Leet Code Version II

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

# 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.

# 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**

[Description](https://leetcode.com/problems/sentence-similarity-ii/description/)[Hints](https://leetcode.com/problems/sentence-similarity-ii/hints/)[Submissions](https://leetcode.com/problems/sentence-similarity-ii/submissions/)[Discuss](https://leetcode.com/problems/sentence-similarity-ii/discuss/)[Solution](https://leetcode.com/problems/sentence-similarity-ii/solution/)

[Discuss](https://discuss.leetcode.com/category/1626)[Pick One](https://leetcode.com/problems/random-one-question/)

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.

/// <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:

/// <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;

}

};