// of e2.

///

/// For the purposes of this question, we will use a smaller subset of variable

/// names. A variable starts with a lowercase letter, then zero or more

/// lowercase letters or digits. Additionally for your convenience, the names

/// "add", "let", or "mult" are protected and will never be used as variable

/// names.

/// Finally, there is the concept of scope. When an expression of a variable

/// name is evaluated, within the context of that evaluation, the innermost

/// scope (in terms of parentheses) is checked first for the value of that

/// variable, and then outer scopes are checked sequentially. It is guaranteed

/// that every expression is legal. Please see the examples for more details on

/// scope.

/// Evaluation Examples:

/// Input: (add 1 2)

/// Output: 3

///

/// Input: (mult 3 (add 2 3))

/// Output: 15

///

/// Input: (let x 2 (mult x 5))

/// Output: 10

///

/// Input: (let x 2 (mult x (let x 3 y 4 (add x y))))

/// Output: 14

/// Explanation: In the expression (add x y), when checking for the value of

/// the variable x,

/// we check from the innermost scope to the outermost in the context of the

/// variable we are trying to evaluate.

/// Since x = 3 is found first, the value of x is 3.

///

/// Input: (let x 3 x 2 x)

/// Output: 2

/// Explanation: Assignment in let statements is processed sequentially.

///

/// Input: (let x 1 y 2 x (add x y) (add x y))

/// Output: 5

/// Explanation: The first (add x y) evaluates as 3, and is assigned to x.

/// The second (add x y) evaluates as 3+2 = 5.

///

/// Input: (let x 2 (add (let x 3 (let x 4 x)) x))

/// Output: 6

/// Explanation: Even though (let x 4 x) has a deeper scope, it is outside

/// the context

/// of the final x in the add-expression. That final x will equal 2.

///

/// Input: (let a1 3 b2 (add a1 1) b2)

/// Output 4

/// Explanation: Variable names can contain digits after the first character.

///

/// Note:

///

/// 1. The given string expression is well formatted: There are no leading or

/// trailing spaces, there is only a single space separating different

/// components of the string, and no space between adjacent parentheses.

/// 2. The expression is guaranteed to be legal and evaluate to an integer.

/// 3. The length of expression is at most 2000. (It is also non-empty, as that

/// would not be a legal expression.)

/// 4. The answer and all intermediate calculations of that answer are guaranteed

/// to fit in a 32-bit integer.

/// </summary>

int LeetCode::evaluate(string expression)

{

int index = 0;

unordered\_map<string, int> variables;

return processLispCommand(expression, index, variables);

}

## Deterministic finite automata

We can also try automaton here, but the let command is not a simple automata, it is some kind of push down automata, only we saw the end of command “)”, we will know the term is an variable or expression which should return value. For such scenario, normally you need to use stack to process it.

/// <summary>

/// Leet code #736. Parse Lisp Expression

/// </summary>

string LeetCode::parseLispToken(string& expression, int& index)

{

string result;

while (index < (int)expression.size())

{

if (isspace(expression[index]))

{

index++;

if (!result.empty()) break;

}

else if ((expression[index] == '(') || (expression[index] == ')'))

{

if (!result.empty()) break;

else

{

result.push\_back(expression[index]);

index++;

break;

}

}

else

{

result.push\_back(expression[index]);

index++;

}

}

return result;

}

/// <summary>

/// Leet code #736. Parse Lisp Expression

/// </summary>

int LeetCode::processLispExpression(vector<string>& tokens, int& index, unordered\_map<string, int> variables)

{

int result;

if (islower(tokens[index][0]))

{

result = variables[tokens[index]];

}

else if (tokens[index] == "(")

{

result = processLispCommand(tokens, index, variables);

}

else

{

result = atoi(tokens[index].c\_str());

}

index++;

return result;

}

/// <summary>

/// Leet code #736. Parse Lisp Expression

/// </summary>

int LeetCode::processLispCommand(vector<string>& tokens, int& index, unordered\_map<string, int> variables)

{

if (tokens[index] == "(") index++;

string command = tokens[index];

index++;

int result = 0;

while (tokens[index] != ")")

{

if (command == "add")

{

result = processLispExpression(tokens, index, variables) +

processLispExpression(tokens, index, variables);

}

else if (command == "mult")

{

result = processLispExpression(tokens, index, variables) \*

processLispExpression(tokens, index, variables);

}

else if (command == "let")

{

int count = 0;

string variable;

int value;

while (tokens[index] != ")")

{

if (islower(tokens[index][0]) && (count == 0))

{

variable = tokens[index];

value = variables[variable];

index++;

}

else

{

value = processLispExpression(tokens, index, variables);

}

count++;

if (count == 2)

{

variables[variable] = value;

count = 0;

}

}

result = value;

}

}

return result;

}

/// <summary>

/// Leet code #736. Parse Lisp Expression

/// </summary>

int LeetCode::evaluate\_V2(string expression)

{

int index = 0;

vector<string> tokens;

while (index < (int)expression.size())

{

string token = parseLispToken(expression, index);

tokens.push\_back(token);

}

unordered\_map<string, int> variables;

index = 0;

return processLispCommand(tokens, index, variables);

}