Leet Code Basic

This paper is targeted for who want to practice leetcode or other similar coding challenge, we will have same papers for advanced users.

# 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.

### **11. Container With Most Water**

Given *n* non-negative integers *a1*, *a2*, ..., *an*, where each represents a point at coordinate (*i*, *ai*). *n* vertical lines are drawn such that the two endpoints of line *i* is at (*i*, *ai*) and (*i*, 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water.

Note: You may not slant the container and *n* is at least 2.

/// <summary>

/// Leet code #11. Container With Most Water

///

/// Given n non-negative integers a1, a2, ..., an, where each represents

/// a point at coordinate (i, ai). n vertical lines are drawn such that

/// the two endpoints of line i is at (i, ai) and (i, 0).

/// Find two lines, which together with x-axis forms a container, such

/// that the container contains the most water.

/// Note: You may not slant the container.

/// </summary>

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

{

int maxArea = 0;

int first = 0, last = height.size() - 1;

while (first < last)

{

int area = min(height[first], height[last]) \* (last - first);

if (area > maxArea)

{

maxArea = area;

}

if (height[first] < height[last])

{

first++;

}

else

{

last--;

}

}

return maxArea;

}

### **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;

}

# String

### **49. Group Anagrams**

Given an array of strings, group anagrams together.

**Example:**

**Input:** ["eat", "tea", "tan", "ate", "nat", "bat"],

**Output:**

[

["ate","eat","tea"],

["nat","tan"],

["bat"]

]

**Note:**

* All inputs will be in lowercase.
* The order of your output does not matter.

Analysis: To determine whether two strings are anagrams or not, you can use hashtable to count characters then compare, since they are lowercase you can also use an array of 26 cells to count characters.

However there is a more convienent way, because we match the whole word, we can destructively sort each word in alphabet order, after sort all anagrams looks same.

/// <summary>

/// Leet code #49. Group Anagrams

/// Given an array of strings, group anagrams together.

/// For example, given: ["eat", "tea", "tan", "ate", "nat", "bat"],

/// Return:

/// [

/// ["ate", "eat","tea"],

/// ["nat","tan"],

/// ["bat"]

/// ]

/// </summary>

vector<vector<string>> LeetCode::groupAnagrams(vector<string>& strs)

{

vector<vector<string>> result;

unordered\_map<string, int> anagram\_map;

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

{

string str = strs[i];

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

if (anagram\_map.count(str) > 0)

{

result[anagram\_map[str]].push\_back(strs[i]);

}

else

{

anagram\_map[str] = result.size();

result.push\_back({ strs[i] });

}

}

return result;

}

# 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;

}

# 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

### **1. Two Sum**

Given an array of integers, return **indices** of the two numbers such that they add up to a specific target.

You may assume that each input would have **exactly** one solution, and you may not use the same element twice.

**Example:**

Given nums = [2, 7, 11, 15], target = 9,

Because nums[**0**] + nums[**1**] = 2 + 7 = 9,

return [**0**, **1**].

Analysis: The two number are integers, so if the first number is already picked an stored in hashtable, and when we get the second number we can check if the target - nums[i] is already in hashtable.

vector<int> LeetCode::twoSum(vector<int>& nums, int target)

{

vector<int> result;

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

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

{

if (num\_map.count(target - nums[i]) > 0)

{

result = { num\_map[target - nums[i]], i };

return result;

}

num\_map[nums[i]] = i;

}

return result;

}

Follow up:

1. Can I sort the array of number and then use two pointer check from two ends?

[Answer] Yes, you can but it will become a N\*log(N) time complexity.

1. Can I use this method if the numbers are type of double?

[Answer] No, you cannot, the double is not precise by bits.

1. What if there are multiple solutions?

[Answer] You can still use this method, but you can check the value of map to an array (vector in C++) which store the index of the numbers

### **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;

}

# Dynamic Programming

Dynamic programming is the algorithm which calculate based on the stage (partial) result.

Think about the Fibonacci numbers, f(n) = f(n-1) + f(n-2), to calculate every single f(n) value, you need to be based on f(n-1) and f(n-2), since both f(n-1) and f(n-2) appear before f(n). This is called Look back.

### **62. Unique Paths**

A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?

  
Above is a 7 x 3 grid. How many possible unique paths are there?

**Note:** m and n will be at most 100.

**Example 1:**

**Input:** m = 3, n = 2

**Output:** 3

**Explanation:**

From the top-left corner, there are a total of 3 ways to reach the bottom-right corner:

1. Right -> Right -> Down

2. Right -> Down -> Right

3. Down -> Right -> Right

**Example 2:**

**Input:** m = 7, n = 3

**Output:** 28

/// <summary>

/// Leet code #62. Unique Paths

/// A robot is located at the top-left corner of a m x n grid (marked

/// 'Start' in the diagram below).

/// The robot can only move either down or right at any point in time.

/// The robot is trying to reach the bottom-right corner of the grid

/// (marked 'Finish' in the diagram below).

/// How many possible unique paths are there?

/// Above is a 3 x 7 grid. How many possible unique paths are there?

/// Note: m and n will be at most 100.

/// </summary>

int LeetCode::uniquePaths(int m, int n)

{

vector<vector<int>> matrix(m, vector<int>(n));

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

{

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

{

if ((i == 0) && (j == 0))

{

matrix[i][j] = 1;

}

else if (i == 0)

{

matrix[i][j] = matrix[i][j - 1];

}

else if (j == 0)

{

matrix[i][j] = matrix[i - 1][j];

}

else

{

matrix[i][j] = matrix[i][j - 1] + matrix[i - 1][j];

}

}

}

return matrix[m - 1][n - 1];

}

### **63. Unique Paths II**

A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

Now consider if some obstacles are added to the grids. How many unique paths would there be?



An obstacle and empty space is marked as 1 and 0 respectively in the grid.

**Note:** m and n will be at most 100.

**Example 1:**

**Input:**

[

  [0,0,0],

  [0,1,0],

  [0,0,0]

]

**Output:** 2

**Explanation:**

There is one obstacle in the middle of the 3x3 grid above.

There are two ways to reach the bottom-right corner:

1. Right -> Right -> Down -> Down

2. Down -> Down -> Right -> Right

/// <summary>

/// Leet code #63. Unique Paths II

/// Follow up for "Unique Paths":

/// Now consider if some obstacles are added to the grids. How many unique

/// paths would there be?

/// An obstacle and empty space is marked as 1 and 0 respectively in the grid.

/// For example,

/// [

/// [0,0,0],

/// [0,1,0],

/// [0,0,0]

/// ]

/// There is one obstacle in the middle of a 3x3 grid as illustrated below.

/// The total number of unique paths is 2.

/// Note: m and n will be at most 100.

/// </summary>

int LeetCode::uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid)

{

int m = obstacleGrid.size();

int n = obstacleGrid[0].size();

vector<vector<int>> matrix(m, vector<int>(n));

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

{

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

{

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

{

matrix[i][j] = 0;

}

else if ((i == 0) && (j == 0))

{

matrix[i][j] = 1;

}

else if (i == 0)

{

matrix[i][j] = matrix[i][j - 1];

}

else if (j == 0)

{

matrix[i][j] = matrix[i - 1][j];

}

else

{

matrix[i][j] = matrix[i][j - 1] + matrix[i - 1][j];

}

}

}

return matrix[m - 1][n - 1];

}

### **64. Minimum Path Sum**

Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right which minimizes the sum of all numbers along its path.

**Note:** You can only move either down or right at any point in time.

**Example:**

**Input:**

[

  [1,3,1],

[1,5,1],

[4,2,1]

]

**Output:** 7

**Explanation:** Because the path 1→3→1→1→1 minimizes the sum.

/// <summary>

/// Leet code #64. Minimum Path Sum

/// Given a m x n grid filled with non-negative numbers, find a path from top

/// left to bottom right which minimizes the sum of all numbers along its path.

/// Note: You can only move either down or right at any point in time.

/// </summary>

int LeetCode::minPathSum(vector<vector<int>>& grid)

{

vector<vector<int>> matrix(grid.size(), vector<int>(grid[0].size()));

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

{

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

{

if ((i == 0) && (j == 0))

{

matrix[i][j] = grid[i][j];

}

else if (i == 0)

{

matrix[i][j] = matrix[i][j - 1] + grid[i][j];

}

else if (j == 0)

{

matrix[i][j] = matrix[i - 1][j] + grid[i][j];

}

else

{

matrix[i][j] = min(matrix[i - 1][j], matrix[i][j - 1]) + grid[i][j];

}

}

}

return matrix[grid.size() - 1][grid[0].size() - 1];

}

# Binary Search

There two fundamental ideas in the binary search.

1. Assume an array of sorted numbers, if you want to search a specific number, you first select the middle of the array, if the searched number is smaller than the middle number, you go to the first half (0 to middle -1), otherwise if you the number is greater than the middle, you go to the second half.
2. Another way to think this algorithm is that if you can determine that if the specific target is not in this range, you can discard this range.

**Standard binary algorithm**

First let’s look at the standard binary algorithm.

### **35. Search Insert Position**

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You may assume no duplicates in the array.

**Example 1:**

**Input:** [1,3,5,6], 5

**Output:** 2

**Example 2:**

**Input:** [1,3,5,6], 2

**Output:** 1

**Example 3:**

**Input:** [1,3,5,6], 7

**Output:** 4

**Example 4:**

**Input:** [1,3,5,6], 0

**Output:** 0

/// <summary>

/// Leet code #35. Search Insert Position

/// Given a sorted array and a target value, return the index if the target

/// is found. If not, return the index where it would be if it were inserted

/// in order.

/// You may assume no duplicates in the array.

/// Here are few examples.

/// [1,3,5,6], 5 -> 2

/// [1,3,5,6], 2 -> 1

/// [1,3,5,6], 7 -> 4

/// [1,3,5,6], 0 -> 0

/// </summary>

int LeetCode::searchInsert(vector<int>& nums, int target)

{

// assign the first as 0, last can be the last item or out of boundary

int first = 0, last = nums.size();

// middle should start with first

int middle = 0;

// when first == last break out

while (first < last)

{

// check the middle point, make sure out overflow

// please remember this is an int, so always left bias

middle = first + (last - first) / 2;

// if target is located at second half, should go

// beyond middle point to avoid dead loop.

if (target > nums[middle])

{

first = middle + 1;

middle++;

}

else

{

last = middle;

}

}

// you can return middle or first, return first is better

// because no need to keep track middle

return first;

}

# Binary Tree

The most common solution for the binary tree problem is to search the result by tree traversal. There are 3 type of tree traversal, preoder, inorder, and postorder. The easiest way to write tree traverse is using recursive call. Unlike Backtracking, you do not need to be concern on path circle (repeated path), because tree path is single direction, you also do not need to record the path, because we just care current node and two children of the current node.

The most important thing to consider is that what information do we want to return from the two children and such information will be used to judge the result.

A preorder traversal can be as simple as below:

/// <summary>

/// Binary Tree Preorder Traversal with recursive

/// <summary>

void LeetCode::preorderTraversal(TreeNode\* root, vector<int>& output)

{

if (root == nullptr) return;

output.push\_back(root->val);

preorderTraversal(root->left, output);

preorderTraversal(root->right, output);

}

### **110. Balanced Binary Tree**

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as:

a binary tree in which the depth of the two subtrees of every node never differ by more than 1.

**Example 1:**

Given the following tree [3,9,20,null,null,15,7]:

3

/ \

9 20

/ \

15 7

Return true.  
  
**Example 2:**

Given the following tree [1,2,2,3,3,null,null,4,4]:

1

/ \

2 2

/ \

3 3

/ \

4 4

Return false.

/// <summary>

/// Leet code #110. Balanced Binary Tree

/// </summary>

bool LeetCode::isBalanced(TreeNode\* root, int& depth)

{

if (root == nullptr)

{

depth = 0;

return true;

}

else

{

int left\_depth = 0;

int right\_depth = 0;

if (!isBalanced(root->left, left\_depth) ||

!isBalanced(root->right, right\_depth))

{

depth = max(left\_depth, right\_depth) + 1;

return false;

}

else

{

depth = max(left\_depth, right\_depth) + 1;

if (abs(left\_depth - right\_depth) <= 1)

{

return true;

}

else

{

return false;

}

}

}

}

/// <summary>

/// Leet code #110. Balanced Binary Tree

/// Given a binary tree, determine if it is height-balanced.

/// For this problem, a height-balanced binary tree is defined as a binary

/// tree in which the depth of the

/// two subtrees of every node never differ by more than 1.

/// </summary>

bool LeetCode::isBalanced(TreeNode\* root)

{

int depth;

return isBalanced(root, depth);

}

# Sort

When we talk about sort, it is default N\*Log(N) as the time complexity, but in some case we can make it as N complexity.

### **739. Daily Temperatures**

Given a list of daily temperatures, produce a list that, for each day in the input, tells you how many days you would have to wait until a warmer temperature. If there is no future day for which this is possible, put 0 instead.

For example, given the list temperatures = [73, 74, 75, 71, 69, 72, 76, 73], your output should be [1, 1, 4, 2, 1, 1, 0, 0].

**Note:** The length of temperatures will be in the range [1, 30000]. Each temperature will be an integer in the range [30, 100].

/// <summary>

/// Leet code #739. Daily Temperatures

///

/// Given a list of daily temperatures, produce a list that, for each day

/// in the input, tells you how many days you would have to wait until a

/// warmer temperature. If there is no future day for which this is

/// possible, put 0 instead.

/// For example, given the list temperatures = [73, 74, 75, 71, 69, 72,

/// 76, 73], your output should be [1, 1, 4, 2, 1, 1, 0, 0].

/// Note: The length of temperatures will be in the range [1, 30000]. Each

/// temperature will be an integer in the range [30, 100].

/// </summary>

vector<int> LeetCode::dailyTemperatures(vector<int>& temperatures)

{

vector<int> result(temperatures.size());

stack<pair<int, int>> temp\_stack;

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

{

while (!temp\_stack.empty())

{

auto temperature = temp\_stack.top();

// if new temperature is lower then simply push it.

if (temperature.first >= temperatures[i]) break;

// set the day and pop up

result[temperature.second] = i - temperature.second;

temp\_stack.pop();

}

temp\_stack.push(make\_pair(temperatures[i], i));

}

return result;

}

# BFS

Breadth First Search is that you start from a specific position, push to a queue, search the neighbor, find any target cell, if it is valid and not visited, we add to it the queue for next search iteration.

We loop the process until the search queue is empty.

One classical problem is flood fill, which we can paint a region with specific number to resolve a more complicated problem.

### **733. Flood Fill**

An image is represented by a 2-D array of integers, each integer representing the pixel value of the image (from 0 to 65535).

Given a coordinate (sr, sc) representing the starting pixel (row and column) of the flood fill, and a pixel value newColor, "flood fill" the image.

To perform a "flood fill", consider the starting pixel, plus any pixels connected 4-directionally to the starting pixel of the same color as the starting pixel, plus any pixels connected 4-directionally to those pixels (also with the same color as the starting pixel), and so on. Replace the color of all of the aforementioned pixels with the newColor.

At the end, return the modified image.

**Example 1:**

**Input:**

image = [[1,1,1],[1,1,0],[1,0,1]]

sr = 1, sc = 1, newColor = 2

**Output:** [[2,2,2],[2,2,0],[2,0,1]]

**Explanation:**

From the center of the image (with position (sr, sc) = (1, 1)), all pixels connected

by a path of the same color as the starting pixel are colored with the new color.

Note the bottom corner is not colored 2, because it is not 4-directionally connected

to the starting pixel.

**Note:**

 The length of image and image[0] will be in the range [1, 50].

 The given starting pixel will satisfy 0 <= sr < image.length and 0 <= sc < image[0].length.

 The value of each color in image[i][j] and newColor will be an integer in [0, 65535].

/// <summary>

/// Leet code #733. Flood Fill

///

/// An image is represented by a 2-D array of integers, each integer

/// representing the pixel value of the image (from 0 to 65535).

///

/// Given a coordinate (sr, sc) representing the starting pixel

/// (row and column) of the flood fill, and a pixel value newColor,

/// "flood fill" the image.

///

/// To perform a "flood fill", consider the starting pixel, plus any

/// pixels connected 4-directionally to the starting pixel of the same

/// color as the starting pixel, plus any pixels connected 4-directionally

/// to those pixels (also with the same color as the starting pixel), and

/// so on. Replace the color of all of the aforementioned pixels with the

/// newColor.

///

/// At the end, return the modified image.

///

/// Example 1:

/// Input:

/// image = [[1,1,1],[1,1,0],[1,0,1]]

/// sr = 1, sc = 1, newColor = 2

/// Output: [[2,2,2],[2,2,0],[2,0,1]]

/// Explanation:

/// From the center of the image (with position (sr, sc) = (1, 1)), all

/// pixels connected by a path of the same color as the starting pixel are

/// colored with the new color.

/// Note the bottom corner is not colored 2, because it is not

/// 4-directionally connected to the starting pixel.

/// Note:

///

/// The length of image and image[0] will be in the range [1, 50].

/// The given starting pixel will satisfy 0 <= sr < image.length and

/// 0 <= sc < image[0].length.

/// The value of each color in image[i][j] and newColor will be an

/// integer in [0, 65535].

/// </summary>

vector<vector<int>> LeetCode::floodFill(vector<vector<int>>& image, int sr, int sc, int newColor)

{

queue<pair<int, int>> search\_queue;

vector<vector<int>> result = image;

vector<vector<int>> visited(image.size(), vector<int>(image[0].size()));

vector<vector<int>> directions = { { -1, 0 },{ 1, 0 },{ 0, -1 },{ 0, 1 } };

int oldColor = image[sr][sc];

visited[sr][sc] = 1;

search\_queue.push(make\_pair(sr, sc));

while (!search\_queue.empty())

{

pair<int, int> pos = search\_queue.front();

search\_queue.pop();

result[pos.first][pos.second] = newColor;

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

{

pair<int, int> new\_pos = make\_pair(pos.first + directions[i][0], pos.second + directions[i][1]);

// if new position out of boundary skip it

if ((new\_pos.first < 0) || (new\_pos.second < 0) || (new\_pos.first >= (int)image.size()) ||

(new\_pos.second >= (int)image[0].size()))

{

continue;

}

// if color not match skip

if (result[new\_pos.first][new\_pos.second] != oldColor)

{

continue;

}

// check visited flag, if already set skip

if (visited[new\_pos.first][new\_pos.second] == 1)

{

continue;

}

visited[new\_pos.first][new\_pos.second] = 1;

search\_queue.push(new\_pos);

}

}

return result;

}

# Topology Sort

The topology sort is based on BFS algorithm. On initialization, we count degree for each node, which is how many nodes depending on it, and record how many nodes depending on it. Then we start a BFS process, which starts a node without dependency, push it to the queueue, visit it and clear all the dependencies on this node, if you get a new free node, add to the search queue. We continue this process until the queue is empty.

### **207. Course Schedule**

There are a total of *n* courses you have to take, labeled from 0 to n-1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]

Given the total number of courses and a list of prerequisite **pairs**, is it possible for you to finish all courses?

**Example 1:**

**Input:** 2, [[1,0]]

**Output:** true

**Explanation:** There are a total of 2 courses to take.

  To take course 1 you should have finished course 0. So it is possible.

**Example 2:**

**Input:** 2, [[1,0],[0,1]]

**Output:** false

**Explanation:** There are a total of 2 courses to take.

  To take course 1 you should have finished course 0, and to take course 0 you should

  also have finished course 1. So it is impossible.

**Note:**

1. The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about [how a graph is represented](https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs).
2. You may assume that there are no duplicate edges in the input prerequisites.

/// <summary>

/// LeetCode #207. Course Schedule

/// There are a total of n courses you have to take, labeled from 0 to

/// n - 1.

/// Some courses may have prerequisites, for example to take course 0 you

/// have to first take course 1, which is expressed as a pair: [0,1]

/// Given the total number of courses and a list of prerequisite pairs,

/// is it possible for you to finish all courses?

///

/// For example:

/// 2, [[1,0]]

/// There are a total of 2 courses to take. To take course 1 you should

/// have finished course 0. So it is possible.

///

/// 2, [[1,0],[0,1]]

/// There are a total of 2 courses to take. To take course 1 you should

/// have finished course 0,

/// and to take course 0 you should also have finished course 1. So it

/// is impossible.

/// Note:

/// The input prerequisites is a graph represented by a list of edges,

/// not adjacency matrices. Read more about how a graph is represented.

/// </summary>

bool LeetCode::canFinishCourse(int numCourses, vector<pair<int, int>>& prerequisites)

{

vector<int> degree(numCourses);

vector<unordered\_set<int>> dependency(numCourses);

queue<int> search;

// remember which course dependes on others and which ones depends on me

for (pair<int, int> pair : prerequisites)

{

if (dependency[pair.second].count(pair.first) == 0)

{

degree[pair.first]++;

dependency[pair.second].insert(pair.first);

}

}

// get all the course not depends on others, this is our starting search scope

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

{

if (degree[i] == 0) search.push(i);

}

// Using queue to manage BFS and get every free course and clear the

// dependency with a free course, i.e. you depend on a free course,

// then such dependency

// does not matter. If all dependencies are clear, we got a new

// free course

while (!search.empty())

{

int free\_course = search.front();

search.pop();

for (int next\_course : dependency[free\_course])

{

degree[next\_course]--;

if (degree[next\_course] == 0)

{

search.push(next\_course);

}

}

}

// if number of free courses equals to the total course, we can finish

// all courses

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

{

if (degree[i] > 0) return false;

}

return true;

}

# Linked List

Linked list problem is the simplest (not easiest) category in LeetCode. There are no special algorithms, except for the linked table clone. It is not easy because you can easily have tons of bugs if you write code randomly.

Almost all the linked-list problems in Leet Code are singly linked list (one direction). Singly linked list is more difficult to modify than doubly linked list which we prefer in our work.

Please remember the following principles which will make your life easy.

1. Iterate the complete list, until you see the next node is full.
2. Please remember the **previous** node if you want to delete, move or insert before the current node.
3. If you need to assign to a pointer, for example current->next, you need to first **save** the original value somewhere before.
4. To get the middle point of the linked, or get the last N position in the linked list, the common strategy is to use two pointers, one is a **fast traveler** which move ahead and another one is a slow traveler which follow the fast one and target for the position you try to search. When the fast traveler reach to the end (null), your **slow traveler** stop at the right position you want to reach.

Please look at the following code example:

**Example 1 build linked list**

It checks if the head is null first, if yes, make the new node as head, if no, append after previous node.

if (head == nullptr)

{

head = current;

prev = head;

}

else

{

prev->next = current;

prev = current;

}

**Example 2 Move a node one position towards tail**

Assume we have previous node already. Keep on doing it will reverse the whole linked list after previous node

ListNode \* next = current->next;

current->next = next->next;

next->next = prev->next;

previous->next = next;

**Example 3 Find the middle point of the linked list**

Please note that slow ends up pointing to the previous node to the middle point, if you want exact middle point, take out the if … break…

while (fast != nullptr)

{

fast = fast->next;

if (fast != nullptr) fast = fast->next;

if (fast == nullptr) break;

slow = slow->next;

}

### **19. Remove Nth Node From End of List**

Given a linked list, remove the n-th node from the end of list and return its head.

**Example:**

Given linked list: **1->2->3->4->5**, and **n = 2**.

After removing the second node from the end, the linked list becomes **1->2->3->5**.

**Note:**

Given n will always be valid.

**Follow up:**

Could you do this in one pass?

/// <summary>

/// Leet code #19. Remove Nth Node From End of List

/// Given a linked list, remove the nth node from the end of list and return its head.

/// For example,

/// Given linked list: 1->2->3->4->5, and n = 2.

/// After removing the second node from the end, the linked list becomes 1->2->3->5.

/// Note:

/// Given n will always be valid.

/// Try to do this in one pass.

/// </summary>

ListNode\* LeetCode::removeNthFromEnd(ListNode\* head, int n)

{

ListNode \* first = head;

ListNode \* last = head;

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

{

last = last->next;

}

if (last == nullptr)

{

head = first->next;

delete first;

}

else

{

while (last->next != nullptr)

{

first = first->next;

last = last->next;

}

ListNode \* next = first->next;

first->next = next->next;

delete next;

}

return head;

}

### **206. Reverse Linked List**

Reverse a singly linked list.

**Example:**

**Input:** 1->2->3->4->5->NULL

**Output:** 5->4->3->2->1->NULL

**Follow up:**

A linked list can be reversed either iteratively or recursively. Could you implement both?

/// <summary>

/// Leet code #206. Reverse Linked List

/// Reverse a singly linked list.

/// </summary>

ListNode\* LeetCode::reverseList(ListNode\* head)

{

ListNode \*first = head, \*last = head;

while (last != nullptr)

{

ListNode \*next = last->next;

// no more next element, complete

if (next == nullptr) break;

last->next = next->next;

next->next = first;

first = next;

}

return first;

}

# Backtracking

The definition of Backtracking is as below:

Your job is to search all possible solutions to the problem, you can define such job as a search, this search has several steps, on each step you have several options, you choose one of them, based on the option you choose, you will have another set of valid options as the next step, eventually you will either reach the destination or exhaust your valid options, when you exhaust your valid options, you will move one step back and take the next valid option from the last choice in the previous step and repeat the processes above.

Because during the backtracking, we always find where to go based on the last choice we made, so it is a deep first search, so another name of backtracking is DFS.

Given the definition above, you can define your algorithm as below:

1. Check whether the path already in cache, if so directly output the memorized result.
2. Check if we have already reached the destination? If so collect the result, you may have many results in a solution, also check if we already exhaust all choice, i.e. the current step is the end of road, if so return.
3. Get all the possible options on the current step, and start a loop to choose the first one, check if it is a valid choice, if yes, add to path, mark the option as visited, if no, next one.
4. Recursive call step 1 **with further step**, wait for return.
5. If return from the recursive call, it means you have already searched all the paths from current position, remove the current choice from the path, mark the option as unvisited, so the option is available for the next choice in the further path, remember the conclusion for the current path in cache for fast exit.
6. Choose the next option in the loop from step 3.
7. Loop exit, return all the result.

There are some key points here.

1. You can choose recursive or non-recursive for back tracking, but recursive is easier, for non-recursive, you need to implement a step which remember the choice in each step.
2. Pass your original data set for search, the step number, traveled path, visited map, cache map, result set in the recursive call. All the vectors, maps and sets should pass by reference, this is due to performance and carry out result.
3. If your original data set has duplication and you need to de-dup in the final solution, just skip the same values in any specific step. For example, if you try to find all the combination in data set with duplication, just make sure the value X does not appear in the ith position twice.
4. if the order in the result does not matter, i.e. combination, which means [3,4] and [4,3] are identical, you can sort the input data set first, later you iterate in an order and avoid duplication. When iteration is in order, you may not need cache.

**What to pass in the recursive function?**

You may need to pass the following information as the parameters to the recursive calls.

1. The original data set to process.
2. The travelled path so far, may be along with the step number.
3. The visited map, or anything we can fast check the next step is valid or not. Calculate again from path will impact performance.
4. The cache, which record our past travel experience, if we know the result if we walk in this way already, we do not need to travel the exhausted paths again.
5. The result set, because we need to record every single record.
6. The exit condition may be the total number of steps you want to travel.

### **39. Combination Sum**

Given a **set** of candidate numbers (candidates) **(without duplicates)** and a target number (target), find all unique combinations in candidates where the candidate numbers sums to target.

The **same** repeated number may be chosen from candidates unlimited number of times.

**Note:**

* All numbers (including target) will be positive integers.
* The solution set must not contain duplicate combinations.

**Example 1:**

**Input:** candidates = [2,3,6,7], target = 7,

**A solution set is:**

[

[7],

[2,2,3]

]

**Example 2:**

**Input:** candidates = [2,3,5], target = 8,

**A solution set is:**

[

  [2,2,2,2],

  [2,3,3],

  [3,5]

]

/// <summary>

/// Leet code #39. Combination Sum

/// </summary>

void LeetCode::combinationSum(vector<int>& candidates, int target, int index, vector<int>& path,

vector<vector<int>>&result)

{

if (target == 0)

{

if (!path.empty()) result.push\_back(path);

return;

}

for (size\_t i = index; i < candidates.size(); i++)

{

if (candidates[i] > target) break;

target -= candidates[i];

path.push\_back(candidates[i]);

combinationSum(candidates, target, i, path, result);

target += candidates[i];

path.pop\_back();

}

}

/// <summary>

/// Leet code #39. Combination Sum

/// Given a set of candidate numbers (C) and a target number (T), find all

/// unique combinations in C where the candidate numbers sums to T.

/// The same repeated number may be chosen from C unlimited number of times.

/// Note:

/// All numbers (including target) will be positive integers.

/// The solution set must not contain duplicate combinations.

/// For example, given candidate set [2, 3, 6, 7] and target 7,

/// A solution set is:

/// [

/// [7],

/// [2, 2, 3]

/// ]

/// </summary>

vector<vector<int>> LeetCode::combinationSum(vector<int>& candidates, int target)

{

vector<int> path;

vector<vector<int>> result;

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

combinationSum(candidates, target, 0, path, result);

return result;

}

# Design Problem

The key point in design problem is to decide what data structure you want to use. For example, is it sorted, access by key, add and remove in O(1)?

Please pay attention to the following data structures, hashtable, (binary sorted) tree map, list (add and remove by O(1).

### **146. LRU Cache**

Design and implement a data structure for [Least Recently Used (LRU) cache](https://en.wikipedia.org/wiki/Cache_replacement_policies#LRU). It should support the following operations: get and put.

get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.  
put(key, value) - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

**Follow up:**  
Could you do both operations in **O(1)** time complexity?

**Example:**

LRUCache cache = new LRUCache( 2 /\* capacity \*/ );

cache.put(1, 1);

cache.put(2, 2);

cache.get(1); // returns 1

cache.put(3, 3); // evicts key 2

cache.get(2); // returns -1 (not found)

cache.put(4, 4); // evicts key 1

cache.get(1); // returns -1 (not found)

cache.get(3); // returns 3

cache.get(4); // returns 4

/// <summary>

/// Leet code #146. LRU Cache

/// Design and implement a data structure for Least Recently Used(LRU) cache.

/// It should support the following operations : get and set.

/// get(key) - Get the value(will always be positive) of the key if the key

/// exists in the cache, otherwise return -1.

/// set(key, value) - Set or insert the value if the key is not already present.

/// When the cache reached its capacity, it should invalidate the least recently

/// used item before inserting a new item.

/// </summary>

class LRUCache

{

private:

size\_t m\_Capacity;

list<pair<int, int>> m\_List;

unordered\_map<int, list<pair<int, int>>::iterator> m\_map;

public:

/// <summary>

/// Constructor an empty LRU cache

/// </summary>

/// <param name="capacity">capacity</param>

/// <returns></returns>

LRUCache(int capacity)

{

m\_Capacity = capacity;

}

/// <summary>

/// Destructor of an LRUCache

/// </summary>

/// <returns></returns>

~LRUCache()

{

}

/// <summary>

/// Set the key value pair in the LRU cache.

/// </summary>

/// <param name="key">The key</param>

/// <param name="value">The value</param>

/// <returns></returns>

void set(int key, int value)

{

if (m\_map.find(key) == m\_map.end())

{

m\_List.push\_front(make\_pair(key, value));

if (m\_List.size() > m\_Capacity)

{

pair<int, int> pair = m\_List.back();

m\_map.erase(pair.first);

m\_List.pop\_back();

}

m\_map[key] = m\_List.begin();

}

else

{

m\_List.erase(m\_map[key]);

m\_List.push\_front(make\_pair(key, value));

m\_map[key] = m\_List.begin();

}

}

/// <summary>

/// Get the value(will always be positive) of the key if the key exists in the cache.

/// otherwise return -1.

/// </summary>

/// <returns>the value</returns>

int get(int key)

{

if (m\_map.find(key) == m\_map.end())

{

return -1;

}

list<pair<int, int>>::iterator iterator = m\_map[key];

pair<int, int> pair = \*iterator;

m\_List.erase(iterator);

m\_List.push\_front(pair);

m\_map[key] = m\_List.begin();

return pair.second;

}

/// <summary>

/// Remove a key in the LRU cache.

/// </summary>

/// <param name="key">The key</param>

/// <returns>true, if found</returns>

bool remove(int key)

{

if (m\_map.find(key) == m\_map.end())

{

return false;

}

else

{

std::list<pair<int, int>>::iterator iterator = m\_map[key];

m\_List.erase(iterator);

m\_map.erase(key);

return true;

}

}

};

# Interval

The interval problem normally can be resolved in two way, one is to sort and merge / split intervals, another is to calculate the overlap.

### **253. Meeting Rooms II**

Given an array of meeting time intervals consisting of start and end times [[s1,e1],[s2,e2],...] (si < ei), find the minimum number of conference rooms required.

**Example 1:**

**Input:** [[0, 30],[5, 10],[15, 20]]

**Output:** 2

**Example 2:**

**Input:** [[7,10],[2,4]]

**Output:** 1

/// <summary>

/// Leet code #253. Meeting Rooms II

///

/// Given an array of meeting time intervals consisting of start

/// and end times [[s1,e1],[s2,e2],...] (si < ei),

/// find the minimum number of conference rooms required.

/// For example,

/// Given [[0, 30],[5, 10],[15, 20]],

/// return 2.

/// </summary>

int LeetCode::minMeetingRoomsII(vector<Interval>& intervals)

{

map<int, int> time\_line;

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

{

time\_line[intervals[i].start]++;

time\_line[intervals[i].end]--;

}

int max\_rooms = 0, rooms = 0;

for (auto time : time\_line)

{

rooms += time.second;

max\_rooms = max(max\_rooms, rooms);

}

return max\_rooms;

}

# Appendix

## Choose a Language

To practice the algorithm, you should choose a programming language. Actually, any programming language is good if you are familiar with it. I personally recommend a object oriented language which packages common data structure objects very well, such as C++, Java and C#. Some high level integrated languages such as Python is also a good choice. However I do not recommend C, Basic, Java scripts, Ruby and other raw or incomplete language to do the algorithm, the reason is you have to either implement some basic stuff yourself, or you are not able to see the in-depth algorithm implementation in the language.

I choose C++ because this is the one I am familiar with.

## STD library

The standard library in C++ implement all the basic data structure, which provides significant convinence for you to write the algorithm in C++, so it is a must to be studied and remembered.

Basically, you should have the following include and name space defined in your header file.

#include <functional>

#include <algorithm>

#include <unordered\_map>

#include <unordered\_set>

#include <map>

#include <stack>

#include <set>

#include <queue>

#include <vector>

using namespace std;

In the following table I will explain them in detail.

|  |  |  |
| --- | --- | --- |
| Data Structure | Descriptions | Code example |
| vector<T> | An array which can hold any object type.  It can auto grow, you can refer to a item in the vector by index in O(1).  Because it is push\_back() and pop\_back(), you can also use vector as a stack. | vector<int> array;  array.push\_back(2);  array[2] = 3;  array.back();  array.size();  array.pop\_back();  if (array.empty()) {};  array.clear(); |
| list<T> | single list, you can push to both ends, travel by single direction iterator, forward or backward, you can also go to previous item. | list<int> l;  l.push\_front(25);  l.push\_back(13);  l.pop\_back();  l.pop\_front();  int v = l.front();  int v = l.back();  auto itr = l.begin();  l.erase(itr); |
| queue<T> | A standard queue, which allow you to push to the back and pop from front. FIFO. The items can be any object type as long as constructors are complete (think big 3).  Please make sure to check if the queue is empty before peeking the item (front()). | queue<int> myQueue;  myQueue.push(2);  int v = myQueue.front();  myQueue.pop();  myQueue.empty(); |
| stack<T> | A standard stack, which allow you to push to back and pop up from back, LIFO. It can hold any object.  Please make sure to check if the stack is empty before peeking the item (top()). | stack<int> myStack;  myStack.push(2);  myStack.top();  myStack.pop();  myStack.empty(); |
| unordered\_map<T1, T2> | A hashtable, which hold the key, value pair. The key must implement a default hash function. You can assume accessing by key is O(1), but do not expect the keys are in order. The value can be a simple type such as integer or string, it also can be a container such as a vector, set or another map.  To test if a key is in the hashtable you can use find() (return iterator which is end if not found) or count(key)(return 0 if not found);  Please notice access the map by a key which does not exist, can cause a new key insert with a default value. | unordered\_map<int, int> myMap;  myMap[3] = 2;  if (myMap.count(2) == 0) {};  myMap.erase(2);  myMap.empty();  myMap.find(2);  // the following will have a 0 returned if the key of 3 does not exist.  int v = myMap[3]; |
| unordered\_set<T> | A hashtable based set, duplication are removed.  Access the key is O(1). | unordered\_set<int> mySet;  mySet.insert(3);  if (mySet.count(2) == 0) {};  mySet.erase(2);  mySet.empty(); |
| map<T1, T2> | It is implemented as binary search tree, in most cases, it is self-balanced tree such as red-black tree.  The object type used as key must have a default comparator implemented.  You can assume that you can access any key by O(log(n)).  The other rules are same as unordered\_map.  Please notice for pair<int, int> no hash function is implemented, so it is not supported in unordered\_map, but you can have it in map because the comparator is implemented. | map<int, int> myMap;  myMap[3] = 2;  if (myMap.count(2) == 0) {};  myMap.erase(2);  myMap.empty();  myMap.find(2);  // the following will have a 0 returned if the key of 3 does not exist.  int v = myMap[3]; |
| set<T> | A binary search tree based set, duplication are removed.  Access the key is O(log(n)) | set<int> mySet;  mySet.insert(3);  if (mySet.count(2) == 0) {};  mySet.erase(2);  mySet.empty(); |
| deque<T> | A double direction queue which you can push or pop from both ends, front and back.  In C++, the deque has far more comprehensive methods such as insert, erase which access the item by any position, but for algorithm please ignore them first. | deque<int> myQueue;  myQueue.puh\_back(2);  myQueue.puh\_front(3);  myQueue.pop\_back();  myQueue.pop\_front();  int value = myQueue.back();  int value = myQueue.front();  myQueue.empty(); |
| priority\_queue<T> | A sorted queue which always has the maximum value on the top. You can push any value to the right position, and you can peek or pop the maximum value from queue.  Push to or Pop from the queue is O(log(n)), while peek the top value is O(1).  A priority\_queue is not de-duplicated, so it allows multiple items with same value in the queue.  A priority\_queue can have its customized comparator, so it can defined its own "highest priority value". However if you just want a smallest value at top you can either make the value as negative or have a default comparator such as great()). | priority\_queue<int> myQueue;  myQueue.push(2);  myQueue.pop();  myQueue.top();  myQueue.empty();  myQueue.size();  priority\_queue<int, vector<int>, greater<int>> m\_Large; |
| multiset<T1> | A binary search tree based set which allows duplicated value, please notice that erase a value will erase all the items with the specific value, so you must erase at iterator. | multiset<int> low\_half;  low\_half.insert(value);  low\_half.erase(low\_half.find(\*low\_half.rbegin())); |
| pair<T1, T2> | This is a key-value pair which is equivalent to a vector<T>(2); We normally use it to represent a interval, a position in a 2-D plane or a key value pair. | pair<int, int> interval;  interval.first = 3;  interval.end = 5;  int distance = interval.second – interval.first; |

### Other Language

The STL in C++ have their cooresponding type in other language.

|  |  |  |
| --- | --- | --- |
| C++ | Java | Python |
| vector<T>  vector.push\_back(T);  vector.pop\_back();  vector.size();  vector.clear(); | ArrayList<T> | list = [“a”, “b”, 1, 2]  list[2]=1  del list[2]  list.append(T)  len(list)  list = [] |
| queue<T>  queue.empty();  queue.push(T);  queue.pop();  queue.front();  queue.size(); | Queue<T>  queue.empty();  queue.Add(T);  queue.Remove();  queue.element() | len(list)  list.pop()  list.insert(0,T)  list[0] |
| stack<T>  stack.empty();  stack.push(T);  stack.pop();  stack.top(); | Stack<T>  stack.empty();  stack.push(T);  stack.pop();  stack.peek(); | stack=[]  len(stack)  stack.append(T)  stack.pop()  stack[-1] |
| unordered\_map<T1, T2>  map.empty();  map.count(key) == 0;  map[key] =value;  value = map[key];  map.erase(key);  map.size(); | HashMap<T1, T2>  map.isEmpty();  map.containsKey(key);  map.put(key, value);  map.get(key);  map.remove(key);  map.size(); | hash = {'a': 1, 'c': 3}  hash[‘a’] =2  if 'key1' in dict:  print "blah"  else:  print "boo"  del myDict['key'] |
| unordered\_set<T>  set.empty();  set.count(T) == 0;  set.insert(T);  set.erase(T);  set.size(); | HashSet<T>  set.isEmpty();  set.contains(T);  set.add(T);  set.remove(T);  set.size(); |  |
| map<T1, T2>  map.empty();  map.count(T) == 0;  map[key] = value;  value = map[key];  map.erase(key);  map.size(); | TreeMap<T1, T2>  map.isEmpty(); map.containsKey(key);  map.put(key) = value;  value = map.get(key);  map.remove(key);  map.size(); | OrderedDict |
| set<T>  set.empty();  set.count(T) == 0;  set.insert(T);  set.erase(T);  set.size(); | TreeSet<T>  set.isEmpty();  set.contains(T);  set.add(T);  set.remove(T);  set.size(); |  |
| deque<T>  dequeue.empty();  deque.push\_back(T);  dequeue.push\_front(T);  dequeue.pop\_back();  dequeue.pop\_front();  dequeue.back();  dequeue.front(); | Deque<T>  dequeue.isEmpty();  dequeue.addLast(T);  dequeue.addFirst(T);  dequeue.removeLast();  dequeue.removeFirst();  dequeue.peekLast();  dequeue.peekFirst(); | deque |
| priority\_queue<T>  pq.empty();  pq.size();  pq.push<T>;  pq.pop();  value = pq.top(); | PriorityQueue<T>  pq.add(T);  pq.size();  pq.add(T);  pq.poll();  value = pq.peek(); |  |
| multiset<T> | N/A, can use PriorityQueue |  |
| pair<T1, T2>  pair.first;  pair.second; | Pair<T1, T2>  pair.getKey();  pair.getValue(); |  |

## Iterator

In STL, every data structrure mentioned is considered as a container which allow you to hold multiple items, so it is very common that we may need to traverse all the items one by one. In this case we will use a new concept called iterator.

The iterator is considered as a pointer to an item in the container. In STL, any iterator starts from begin() and ends by end(). Please notice that end() does not point to a valid item, it point to the end boundary which is the next to the last item.

An example of traverse is as below:

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

unordered\_map<char, int>::iterator itr;

int sum = 0;

for (itr = char\_map.begin(); itr != char\_map.end(); ++itr)

{

if (itr->second >= 2)

{

sum += (itr->second / 2) \* 2;

}

}

The above example is a map, so iterator->first points to the key and iterator->second points to the value.

If this is a simple vector you can use \*itr to access the value, please see the following example:

set<pair<int, int>> range\_set;

set<pair<int, int>>::iterator itr = range\_set.begin();

pair<int, int> curr = \*itr;

When using iterator, you need to watch a couple of things.

### Iterate and Erase

Every single container in the STL will have an iterator, however they are very different. For the interator of list it is constant, which is to say after you get an iterator for the item you can keep it even after that the container is update, for example add or delete an item in the list. However such assumption may be invalid in other container, for example you can not assume an iterator for a tree map or hashtable item is still valid after the container is updated. (Although in C++ most of such assumption are still true).

<http://www.martinbroadhurst.com/iterator-invalidation-rules-for-c-containers.html>

This is also to say after you get the iterator and try to traverse every item in he container, the container can not be modified by adding or removing any item, otherwise iterator may become invalid.

But how can we iterate and remove the item in the container? The answer is that in C++, you can remove the previous item just ahead of the iterator. For example, if you want to erase the item at the position for an iterator, you need to move the iterator to next first, then delete the current one. The following is an example:

while (start != pos\_map.end() && start->first < positions[i].first + positions[i].second)

{

auto temp = start++;

height = max(height, temp->second);

if (temp->first >= positions[i].first)

{

pos\_map.erase(temp);

}

}

#### Java

In Java you can also use iterator.

List<String> names = ....

Iterator<String> i = names.iterator();

while (i.hasNext()) {

String s = i.next(); // must be called before you can call i.remove()

// Do something

i.remove();

}

Please notice that the following code is illegal

List<String> names = ....

for (String name : names) {

// Do something

names.remove(name).

}

### Iterate with distance (DO NOT USE)

Some container has the implementation to calculate the distance between two iterators, but you should know there is no free lunch, a iterator++ or iterator—can be considered as O(1) operation, since it just to get the next or previous item, but iterator + n is not that easy, unless you know the container is a vector, please avoid it. A iterator +n may lead to iterating the items in the container one by one end up with O(n) instead of O(1) as expected.

The same is true for int distance = iterator1 – iterator2;

## Sort()

The function is defined in <algorithm>

The sorting is suitable for a container like vector, for other containers, it makes less sense.

The sort() is You can have a simple sort like below:

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

In some case you may want your own comparator. The following is an example.

struct PointCompare

{

bool operator() (Point &a, Point &b)

{

return (a.x == b.x) ? (a.y < b.y) : (a.x < b.x);

}

};

sort(points.begin(), points.end(), PointCompare());

## Reverse()

The function is defined in <algorithm>

The sorting is suitable for a container like vector, for other containers, it makes less sense.

The following is an example:

reverse(reverse\_str.begin(), reverse\_str.end());

## Lower\_Bound()

The lower\_bound() is a native method for any BST based data structure container. It is to find a key which is equal or greater than the specified value. The time complexity is O(log(n)). This is a very useful function which is to find the right position in the container for an input value.

The following is an example:

map<int, int> time\_map;

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

{

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

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

{

if (itr != time\_map.begin())

{

itr--;

}

}

return itr;

};

The non-BST data structure such as vector can also have a lower\_bound() if the value in the data structure is sorted, but in this case we will use the lower\_bound defined in <algorithm>.

vector<int>::iterator last = lower\_bound(arr.begin(), arr.end(), x);

In both cases the function of lower\_bound returns an iterator, if no key is equal or greater than the specified value, a container::end() is returned.

## Flow control

When we talk about flow control, we are talking about for iteration and while loop. There are some common practices which can make it easy.

## For Iteration

1. A basic for iteration will look like:

for (size\_t j = 0; j < word\_len; j++)

{

checksum\_words += (unsigned int)words[i][j];

}

Please notice that we use size\_t as the variable because most of the size() function returns unsigned integer.

1. In C++ 11, we have something similar to foreach in C# and Java:

for (int i : intersection)

{

result.push\_back(i);

}

The varialble after ":" should be a collection type.

1. In some case we use auto variable to loop in a map:

for (auto itr : num\_map)

{

if (k == 0)

{

if (itr.second > 1) count++;

}

else

{

if (num\_map.count(itr.first + k) > 0) count++;

}

}

Please notice that itr is not an iterator, it is a key value pair in the map, so "->" is replaced by ".".

## While Loop

In most of the cases we can use while loop to do the recurring processing. A while loop can have an condition, for example this is a typical logic in binary search.

int first = 1;

int last = n;

int middle = first;

while (first < last)

{

middle = first + (last - first) / 2;

if (middle < bad\_version)

{

first = middle + 1;

middle++;

}

else

{

last = middle;

}

}

return middle;

Another example is in BFS we loop until queue is empty.

However a while with always true condition is also a common pattern if you do not have a clear exit condition. In this case we can use "break" to break out and "continue" to skip some process logic, both are good for flow control.

Another benefit for such logic is that we can do some finishing operation before we exit.

while (true)

{

if (node != nullptr)

{

search\_stack.push(node);

node = node->left;

continue;

}

if (search\_stack.empty())

{

break;

}

node = search\_stack.top();

value\_list.push\_back(node->val);

search\_stack.pop();

node = node->right;

}

## Numeric Calculation

### Overflow

In some problems, we need to watch integer overflow. Here we assume that integer is 32 bits, if it is signed the range is between INT\_MIN and INT\_MAX. The operation of aggregation, multiply may cause an integer temporarily over flow. In most cases, if we use an 64 bits long to represent the result, it should be good. Please remember in Windows version of C++, long is also 32 bits, we should say long long or unsigned long long to get 64 bits.

The following is an example:

long long long\_dividend = abs((long long)dividend);

long long long\_divisor = abs((long long) divisor);

### Compare float

If you want to compare two float values, you should avoid using "==", this is because after some calculate, there may be 1 least significant bit differs but actual value are same within reasonable precision range.

Here is the example we used in 24 Game, we know any calculation combination on 4 less than 10 positive integers, with "+", "-", "\*" and "/" will not have a gap less than 1/10/10/10.

for (double value : dp[0][nums.size() - 1])

{

if (abs(value - (double)24.0) < 0.001) return true;

}

## Function

### Separate function for independent operation

In many complex processing, it is encouraged to have a separate function to process some independent operation.

For example, in the Atom formular expression evaluation, to process something like "K4(ON(SO3)2)2", we have the following functions, each do a step of the evaluation.

string parseAtom(string &formula, int& index);

int parseAtomCount(string &formula, int& index);

void multiplyFormula(map<string, int> &formula, int& count);

void mergeFormula(map<string, int> &atom\_count, vector<map<string, int>>& formula\_array);

This will make each function isolate, easy to implement and the whole logic become simple and bug free.

## Pass by value and pass by reference

In the function parameters, we can pass the parameters either by value or by reference. Generally, we should do it by the following motivations.

### Pass by value

* The variable is in basic type and we do not need to return the modification.
* The variable should be kept as local in each recursive call.

### Pass by reference

* The variable is large and immutable, we want to avoid unnecessary copy constructor call.
* The variable is modified in the function and we want the updated result to be returned.

The following is a typical recursive function call in backtracking.

void combinationSum(vector<int>& candidates, int target, int index, vector<int>& path, vector<vector<int>>&result);

The candidates and path are considered as big data, and result should be returned.

The following function is to help to construct a tree, we pass reference to right child node.

void connectRight(TreeLinkNode \*&head, TreeLinkNode \*&ptr, TreeLinkNode \* node);

The following is a function call in LISP command processing

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

The variables is a big data structure, but we pass by value, it is because we want to keep the variables with local scope.

Exercise

1. Construct a map<int, int>, and insert 1 to 10 into the map each with a value of 1, and do the following practice.
   1. check if 9, 11 is in the map.
   2. double the value for all the even key.
   3. get the largest and smallest key from the map.
   4. delete all the keys if it is multiple 3, such as 3, 6 and 9.