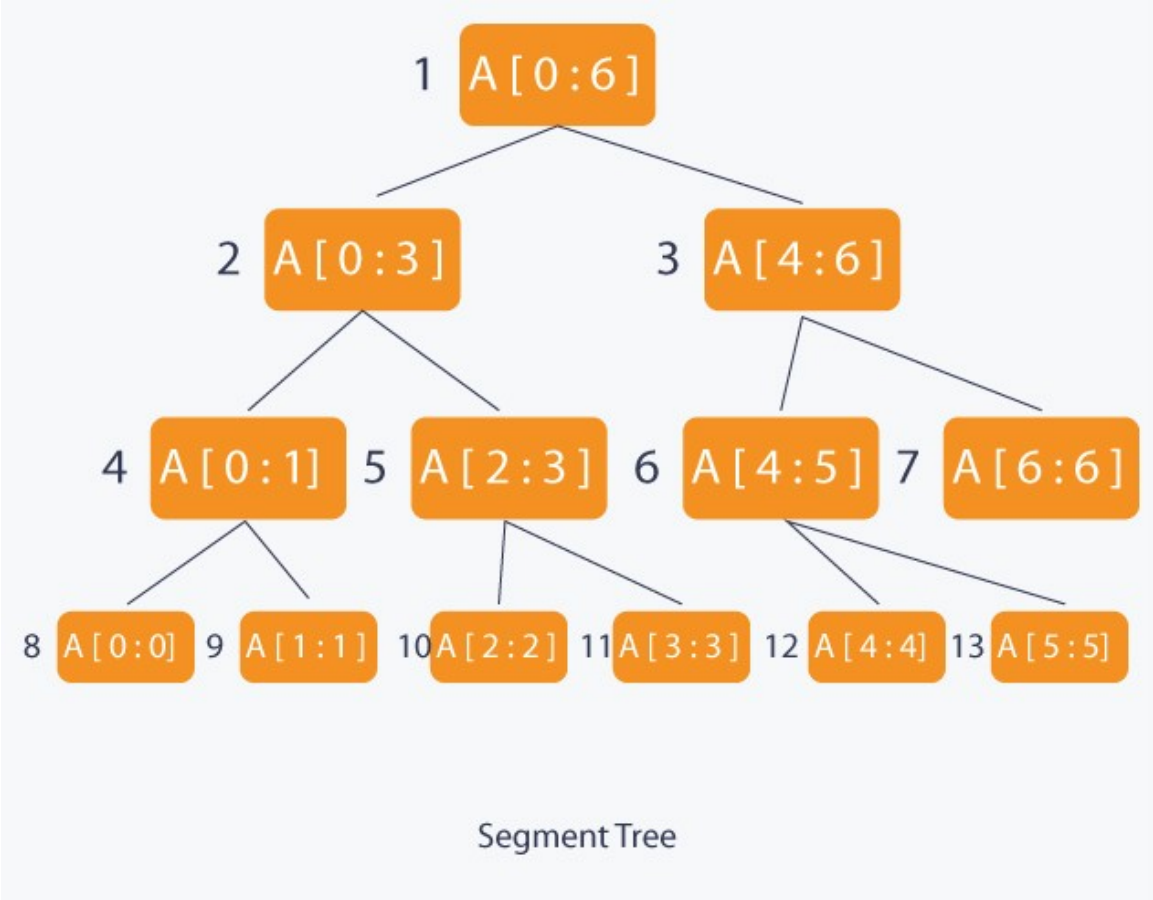
Segment Tree

Segment tree is used when we have an array of elements, we have two operations, **query** and **update**, The query operation should apply a range in the array, the range is given a starting point and ending point. The update should apply for either a single point or a range.

The segment tree is presented as below.



Segment Tree

In the segment tree, the individual elements are always at leaf, the parent nodes (non-leaves) normally indicate a range of the array. The root means the whole range of the array.

How it works:

The segment tree can reduce the query time complexity from O(N) to O(logN) in the sense that it remember result of a range in the non-leaf node, for example if we want to query the minimum value between A[0:6], we can store the minimum value for every range in the intermediate nodes, in this case we do not necessary need to access either leave node to get the answer. For example if we want to know the minimum for A[1:6], we can do by A[1:1] and A[4:6]. The query will go from root downward, if any node is

completely within the query range then we do not need to move down further, otherwise we goes down until the above condition is satisfied.

Scope:

The segment tree works in a way that:

1. The range of the array is fixed, which means you can not grow or shrink the array during the processing.
2. The query pattern is specified, which means you should either query for the minimum of the range or the sum of range, but you can not do both.

Implementation:

There are two implementations of segment tree:

The first implementation is a general binary tree is one of them, besides the left, right child, you can also add range, the value (the query result for that range) as the new attributes in the node.

The second implementation is using a array, where A[0] indicate the root and any A[i], it if is not a simple node will have two children, A[2*i] and A[2*i+1], which indicate the left and right half of the range indicated by A[i].

Binary Index Tree

Binary index tree can also be used to query the sum of range for an array which is being frequently update.

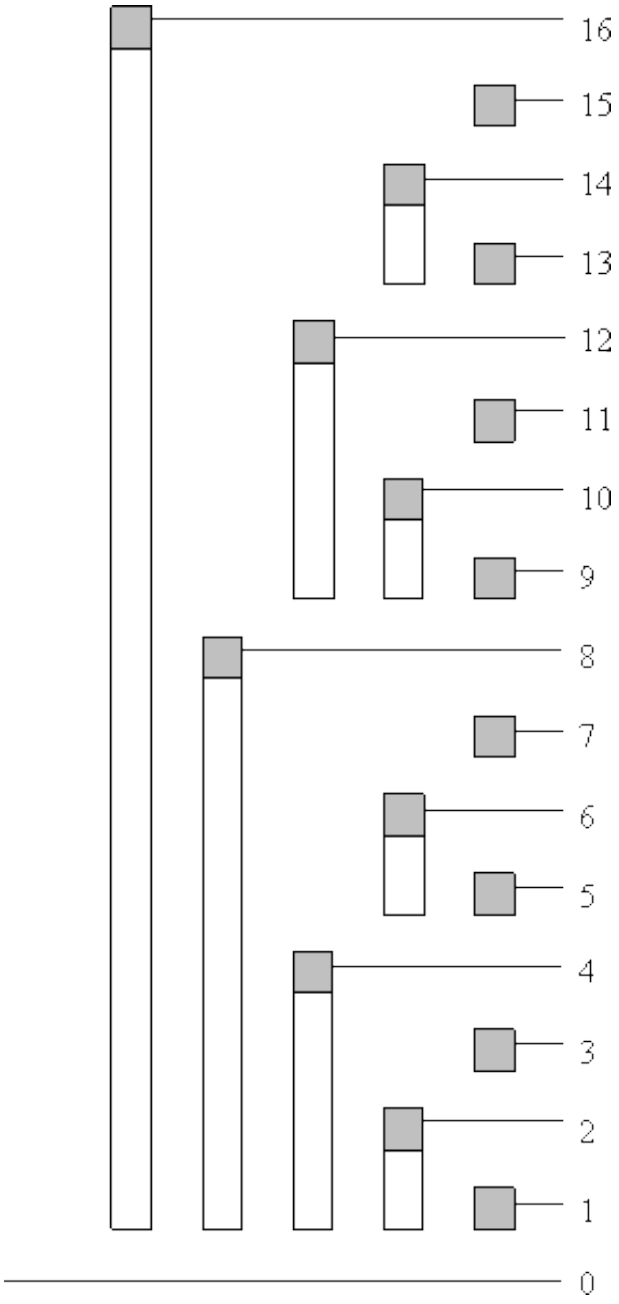
The following blog should cover the original idea of binary tree.

https://www.topcoder.com/community/data-science/data-science-tutorials/binary-indexed-trees/

How it works:

The binary index tree must be an array, and the size is same as the original array, for the value in ant A[i], it must also be added to its parent related node. The parent node is defined in the way that the index of parent node can be calculated by adding the lowest significant bit (1) to the index of child node.

Please look at the following graph.



1. Binary index tree is represented in an array with index as integer.
2. Each node in binary index tree stores a sum of a list of original records. For each original record X[i], the value will be added to all the elements in the binary index tree, from the index itself, and by progressively adding the lowest significant bit (means it is 1). For example for record R[10], the bits for 10 is 1010, first it should be added to the binary index tree element of T[10], then by adding the 0010, which become 1100, the R[10] should be added to T[12], then

T[16] (the next bit index is 10000), until the index reaches the size of array.
3. If you want to get the sum from 1 to N in the binary index tree, you need to deduct the lowest significant bit recursively until it reaches 0. For example if you want to get sum of (T[1] ... T[12]), you should add T12 and T8.
4. If you want to get back record R[N], you can do T[N]-T[N-1].
5. To get the lowest significant bit, the easy way is using x & -x;

Given the above theory, you only need two function for Binary index tree.

```
vector<int> m_sum;
int sum(int i)
{
    int sum = 0;
    int index = i + 1;
    while (index != 0)
    {
        sum += m_sum[index];
        index -= index & -index;
    }
    return sum;
}

void add(int i, int value)
{
    int index = i + 1;
    while (index < (int)m_sum.size())
    {
        m_sum[index] += value;
        index += (index & -index);
    }
}
```

Scope:

The scope of binary index tree is similar to the segment tree, but it has more constraint, it can be used in the query of sum or count, but not such as minimum or maximum.

The binary index tree uses less memory than segment tree.

The binary index tree can be used in two scenarios, one is to calculate a range of sum in a frequently updated array. The is Leet Code 307 and 308. Another is to calculate the relationship between the values from a dynamic array. This is Leet Code 315, in the second case you should transfer the value to the index, remember the index starts from 1 and the size of array is the distance between maximum value and minimum value.

Calculate range from a frequently updated array

```
/// <summary>
/// Leet code #307. Range Sum Query - Mutable
```

```
/// Given an integer array nums, find the sum of the elements between
indices
/// i and j (i ≤ j), inclusive.
/// The update(i, val) function modifies nums by updating the element at
index
/// i to val.
/// Example:
/// Given nums = [1, 3, 5]
/// sumRange(0, 2) -> 9
/// update(1, 2)
/// sumRange(0, 2) -> 8
/// Note:
/// 1.The array is only modifiable by the update function.
/// 2.You may assume the number of calls to update and sumRange function is
/// distributed evenly.
/// </summary>
class NumArrayMutable
{
private:
    vector<int> m_sum;
    int sum(int i)
    {
        int sum = 0;
        int index = i + 1;
        while (index != 0)
        {
            sum += m_sum[index];
            index -= index & -index;
        }
        return sum;
    }

    void add(int i, int value)
    {
        int index = i + 1;
        while (index < (int)m_sum.size())
        {
            m_sum[index] += value;
            index += (index & -index);
        }
    }

public:
    NumArrayMutable(vector<int> &nums)
    {
        m_sum = vector<int>(nums.size() + 1, 0);
        for (size_t i = 0; i < nums.size(); i++)
        {
            add(i, nums[i]);
        }
    }

    void update(int i, int value)
    {
        int old_value = sum(i) - sum(i - 1);
        value = value - old_value;
        add(i, value);
    }
```

```
    int sumRange(int i, int j)
    {
        return sum(j) - sum(i - 1);
    }
};
/// <summary>
/// Leet code #308. Range Sum Query 2D - Mutable
///
/// Given a 2D matrix matrix, find the sum of the elements inside the
rectangle
/// defined by its upper left corner (row1, col1) and lower right corner
(row2, col2).
///
/// Range Sum Query 2D
/// The above rectangle (with the red border) is defined by (row1, col1) =
(2, 1) and
/// (row2, col2) = (4, 3), which contains sum = 8.
/// Example:
/// Given matrix =
/// [
///   [3, 0, 1, 4, 2],
///   [5, 6, 3, 2, 1],
///   [1, 2, 0, 1, 5],
///   [4, 1, 0, 1, 7],
///   [1, 0, 3, 0, 5]
/// ]
/// sumRegion(2, 1, 4, 3) -> 8
/// update(3, 2, 2)
/// sumRegion(2, 1, 4, 3) -> 10
/// Note:
/// 1.The matrix is only modifiable by the update function.
/// 2.You may assume the number of calls to update and sumRegion function
is distributed evenly.
/// 3.You may assume that row1 ≤ row2 and col1 ≤ col2.
/// </summary>
class NumMatrixMutable
{
private:
    vector<vector<int>> m_matrix;
    int sum(int i, int j)
    {
        int sum = 0;
        int row = i + 1;
        while (row != 0)
        {
            int col = j + 1;
            while (col != 0)
            {
                sum += m_matrix[row][col];
                col -= (col & -col);
            }
            row -= (row & -row);
        }
        return sum;
    }

    void add(int i, int j, int value)
```

```
    {
        int row = i + 1;
        while (row < (int)m_matrix.size())
        {
            int col = j + 1;
            while (col < (int)m_matrix[row].size())
            {
                m_matrix[row][col] += value;
                col += (col & -col);
            }
            row += (row & -row);
        }
    }

public:
    NumMatrixMutable(vector<vector<int>> matrix)
    {
        if (matrix.empty() || matrix[0].empty()) return;
        m_matrix = vector<vector<int>>(matrix.size() + 1,
vector<int>(matrix[0].size() + 1));
        for (size_t i = 0; i < matrix.size(); i++)
        {
            for (size_t j = 0; j < matrix[i].size(); j++)
            {
                add(i, j, matrix[i][j]);
            }
        }
    }

    void update(int row, int col, int val)
    {
        int old_val = sumRegion(row, col, row, col);
        val = val - old_val;
        add(row, col, val);
    }

    int sumRegion(int row1, int col1, int row2, int col2)
    {
        if (row1 < 0 || col1 < 0 || row2 >= (int)m_matrix.size() || col2 >=
(int)m_matrix[0].size())
        {
            return 0;
        }
        int value = sum(row2, col2);
        if (row1 > 0) value -= sum(row1 - 1, col2);
        if (col1 > 0) value -= sum(row2, col1 - 1);
        if ((row1 > 0) && (col1 > 0))
        {
            value += sum(row1 - 1, col1 - 1);
        }
        return value;
    }
};
```

The fourth solution is based on binary index tree, assume each number is an integer, then when you have a new number you may update as many as 32 slot. The issue for this solution is that you need a large memory (4GB+) to store the binary index tree, or you can use a map to replace the vector which may further impact the performance. This solution is suit for the scenario when you know the range of the numbers, especially a small range, and a lot of duplicates.

But remember theoretically this is a O(nlog(n)) solution. (even someone can argue it is a O(n) if you consider multiple 32 is a constant)

```
/// <summary>
/// Leet code #315. Count of Smaller Numbers After Self
/// </summary>
void LeetCode::addBIT(int index, vector<int>& accu_slot)
{
    while (index < (int)accu_slot.size())
    {
        accu_slot[index] += 1;
        index += (index & -index);
    }
}

/// <summary>
/// Leet code #315. Count of Smaller Numbers After Self
/// </summary>
int LeetCode::sumBIT(int index, vector<int>& accu_slot)
{
    int sum = 0;
    while (index != 0)
    {
        sum += accu_slot[index];
        index -= index & -index;
    }
    return sum;
}

/// <summary>
/// Leet code #315. Count of Smaller Numbers After Self
///
/// You are given an integer array nums and you have to return a new counts
array.
/// The counts array has the property where counts[i] is the number of
smaller
/// elements to the right of nums[i].
/// Example:
/// Given nums = [5, 2, 6, 1]
/// To the right of 5 there are 2 smaller elements (2 and 1).
/// To the right of 2 there is only 1 smaller element (1).
/// To the right of 6 there is 1 smaller element (1).
/// To the right of 1 there is 0 smaller element.
/// Return the array [2, 1, 1, 0].
/// </summary>
vector<int> LeetCode::countSmallerIV(vector<int>& nums)
{
    vector<int> result(nums.size());
    int max_num = INT_MIN;
```

```
    int min_num = INT_MAX;
    for (size_t i = 0; i < nums.size(); i++)
    {
        max_num = max(max_num, nums[i]);
        min_num = min(min_num, nums[i]);
    }
    // the min_num should be located at 1 and
    // the max_num should be located at max_num-min_num + 1
    vector<int> accu_slot(max_num - min_num + 2);

    for (int i = nums.size() - 1; i >= 0; i--)
    {
        addBIT(nums[i] - min_num + 1, accu_slot);
        result[i] = sumBIT(nums[i] - min_num, accu_slot);
    }
    return result;
}
/// <summary>
/// Leet code #327. Count of Range Sum
/// </summary>
int LeetCode::mergeSort(vector<long long>& sums, int begin, int end, int
lower, int upper)
{
    if (begin + 1 >= end) return 0;
    int middle = begin + (end - begin) / 2;
    int count = 0;
    count = mergeSort(sums, begin, middle, lower, upper) +
            mergeSort(sums, middle, end, lower, upper);
    int m = middle, n = middle;
    for (int i = begin; i < middle; i++)
    {
        while ((m < end) && (sums[m] - sums[i] < lower)) m++;
        while ((n < end) && (sums[n] - sums[i] <= upper)) n++;
        count += n - m;
    }
    inplace_merge(sums.begin() + begin, sums.begin() + middle, sums.begin()
+ end);
    return count;
}
```

Design Problem

There is a group of problems marked as design problem, basically you are asked to implement some data structure which can efficiently fulfill the required operation. There is no very tricky algorithm for such problems, but you should design the best fit data structure for your algorithm.

To resolve such problems, you should be very familiar with the language built-in data structure, for example in C++ it is STL, for Java and C#, it should be Collections. The most common data structures are HashMap (Set), TreeMap (Tree), List, then PriorityQueue, MultiSet, Queue, Stack and Vector (ArrayList). In C++ you can also use pair, in Java you can use ArrayList.

The above data structures are all known as container, to iterate them you should use iterator, being familiar with the iterator for these containers is also important.

Sometimes in the LRU or LFU question, you are asked to store and fetch a top item in O(1) complexity, so you are not allowed to use TreeMap in this case, but you can use List as alternative, and remember the iterator in the list in the hashtable and move (which means erase and insert to front) the list item to top when it is touched.

```
/// <summary>
/// Leet code #716. Max Stack
///
/// Design a max stack that supports push, pop, top, peekMax and popMax.
///
/// push(x) -- Push element x onto stack.
/// pop() -- Remove the element on top of the stack and return it.
/// top() -- Get the element on the top.
/// peekMax() -- Retrieve the maximum element in the stack.
/// popMax() -- Retrieve the maximum element in the stack, and remove it.
/// If you find more than one maximum elements, only remove the top-most
/// one.
/// Example 1:
/// MaxStack stack = new MaxStack();
/// stack.push(5);
/// stack.push(1);
/// stack.push(5);
/// stack.top(); -> 5
/// stack.popMax(); -> 5
/// stack.top(); -> 1
/// stack.peekMax(); -> 5
/// stack.pop(); -> 1
/// stack.top(); -> 5
/// Note:
/// -1e7 <= x <= 1e7
/// Number of operations won't exceed 10000.
/// The last four operations won't be called when stack is empty.
/// </summary>
class MaxStack {
private:
```

```
	map<int, stack<int>> m_ValueMap;
	map<int, int> m_StackMap;
	int m_Index;
public:
	/// <summary>
	/// Constructor an empty max stack
	/// </summary>
	MaxStack()
	{
		m_Index = 0;
	}

	void push(int x)
	{
		// Add to value map, biggest first
		m_ValueMap[-x].push(m_Index);
		// Add to stack map, last one first
		m_StackMap[-m_Index] = x;
		m_Index++;
	}

	int pop()
	{
		// take first item from m_StackMap, which is the last item
based on index
		auto itr = m_StackMap.begin();
		int index = -itr->first;
		int value = itr->second;
		m_StackMap.erase(-index);
		// take out the top index from specific value
		m_ValueMap[-value].pop();
		if (m_ValueMap[-value].empty()) m_ValueMap.erase(-value);
		return value;
	}

	int top()
	{
		// take first item from m_StackMap, which is the last item
based on index
		auto itr = m_StackMap.begin();
		int index = -itr->first;
		int value = itr->second;
		return value;
	}

	int peekMax()
	{
		// take first item from m_ValueMap, and the first one in the
queue
		auto itr = m_ValueMap.begin();
		int value = -itr->first;
		int index = itr->second.top();
		return value;
	}

	int popMax()
	{
```

```
            // take first item from m_ValueMap, and the first one in the
queue
            auto itr = m_ValueMap.begin();
            int value = -itr->first;
            int index = itr->second.top();
            itr->second.pop();
            if (itr->second.empty()) m_ValueMap.erase(itr->first);
            m_StackMap.erase(-index);
            return value;
        }
};

/// <summary>
/// LeetCode #355. Design Twitter
/// Design a simplified version of Twitter where users can post tweets,
/// follow/unfollow another user and
/// is able to see the 10 most recent tweets in the user's news feed.
/// Your design should support the following methods:
/// 1.postTweet(userId, tweetId): Compose a new tweet.
/// 2.getNewsFeed(userId): Retrieve the 10 most recent tweet ids in the
user's
///   news feed.
/// Each item in the news feed must be posted by users who the user
followed
/// or by the user herself.
/// Tweets must be ordered from most recent to least recent.
/// 3.follow(followerId, followeeId): Follower follows a followee.
/// 4.unfollow(followerId, followeeId): Follower unfollows a followee.
/// Example:
/// Twitter twitter = new Twitter();
/// User 1 posts a new tweet (id = 5).
/// twitter.postTweet(1, 5);
/// User 1's news feed should return a list with 1 tweet id -> [5].
/// twitter.getNewsFeed(1);
/// User 1 follows user 2.
/// twitter.follow(1, 2);
/// User 2 posts a new tweet (id = 6).
/// twitter.postTweet(2, 6);
/// User 1's news feed should return a list with 2 tweet ids -> [6, 5].
/// Tweet id 6 should precede tweet id 5 because it is posted after tweet
id 5.
/// twitter.getNewsFeed(1);
///
/// User 1 unfollows user 2.
/// twitter.unfollow(1, 2);
/// User 1's news feed should return a list with 1 tweet id -> [5],
/// since user 1 is no longer following user 2.
/// twitter.getNewsFeed(1);
/// </summary>
class Twitter
{
private:
    long m_TimeTicks;

      unordered_map<int, vector<pair<long, int>>> m_TwitterList;
    unordered_map<int, unordered_set<int>> m_FollowList;

public:
```

```
    /// <summary>
    /// Constructor an empty Twitter cache
    /// </summary>
    /// <returns></returns>
    Twitter()
    {
        m_TimeTicks = 0;
    }

    /// <summary>
    /// Destructor of an Twitter
    /// </summary>
    /// <returns></returns>
    ~Twitter()
    {
    }

    /// <summary>
    /// Compose a new tweet
    /// </summary>
    /// <returns></returns>
    void postTweet(int userId, int tweetId)
    {
        m_TimeTicks++;
        long now = m_TimeTicks;

        m_TwitterList[userId].push_back(make_pair(m_TimeTicks, tweetId));
    }

    /// <summary>
    /// Retrieve the 10 most recent tweet ids in the user's news feed.
    /// Each item in the news feed must be posted by users who the user
      /// followed or by the user herself.
    /// Tweets must be ordered from most recent to least recent.
    /// </summary>
    /// <returns></returns>
    vector<int> getNewsFeed(int userId)
    {
        vector<int> result;
        priority_queue <pair<long, pair<int,int>>> candidate_list;
             if (!m_TwitterList[userId].empty())
             {
                   int clock = m_TwitterList[userId].back().first;
                   int index = (int)m_TwitterList[userId].size() - 1;
                   candidate_list.push(make_pair(clock, make_pair(userId,
index)));
             }
             for (int followee : m_FollowList[userId])
             {
                   if (!m_TwitterList[followee].empty())
                   {
                         int clock = m_TwitterList[followee].back().first;
                         int index = (int)m_TwitterList[followee].size() -
1;
                         candidate_list.push(make_pair(clock,
make_pair(followee, index)));
                   }
             }
```

```
            for (size_t i = 0; i < 10; i++)
            {
                  if (candidate_list.empty())
                  {
                        break;
                  }
                  pair<long, pair<int, int>> tweet_itr =
candidate_list.top();
                  candidate_list.pop();
                  int user_id = tweet_itr.second.first;
                  int index = tweet_itr.second.second;
                  int tweet_id = m_TwitterList[user_id][index].second;
                  result.push_back(tweet_id);
                  // push the next tweet of this user to priority queue
                  if (index > 0)
                  {
                        index--;
                        int clock = m_TwitterList[user_id][index].first;
                        candidate_list.push(make_pair(clock,
make_pair(user_id, index)));
                  }
            }

        return result;
    }

    /// <summary>
    /// Follower follows a followee. If the operation is invalid, it should
      /// be a no-op.
    /// </summary>
    /// <param name="followerId">The follower id</param>
    /// <param name="followeeId">The followee id</param>
    /// <returns></returns>
    void follow(int followerId, int followeeId)
    {
        if (followerId != followeeId)
        {
            m_FollowList[followerId].insert(followeeId);
        }
    }

    /// <summary>
    /// Follower unfollows a followee. If the operation is invalid, it
should
      /// be a no-op.
    /// </summary>
    /// <param name="followerId">The follower id</param>
    /// <param name="followeeId">The followee id</param>
    /// <returns></returns>
    void unfollow(int followerId, int followeeId)
    {
        m_FollowList[followerId].erase(followeeId);
    }
};

/// <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;
    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;
        }
    }
};


/// <summary>
/// Leet code #460. LFU Cache
/// Design and implement a data structure for Least Frequently
/// Used (LFU) cache. 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 reaches its capacity, it should
invalidate
///                   the least frequently used item before inserting a new
///                   item. For the purpose of this problem, when there is
a
///                   tie (i.e., two or more keys that have the same
frequency),
///                   the least recently used key would be evicted.
///
/// Follow up:
/// Could you do both operations in O(1) time complexity?
/// Example:
/// LFUCache cache = new LFUCache( 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.get(3);       // returns 3.
/// 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>
class LFUCache
{
private:
    size_t m_capacity;
    int m_minFreq;
    // map key to the frequency list position
    unordered_map<int, pair<int, list<pair<int, int>>::iterator>> m_keyMap;
    // map frequency to the value list
    unordered_map<int, list<pair<int, int>>> m_freqMap;
public:
    LFUCache(int capacity)
    {
        m_capacity = capacity;
        m_minFreq = 0;
    }

    int get(int key)
    {
        if (m_keyMap.count(key) == 0)
        {
            return -1;
        }
        pair<int, list<pair<int, int>>::iterator> freq_itr = m_keyMap[key];
        int frequency = freq_itr.first;
        list<pair<int, int>>::iterator key_val_itr = freq_itr.second;
        pair<int, int> key_val = *key_val_itr;
        // key_val_itr become invalid
        m_freqMap[frequency].erase(key_val_itr);
        if (m_freqMap[frequency].empty())
        {
            m_freqMap.erase(frequency);
            if (m_minFreq == frequency) m_minFreq++;
```

```
        }
        frequency++;
        m_freqMap[frequency].push_front(key_val);
        // new key value pair iterator
        key_val_itr = m_freqMap[frequency].begin();
        // assign back to kay map
        m_keyMap[key] = make_pair(frequency, key_val_itr);
        return key_val.second;
    }

    void put(int key, int value)
    {
        if (m_capacity == 0) return;
        list<pair<int, int>>::iterator key_val_itr;
        if (get(key) != -1)
        {
            key_val_itr = m_keyMap[key].second;
            key_val_itr->second = value;
        }
        else
        {
            if (m_keyMap.size() == m_capacity)
            {
                // erase LFU key from frequency map
                pair<int, int> key_val = m_freqMap[m_minFreq].back();
                m_freqMap[m_minFreq].pop_back();
                        if (m_freqMap[m_minFreq].empty())
                        {
                              m_freqMap.erase(m_minFreq);
                        }
                // erase LFU key from key map
                m_keyMap.erase(key_val.first);
            }
            m_minFreq = 1;
            m_freqMap[m_minFreq].push_front(make_pair(key,value));
            // new key value pair iterator
            key_val_itr = m_freqMap[m_minFreq].begin();
            // assign back to kay map
            m_keyMap[key] = make_pair(m_minFreq, key_val_itr);
        }
    }
};

/// <summary>
/// Leet code #341. Flatten Nested List Iterator
/// Given a nested list of integers, implement an iterator to flatten it.
/// Each element is either an integer, or a list -- whose elements may also
/// be integers or other lists.
///
/// Example 1:
/// Given the list [[1,1],2,[1,1]],
/// By calling next repeatedly until hasNext returns false, the order of
/// elements returned by next should be: [1,1,2,1,1].
///
/// Example 2:
/// Given the list [1,[4,[6]]],
/// By calling next repeatedly until hasNext returns false, the order of
/// elements returned by next should be: [1,4,6].
```

```
///
/// This is the interface that allows for creating nested lists.
/// You should not implement it, or speculate about its implementation
/// Your NestedIterator object will be instantiated and called as such:
/// NestedIterator i(nestedList);
/// while (i.hasNext()) cout << i.next();
/// </summary>
class NestedInteger
{
private:
    vector<NestedInteger> m_NestedIntegers;
    int m_Integer;
    bool m_IsInteger;
    int m_Size;
public:
    // Constructure on single integter.
    NestedInteger()
    {
        m_Size = 0;
    }

    // Constructure on single integter.
    NestedInteger(int value)
    {
        m_Integer = value;
        m_IsInteger = true;
        m_Size = 1;
    }

    // Constructure on nested integers.
    NestedInteger(vector<NestedInteger> nested_integers)
    {
        m_NestedIntegers = nested_integers;
        m_IsInteger = false;
        m_Size = nested_integers.size();
    }

    // Return true if this NestedInteger holds a single integer, rather
than a nested list.
    bool isInteger() const
    {
        return m_IsInteger;
    };

    // Return the single integer that this NestedInteger holds, if it holds
a single integer
    // The result is undefined if this NestedInteger holds a nested list
    int getInteger() const
    {
        return m_Integer;
    };

    // Set this NestedInteger to hold a single integer.
    void setInteger(int value)
    {
        m_Integer = value;
        m_IsInteger = true;
        m_NestedIntegers.clear();
```

```
        m_Size = 1;
    };

    // Set this NestedInteger to hold a nested list and adds a nested
integer to it.
    void add(const NestedInteger &ni)
    {
        if (m_Size == 1 && isInteger())
        {
            m_NestedIntegers.push_back(m_Integer);
        }
        m_NestedIntegers.push_back((NestedInteger)ni);
        m_IsInteger = false;
        m_Size++;
    };

    // Return the nested list that this NestedInteger holds, if it holds a
nested list
    // The result is undefined if this NestedInteger holds a single integer
    const vector<NestedInteger> &getList() const
    {
        return m_NestedIntegers;
    };
};


class NestedIterator {
private:
    deque<NestedInteger> m_NestedQueue;
    bool isEmpty(NestedInteger& ni)
    {
        if (ni.isInteger())
        {
            return false;
        }
        else
        {
            vector<NestedInteger> ni_list = ni.getList();
            for (size_t i = 0; i < ni_list.size(); i++)
            {
                if (!isEmpty(ni_list[i])) return false;
            }
            return true;
        }
    }

public:
    NestedIterator(vector<NestedInteger> &nestedList)
    {
        for (size_t i = 0; i < nestedList.size(); i++)
        {
            m_NestedQueue.push_back(nestedList[i]);
        }
    }

    int next()
    {
        int value = 0;
```

```
        if (!m_NestedQueue.empty())
        {
            NestedInteger nested_integer = m_NestedQueue.front();
            m_NestedQueue.pop_front();
            if (nested_integer.isInteger())
            {
                value = nested_integer.getInteger();
            }
            else
            {
                vector<NestedInteger> nested_list =
nested_integer.getList();
                while (!nested_list.empty())
                {
                    m_NestedQueue.push_front(nested_list.back());
                    nested_list.pop_back();
                }
                value = next();
            }
        }
        return value;
    }

    bool hasNext()
    {
        bool has_next = false;

        while (!m_NestedQueue.empty())
        {
            NestedInteger ni = m_NestedQueue.front();
            if (isEmpty(ni))
            {
                m_NestedQueue.pop_front();
            }
            else
            {
                return true;
            }
        }
        return false;
    }
};
```

Binary Search II

One of the reason that we need to use binary search is due to split the data set and check the edge to see if the answer we want to find is in which side.

```
/// <summary>
/// Leet code #540. Single Element in a Sorted Array
///
/// Given a sorted array consisting of only integers where every element
/// appears twice except for one element which appears once. Find this
/// single element that appears only once.
/// Example 1:
/// Input: [1,1,2,3,3,4,4,8,8]
/// Output: 2
```

```
///
/// Example 2:
/// Input: [3,3,7,7,10,11,11]
/// Output: 10
/// Note: Your solution should run in O(log n) time and O(1) space.
/// </summary>
int LeetCode::singleNonDuplicate(vector<int>& nums)
{
    int begin = 0, end = nums.size();
    while (begin < end - 1)
    {
        int mid = begin + (end - begin) / 2;
        if ((mid + 1 < end) && (nums[mid] == nums[mid + 1]))
        {
            if (mid > begin) mid--;
            else mid++;
        }
        // midd point to a actul number, so should calculate as + 1
        if ((mid - begin + 1) % 2 == 0) begin = mid + 1;
        else end = mid + 1;
    }
    return nums[begin];
}
```

In some of the problems, finding the answer directly is difficult, but it is easy to guess a answer first, the check if your guess is too small or too large, then this problem is translated to a binary search mode. But you should only do so if you are sure the statement in yellow is true.

```
/// <summary>
/// Leet code #378. Kth Smallest Element in a Sorted Matrix
/// </summary>
int LeetCode::countNoGreaterValue(vector<vector<int>>& matrix, int value,
int k)
{
      int i = 0, j = matrix[0].size() - 1;
      int count = 0;
      while (i < (int)matrix.size() && j >= 0)
      {
            if (matrix[i][j] <= value)
            {
                  i++;
                  count += j + 1;
                  if (count > k) return count;
            }
            else j--;
      }
      return count;
}


/// <summary>
/// Leet code #378. Kth Smallest Element in a Sorted Matrix
///
/// Given a n x n matrix where each of the rows and columns are sorted
```

```
/// in ascending order,
/// find the kth smallest element in the matrix. Note that it is the kth
/// smallest element in the sorted order, not the kth distinct element.
/// Example:
/// matrix =
/// [
///   [ 1,  5,  9],
///   [10, 11, 13],
///   [12, 13, 15]
/// ],
/// k = 8,
/// return 13.
/// </summary>
int LeetCode::kthSmallest(vector<vector<int>>& matrix, int k)
{
	int low = matrix[0][0];
	int high = matrix[matrix.size() - 1][matrix[0].size() - 1];
	while (low < high)
	{
		int mid = low + (high - low) / 2;
		if (countNoGreaterValue(matrix, mid, k) < k)
		{
			low = mid + 1;
		}
		else
		{
			high = mid;
		}
	}
	return low;
}
```

However, if you are asked to return first K pairs, the binary search may not be the best solution, and you may consider the alternative solution, heap.

```
/// <summary>
/// Leet code #373. Find K Pairs with Smallest Sums
/// You are given two integer arrays nums1 and nums2 sorted in ascending
/// order and an integer k.
/// Define a pair (u,v) which consists of one element from the first array
/// and one element from the second array.
/// Find the k pairs (u1,v1),(u2,v2) ...(uk,vk) with the smallest sums.
/// Example 1:
/// Given nums1 = [1,7,11], nums2 = [2,4,6],  k = 3
/// Return: [1,2],[1,4],[1,6]
/// The first 3 pairs are returned from the sequence:
/// [1,2],[1,4],[1,6],[7,2],[7,4],[11,2],[7,6],[11,4],[11,6]
///
/// Example 2:
/// Given nums1 = [1,1,2], nums2 = [1,2,3],  k = 2
/// Return: [1,1],[1,1]
/// The first 2 pairs are returned from the sequence:
/// [1,1],[1,1],[1,2],[2,1],[1,2],[2,2],[1,3],[1,3],[2,3]
///
/// Example 3:
/// Given nums1 = [1,2], nums2 = [3],  k = 3
/// Return: [1,3],[2,3]
```

```
/// All possible pairs are returned from the sequence:
/// [1,3],[2,3]
/// </summary>
vector<pair<int, int>> LeetCode::kSmallestPairs(vector<int>& nums1,
vector<int>& nums2, int k)
{
    vector<pair<int, int>> result;
    if (nums1.empty() || nums2.empty()) return result;
    vector<int> array1(nums1.size(), 0);
    priority_queue<pair<int, int>> priority_queue;
    for (size_t i = 0; i < nums1.size(); i++)
    {
        priority_queue.push(make_pair(-(nums1[i] + nums2[0]), i));
    }
    for (int i = 0; i < k; i++)
    {
        if (priority_queue.empty()) break;
        pair<int, int> value_pair = priority_queue.top();
        priority_queue.pop();
        result.push_back(make_pair(nums1[value_pair.second],
nums2[array1[value_pair.second]]));
        array1[value_pair.second]++;
        if (array1[value_pair.second] < (int)nums2.size())
        {
            priority_queue.push(make_pair(-(nums1[value_pair.second] +
nums2[array1[value_pair.second]]), value_pair.second));
        }
    }
    return result;
}

/// <summary>
/// Leet code #668. Kth Smallest Number in Multiplication Table
/// Nearly every one have used the Multiplication Table. But could you
/// find out the k-th smallest number quickly from the multiplication
/// table?
///
/// Given the height m and the length n of a m * n Multiplication Table,
/// and a positive integer k, you need to return the k-th smallest number
/// in this table.
///
/// Example 1:
/// Input: m = 3, n = 3, k = 5
/// Output: 3
/// Explanation:
/// The Multiplication Table:
/// 1    2    3
/// 2    4    6
/// 3    6    9
///
/// The 5-th smallest number is 3 (1, 2, 2, 3, 3).
/// Example 2:
/// Input: m = 2, n = 3, k = 6
/// Output:
/// Explanation:
/// The Multiplication Table:
/// 1    2    3
/// 2    4    6
```

```
///
/// The 6-th smallest number is 6 (1, 2, 2, 3, 4, 6).
/// Note:
/// The m and n will be in the range [1, 30000].
/// The k will be in the range [1, m * n]
/// </summary>
int LeetCode::findKthNumber(int m, int n, int k)
{
    int first = 1, last = m * n;
    while (first < last)
    {
        int v = (first + last) / 2;
        int count = 0;
        for (int i = 1; i <= m; i++)
        {
            count += min(n, v / i);
        }
        if (count < k) first = v + 1;
        else last = v;
    }
    return first;
}

/// <summary>
/// Leet code #719. Find K-th Smallest Pair Distance
/// Given an integer array, return the k-th smallest distance among all
/// the pairs. The distance of a pair (A, B) is defined as the absolute
/// difference between A and B.
///
/// Example 1:
/// Input:
/// nums = [1,3,1]
/// k = 1
/// Output: 0
/// Explanation:
/// Here are all the pairs:
/// (1,3) -> 2
/// (1,1) -> 0
/// (3,1) -> 2
/// Then the 1st smallest distance pair is (1,1), and its distance is 0.
/// Note:
/// 1. 2 <= len(nums) <= 10000.
/// 2. 0 <= nums[i] < 1000000.
/// 3. 1 <= k <= len(nums) * (len(nums) - 1) / 2.
/// </summary>
```

At the first glance, you may think the problem can be resolved by using BST to build a size of K heap. The complexity of N^2 * log(k) may be too big. Another important fact is that count how many distances which are below certain value v is a O(N) instead of O(N^2) algorithm.

```
/// <summary>
/// Leet code #719. Find K-th Smallest Pair Distance
/// </summary>
int LeetCode::countNoGreaterDistance(vector<int>& nums, int distance, int
k)
{
```

```
	int count = 0;
	int j = 1;
	for (size_t i = 0; i < nums.size(); i++)
	{
		while (j < (int)nums.size() && nums[j] - nums[i] <= distance)
j++;
		count += j - i - 1;
		if (count > k) return count;
	}
	return count;
}

int LeetCode::smallestDistancePair(vector<int>& nums, int k)
{
	sort(nums.begin(), nums.end());
	// Minimum absolute difference
	int low = nums[1] - nums[0];
	for (int i = 1; i < (int)nums.size() - 1; i++)
	{
		low = min(low, nums[i + 1] - nums[i]);
	}
	int high = nums[nums.size() - 1] - nums[0];
	while (low < high)
	{
		int mid = (low + high) / 2;
		int count = countNoGreaterDistance(nums, mid, k);
		if (count >= k) high = mid;
		else low = mid + 1;
	}
	return low;
}
```

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;
```

```
/// <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;
}


/// <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;
}


/// <summary>
/// Leet code #721. Accounts Merge
///
/// Given a list accounts, each element accounts[i] is a list of strings,
/// where the first element accounts[i][0] is a name, and the rest of the
/// elements are emails representing emails of the account.
///
/// Now, we would like to merge these accounts. Two accounts definitely
/// belong to the same person if there is some email that is common to
/// both accounts. Note that even if two accounts have the same name, they
/// may belong to different people as people could have the same name. A
/// person can have any number of accounts initially, but all of their
/// accounts definitely have the same name.
///
/// After merging the accounts, return the accounts in the following
/// format: the first element of each account is the name, and the rest of
/// the elements are emails in sorted order. The accounts themselves can be
/// returned in any order.
///
/// Example 1:
/// Input:
```

```
/// accounts = [["John", "johnsmith@mail.com", "john00@mail.com"], ["John",
/// "johnnybravo@mail.com"], ["John", "johnsmith@mail.com",
/// "john_newyork@mail.com"], ["Mary", "mary@mail.com"]]
/// Output: [["John", 'john00@mail.com', 'john_newyork@mail.com',
/// 'johnsmith@mail.com'],  ["John", "johnnybravo@mail.com"], ["Mary",
/// "mary@mail.com"]]
/// Explanation:
/// The first and third John's are the same person as they have the common
/// email "johnsmith@mail.com".
/// The second John and Mary are different people as none of their email
/// addresses are used by other accounts.
/// We could return these lists in any order, for example the answer
/// [['Mary', 'mary@mail.com'], ['John', 'johnnybravo@mail.com'],
/// ['John', 'john00@mail.com', 'john_newyork@mail.com',
/// 'johnsmith@mail.com']] would still be accepted.
/// Note:
///
/// 1.The length of accounts will be in the range [1, 1000].
/// 2.The length of accounts[i] will be in the range [1, 10].
/// 3.The length of accounts[i][j] will be in the range [1, 30].
/// </summary>
vector<vector<string>> LeetCode::accountsMergeII(vector<vector<string>>&
accounts)
{
	unordered_map<string, int> email_map;
	unordered_map<int, int> parent_map;
	unordered_map<int, set<string>> merged_account;
	vector<vector<string>> result;

	for (size_t i = 0; i < accounts.size(); i++)
	{
		string name = accounts[i][0];
		int self = i;
		parent_map[i] = i;
		for (size_t j = 1; j < accounts[i].size(); j++)
		{
			string email = accounts[i][j];
			if (email_map.count(email) == 0)
			{
				email_map[email] = i;
			}
			else
			{
				// union merge with root
				int parent = email_map[email];
				while (parent_map[parent] != parent) parent =
parent_map[parent];
				if (parent != self)
				{
					parent_map[self] = parent;
					self = parent;
				}
			}
		}
	}

	for (auto itr : parent_map)
	{
```

```
            for (size_t j = 1; j < accounts[itr.first].size(); j++)
            {
                  int parent = itr.second;
                  while (parent_map[parent] != parent) parent =
parent_map[parent];
                  merged_account[parent].insert(accounts[itr.first][j]);
            }
      }

      for (auto itr : merged_account)
      {
            vector<string> account;
            account.push_back(accounts[itr.first][0]);
            for (string str : itr.second) account.push_back(str);
            result.push_back(account);
      }
      return result;
}
```

Interval Count

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 #729. My Calendar I
///
/// Implement a MyCalendar class to store your events. A new event can be
/// added if adding the event will not cause a double booking.
///
/// Your class will have the 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 double booking happens when two events have some non-empty
```

```
/// intersection (ie., there is some time that is common to both events.)
///
/// For each call to the method MyCalendar.book, return true if the event
/// can be added to the calendar successfully without causing a double
/// booking. Otherwise, return false and do not add the event to the
/// calendar.
///
/// Your class will be called like this: MyCalendar cal = new MyCalendar();
/// MyCalendar.book(start, end)
///
/// Example 1:
/// MyCalendar();
/// MyCalendar.book(10, 20); // returns true
/// MyCalendar.book(15, 25); // returns false
/// MyCalendar.book(20, 30); // returns true
/// Explanation:
/// The first event can be booked.  The second can't because time 15 is
/// already booked by another event.
/// The third event can be booked, as the first event takes every time
/// less than 20, but not including 20.
/// Note:
///
/// 1. The number of calls to MyCalendar.book per test case will be at
///    most 1000.
/// 2. In calls to MyCalendar.book(start, end), start and end are integers
///    in the range [0, 10^9].
/// </summary>
class MyCalendar {
private:
	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:
	MyCalendar()
	{
		m_TimeMap[0] = 0;
	}

	bool book(int start, int end)
	{
		auto itr = getLocation(start);
		while (itr != m_TimeMap.end() && itr->first < end)
		{
			if (itr->second == 1) return false;
			itr++;
		}

		itr = getLocation(end);
		m_TimeMap[end] = itr->second;

		itr = getLocation(start);
```

```
            m_TimeMap[start] = itr->second + 1;


            itr = m_TimeMap.find(start);
            itr++;
            while (itr != m_TimeMap.end() && itr->first < end)
            {
                  m_TimeMap.erase(itr++);
            }
            return true;
      }
};

/// <summary>
/// Leet code #731. My Calendar II
///
/// Implement a MyCalendarTwo class to store your events. A new event can
/// be added if adding the event will not cause a triple booking.
///
/// 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 triple booking happens when three events have some non-empty
/// intersection (ie., there is some time that is common to all 3 events.)
///
/// For each call to the method MyCalendar.book, return true if the event
/// can be added to the calendar successfully without causing a triple
/// booking. Otherwise, return false and do not add the event to the
/// calendar.
///
/// Your class will be called like this:
/// MyCalendar cal = new MyCalendar(); MyCalendar.book(start, end)
/// Example 1:
/// MyCalendar();
/// MyCalendar.book(10, 20); // returns true
/// MyCalendar.book(50, 60); // returns true
/// MyCalendar.book(10, 40); // returns true
/// MyCalendar.book(5, 15);  // returns false
/// MyCalendar.book(5, 10);  // returns true
/// MyCalendar.book(25, 55); // returns true
/// Explanation:
/// The first two events can be booked.  The third event can be double
/// booked.
/// The fourth event (5, 15) can't be booked, because it would result
/// in a triple booking.
/// The fifth event (5, 10) can be booked, as it does not use time 10
/// which is already double booked.
/// The sixth event (25, 55) can be booked, as the time in [25, 40) will
/// be double booked with the third event;
/// the time [40, 50) will be single booked, and the time [50, 55) will be
/// double booked with the second event.
/// Note:
///
/// 1. The number of calls to MyCalendar.book per test case will be at most
/// 1000.
/// 2. In calls to MyCalendar.book(start, end), start and end are integers
in
```

```
/// the range [0, 10^9].
/// </summary>
class MyCalendarTwo {
private:
	map<int, int> m_TimeMap;
	// fine the location for the point in space which is less than or
equal
	// to the input point, the input point may start with the existing
value
	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:
	MyCalendarTwo()
	{
		m_TimeMap[0] = 0;
	}

	bool book(int start, int end)
	{

		auto itr = getLocation(start);
		while (itr != m_TimeMap.end() && itr->first < end)
		{
			if (itr->second == 2) return false;
			itr++;
		}

		itr = getLocation(end);
		m_TimeMap[end] = itr->second;

		itr = getLocation(start);
		m_TimeMap[start] = itr->second + 1;


		itr = m_TimeMap.find(start);
		itr++;
		while (itr != m_TimeMap.end() && itr->first < end)
		{
			itr->second++;
			itr++;
		}
		return true;
	}
};

/// <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;
        }
};



/// <summary>
/// Leet code #699. Falling Squares
/// </summary>
map<int, int>::iterator LeetCode::findLocation(map<int, int>& pos_map, int
pos)
{
        auto itr = pos_map.lower_bound(pos);
        if (itr == pos_map.end() || itr->first > pos) itr--;
        return itr;
}

/// <summary>
/// Leet code #699. Falling Squares
///
/// On an infinite number line (x-axis), we drop given squares in the order
/// they are given.
///
/// The i-th square dropped (positions[i] = (left, side_length)) is a
/// square with the left-most point being positions[i][0] and sidelength
```

```
/// positions[i][1].
///
/// The square is dropped with the bottom edge parallel to the number line,
/// and from a higher height than all currently landed squares. We wait for
/// each square to stick before dropping the next.
///
/// The squares are infinitely sticky on their bottom edge, and will remain
/// fixed to any positive length surface they touch (either the number line
/// or another square). Squares dropped adjacent to each other will not
/// stick together prematurely.
///
/// Return a list ans of heights. Each height ans[i] represents the current
/// highest height of any square we have dropped, after dropping squares
/// represented by positions[0], positions[1], ..., positions[i].
///
/// Example 1:
/// Input: [[1, 2], [2, 3], [6, 1]]
/// Output: [2, 5, 5]
/// Explanation:
///
/// After the first drop of
/// positions[0] = [1, 2]:
/// _aa
/// _aa
/// -------
/// The maximum height of any square is 2.
///
/// After the second drop of
/// positions[1] = [2, 3]:
/// __aaa
/// __aaa
/// __aaa
/// _aa__
/// _aa__
/// --------------
/// The maximum height of any square is 5.
/// The larger square stays on top of the smaller square despite where its
/// center of gravity is, because squares are infinitely sticky on their
/// bottom edge.
///
/// After the third drop of
/// positions[1] = [6, 1]:
/// __aaa
/// __aaa
/// __aaa
/// _aa
/// _aa___a
/// --------------
/// The maximum height of any square is still 5.
///
/// Thus, we return an answer of
/// [2, 5, 5]
///
/// Example 2:
/// Input: [[100, 100], [200, 100]]
/// Output: [100, 100]
/// Explanation: Adjacent squares don't get stuck prematurely - only their
/// bottom edge can stick to surfaces.
```

```
///
/// Note:
/// 1 <= positions.length <= 1000.
/// 1 <= positions[i][0] <= 10^8.
/// 1 <= positions[i][1] <= 10^6.
/// </summary>
vector<int> LeetCode::fallingSquares(vector<pair<int, int>>& positions)
{
	vector<int> result;
	// remember the start position and height
	map<int, int> pos_map;
	pos_map[0] = 0;
	int max_height = 0;
	for (size_t i = 0; i < positions.size(); i++)
	{
		auto start = findLocation(pos_map, positions[i].first);
		auto end = findLocation(pos_map, positions[i].first +
positions[i].second);
		int end_height = end->second;
		int height = 0;
		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);
			}
		}
		// add height on this box
		height += positions[i].second;
		// set start as new height
		pos_map[positions[i].first] = height;
		// set next to end as its original height
		pos_map[positions[i].first + positions[i].second] = end_height;
		// calculate max height
		max_height = max(max_height, height);
		result.push_back(max_height);
	}
	return result;
}
```

Conditional Dynamic Programming

So far in all the conditional dynamic programming, the search target is static. However there is a scanrio when the search target is moving, for example two people are walking along some paths to pick up cherries, to maximum the total harvest, the best path of the second peron depends on the path of the first person.

Please look at the example below:

```
/// <summary>
/// Leet code #741. Cherry Pickup
///
```

```cpp
/// In a N x N grid representing a field of cherries, each cell is one of
/// three possible integers.
///
/// 0 means the cell is empty, so you can pass through;
/// 1 means the cell contains a cherry, that you can pick up and pass
/// through;
/// -1 means the cell contains a thorn that blocks your way.
/// Your task is to collect maximum number of cherries possible by
/// following the rules below:
///
/// Starting at the position (0, 0) and reaching (N-1, N-1) by moving
/// right or down through valid path cells (cells with value 0 or 1);
/// After reaching (N-1, N-1), returning to (0, 0) by moving left or up
/// through valid path cells;
/// When passing through a path cell containing a cherry, you pick it up
/// and the cell becomes an empty cell (0);
/// If there is no valid path between (0, 0) and (N-1, N-1), then no
/// cherries can be collected.
/// Example 1:
/// Input: grid =
/// [[0, 1, -1],
///  [1, 0, -1],
/// [1, 1,  1]]
/// Output: 5
/// Explanation:
/// The player started at (0, 0) and went down, down, right right to reach
/// (2, 2).
/// 4 cherries were picked up during this single trip, and the matrix
/// becomes [[0,1,-1],[0,0,-1],[0,0,0]].
/// Then, the player went left, up, up, left to return home, picking up one
/// more cherry.
/// The total number of cherries picked up is 5, and this is the maximum
/// possible.
///
/// Note:
/// grid is an N by N 2D array, with 1 <= N <= 50.
/// Each grid[i][j] is an integer in the set {-1, 0, 1}.
/// It is guaranteed that grid[0][0] and grid[N-1][N-1] are not -1.
/// </summary>
int LeetCode::cherryPickup(vector<vector<int>>& grid)
{
    size_t n = grid.size();
    if (n == 0) return 0;

    vector<vector<int>> dp(n, vector<int>(n, -1));
    dp[0][0] = 0;
    // iterate 2 N - 1 steps
    for (size_t k = 0; k < 2 * grid.size() - 1; k++)
    {
        vector<vector<int>> next(n, vector<int>(n, -1));
        // path one has i steps down
        for (size_t i = 0; i <= k && i < n; i++)
        {
            // too few steps down on total k steps
            if (k - i >= n) continue;
            // path two has j steps down
            for (size_t j = 0; j <= k && j < n; j++)
            {
```

```
                // too few steps down on total k steps
                if (k - j >= n) continue;
                // there is a thorn in of them path not possible
                if ((grid[i][k - i] < 0) || (grid[j][k - j] < 0))
                {
                    continue;
                }
                // path one from left to right, path two from left to right
                int cherries = dp[i][j];
                // path one from up to down, path two from up to down
                if ((i > 0) && (j > 0)) cherries = max(cherries, dp[i - 1]
[j - 1]);
                // path one from up to down, path two from left to right
                if (i > 0) cherries = max(cherries, dp[i - 1][j]);
                // path one from left to right, path two from up to down
                if (j > 0) cherries = max(cherries, dp[i][j - 1]);
                // due to the block no way to reach the position

                if (cherries < 0) continue;

                // add the current position on path one
                cherries += grid[i][k - i];
                // add the current position on path two.
                if (i != j) cherries += grid[j][k - j];

                next[i][j] = cherries;

            }
        }
        // copy over current steps
        dp = next;
    }
    return max(0, dp[n - 1][n - 1]);
}
```

For the above soluton, we convert the problem to allow two people both move from left top to right bottom.

We think this way we start from step 1 to step 2 * N -1. On each step, we exhaust all the possible fpositions for person one and person two, we calculate how many cherries these two people can pick up in maximum. Please notice if the position are overlap, the cherries are collected once.

To find out all final positions, we can say in step K, a specific person x, he can move I steps down and K-I steps right, as long as both 0<=i< N and 0<=K-I<N, if the there is a block at the final position we skip this choice.

Finally we iterate step K to step K+1. This is to say if we test all the K-1 steps, and we know on every possible final position what is the maximum cherry pick, we should be able to calculate the maximum cherry pick up PosX and PosY.

For the above problem, I have another more straightforward solution, instead of speculating how many steps down and how many steps right, we simply calculate what are the possible next positions for two person on step K.

```
int LeetCode::cherryPickupII(vector<vector<int>>& grid)
```

```
{
    int n = grid.size();
    if (n == 0) return 0;

    // on any position combination what is the maximum cherry pick at
specific step.
    map<pair<pair<int, int>, pair<int, int>>, int> dp;
    // all possible position for perosn one and person two at specific
step.
    vector<vector<int>> directions = { {1, 0}, {0, 1} };
    // iterate 2 N - 1 steps
    for (int k = 0; k < 2 * n - 1; k++)
    {
        map<pair<pair<int, int>, pair<int, int>>, int> next;
        if (k == 0)
        {
            pair<int, int> pos = make_pair(0, 0);
            next[make_pair(pos, pos)] = grid[0][0];
        }
        else
        {
            for (auto itr : dp)
            {
                pair<int, int> pos1 = itr.first.first;
                pair<int, int> pos2 = itr.first.second;
                for (size_t i = 0; i < 2; i++)
                {
                    pair<int, int> next_pos1 = pos1;
                    next_pos1.first += directions[i][0];
                    next_pos1.second += directions[i][1];
                    if ((next_pos1.first >= n) || (next_pos1.second >= n)
|| (grid[next_pos1.first][next_pos1.second] < 0))
                    {
                        continue;
                    }
                    for (size_t j = 0; j < 2; j++)
                    {
                        pair<int, int> next_pos2 = pos2;
                        next_pos2.first += directions[j][0];
                        next_pos2.second += directions[j][1];
                        if ((next_pos2.first >= n) || (next_pos2.second >=
n) || (grid[next_pos2.first][next_pos2.second] < 0))
                        {
                            continue;
                        }

                        int value = dp[make_pair(pos1, pos2)] +
grid[next_pos1.first][next_pos1.second];
                        if (next_pos1 != next_pos2)
                        {
                             value += grid[next_pos2.first]
[next_pos2.second];
                        }
                        next[make_pair(next_pos1, next_pos2)] =
max(next[make_pair(next_pos1, next_pos2)], value);
                    }
                }
            }
```

```
            }
            dp = next;
        }
        pair<int, int> pos = make_pair(n - 1, n - 1);
        return dp[make_pair(pos, pos)];
    }
```

Word Break

There two word-break problems, the first one just asks if you can break it, the second one requires that you return all possible sentences.

Please think why the second one can not use DP to resolve it.

```
/// <summary>
/// Leet code #139. Word Break
/// Recursive break the word according to diction
/// </summary>
bool LeetCode::wordBreak(string s, unordered_set<string>& wordDict,
unordered_map<string, bool>&search_map)
{
    if (s.size() == 0) return true;
    if (search_map.find(s) != search_map.end())
    {
        return search_map[s];
    }

    for (int i = (int)(s.size() -1); i >= 0; i--)
    {
        string sub_word = s.substr(0, i + 1);
        if (wordDict.find(sub_word) != wordDict.end())
        {
            if (i == s.size() - 1) return true;
            search_map[sub_word] = true;
            if (wordBreak(s.substr(i + 1), wordDict, search_map))
            {
                return true;
            }
        }
        search_map[sub_word] = false;
    }
    return false;
}

/// <summary>
/// Leet code #139. Word Break
/// Given a string s and a dictionary of words dict, determine if s can be
/// segmented into a space-separated sequence
/// of one or more dictionary words.
/// For example, given
/// s = "leetcode",
/// dict = ["leet", "code"].
/// Return true because "leetcode" can be segmented as "leet code".
/// </summary>
bool LeetCode::wordBreak(string s, unordered_set<string>& wordDict)
{
```

```
    unordered_map<string, bool> search_map;
    return wordBreak(s, wordDict, search_map);
}

/// <summary>
/// Leet code #139. Word Break
/// Given a string s and a dictionary of words dict, determine if s can
/// be segmented into a space-separated sequence
/// of one or more dictionary words.
/// For example, given
/// s = "leetcode",
/// dict = ["leet", "code"].
/// Return true because "leetcode" can be segmented as "leet code".
/// </summary>
bool LeetCode::wordBreakDP(string s, unordered_set<string>& wordDict)
{
    vector<int> word_break;
    word_break.push_back(-1);

    for (size_t i = 0; i < s.size(); i++)
    {
        int size = word_break.size();
        for (int j = size - 1; j >= 0; j--)
        {
            int start = word_break[j] + 1;
            string sub_string = s.substr(start, i - start + 1);
            if (wordDict.find(sub_string) != wordDict.end())
            {
                word_break.push_back(i);
                break;
            }
        }
    }
    if (word_break.back() == s.size() - 1)
    {
        return true;
    }
    else
    {
        return false;
    }
}


/// <summary>
/// Leet code #140. Word Break II
/// Recursive break the word according to dictionary, return word list
/// </summary>
vector<string> LeetCode::wordBreakII(string s, unordered_set<string>&
wordDict,
    unordered_map<string, vector<string>>&search_map)
{
    vector<string> result;

    if (s.size() == 0)
    {
        return result;
    }
```

```
    if (search_map.find(s) != search_map.end())
    {
        return search_map[s];
    }

    for (size_t i = 0; i < s.size(); i++)
    {
        string sub_word = s.substr(0, i + 1);
        if (wordDict.find(sub_word) != wordDict.end())
        {
            if (i == s.size() - 1)
            {
                result.push_back(sub_word);
            }
            else
            {
                vector<string> sentence_list = wordBreakII(s.substr(i + 1),
wordDict, search_map);
                for (string sentence : sentence_list)
                {
                    result.push_back(sub_word + " " + sentence);
                }
            }
        }
    }
    search_map[s] = result;
    return result;
}

/// <summary>
/// Leet code #140. Word Break II
/// Given a string s and a dictionary of words dict, add spaces in s to
construct a sentence where each word is a valid dictionary word.
/// Return all such possible sentences.
/// For example, given
/// s = "catsanddog",
/// dict = ["cat", "cats", "and", "sand", "dog"].
/// A solution is ["cats and dog", "cat sand dog"].
/// </summary>
vector<string> LeetCode::wordBreakII(string s, vector<string>& wordDict)
{
    unordered_map<string, vector<string>> search_map;
    unordered_set<string> word_set;
    for (string str : wordDict) word_set.insert(str);
    return wordBreakII(s, word_set, search_map);
}
```

The reason that word break II can not be resolved by DP is not because the algorithm itself is expensive in time spending. The real root cause is too many possibilities on the combination of result!

24 Game

The 24 game is hard in the sense that code is long, but actually, the idea is very straightforward. You need to think the following:

1. The different order of the numbers
2. The different combinations on operations, be careful on divide by zero.
3. Do you need fraction to represent the intermediate result?

My answer to the above three questions are:

1. Use simple back tracking to result all possible orders in number, even it is duplicated, we do not care, because only 24 of them. (permutation 4 = 4 *3*2*1)
2. We use 2-D DP (expand from 1 number to 2 number then 3, finally 4 numbers) to calculate all possible result. We can also use backtracking with memorization, but theratically they are same.
3. Because the number is only 1-9, no need to use fraction, just use float or double, the maximum precision requirement is 0.001.

```
/// <summary>
/// Leet code #679. 24 Game
///
/// You have 4 cards each containing a number from 1 to 9. You need to
/// judge whether they could operated through *, /, +, -, (, ) to get
/// the value of 24.
///
/// Example 1:
/// Input: [4, 1, 8, 7]
/// Output: True
/// Explanation: (8-4) * (7-1) = 24
/// Example 2:
/// Input: [1, 2, 1, 2]
/// Output: False
/// Note:
/// The division operator / represents real division, not integer
/// division. For example, 4 / (1 - 2/3) = 12.
/// Every operation done is between two numbers. In particular, we cannot
/// use - as a unary operator. For example, with [1, 1, 1, 1] as input,
/// the expression -1 - 1 - 1 - 1 is not allowed.
/// </summary>
bool LeetCode::judgePoint24(vector<int>& nums)
{
    vector<bool> visited(nums.size());

    vector<int> path;
    vector<vector<int>> result;
    getPoint24Rotation(nums, path, visited, result);
    for (size_t i = 0; i < result.size(); i++)
    {
        if (calculatePoint24(result[i])) return true;
    }
    return false;
}
```

```
/// <summary>
/// Leet code #679. 24 Game
/// </summary>
void LeetCode::getPoint24Rotation(vector<int>& nums, vector<int> &path,
vector<bool> &visited, vector<vector<int>>& result)
{
    if (path.size() == nums.size())
    {
        result.push_back(path);
    }
    for (size_t i = 0; i < nums.size(); i++)
    {
        if (visited[i]) continue;
        path.push_back(nums[i]);
        visited[i] = true;
        getPoint24Rotation(nums, path, visited, result);
        path.pop_back();
        visited[i] = false;
    }
}

/// <summary>
/// Leet code #679. 24 Game
/// </summary>
bool LeetCode::calculatePoint24(vector<int>& nums)
{
    vector<vector<vector<double>>> dp =
        vector<vector<vector<double>>>(nums.size(), vector
<vector<double>>(nums.size(), vector<double>()));
    for (int len = 0; len < 4; len++)
    {
        for (size_t i = 0; i < nums.size(); i++)
        {
            if (i + len >= nums.size()) continue;
            if (len == 0)
            {
                dp[i][i + len].push_back(nums[i]);
            }
            else
            {
                for (size_t j = i; j < i + len; j++)
                {
                    for (double first : dp[i][j])
                    {
                        for (double second : dp[j + 1][i + len])
                        {
                            dp[i][i + len].push_back(first + second);
                            dp[i][i + len].push_back(first - second);
                            dp[i][i + len].push_back(first * second);
                            if (abs(second) > 0.001) dp[i][i +
len].push_back(first / second);
                        }
                    }
                }
            }
        }
    }
    for (double value : dp[0][nums.size() - 1])
```

```
    {
        if (abs(value - (double)24.0) < 0.001) return true;
    }
    return false;
}
```

Math

Reservoir Sampling

https://en.wikipedia.org/wiki/Reservoir_sampling

Reservoir sampling is a solution when you want to get a random number in a variable space size. The common scenario is that you have an unknown size of the queue and you need to decide which item to pick when the queue ends and give each item a fair chance.

The basic idea is that starting from size one, there is 100% to choose the first item. But when the queue size become 2, there is ½ chance that the selected item will be **replaced** by the second item. So if the queue ends up with size of two, each item has 50% of chance. Now consider the queue size is 3, the chance to choose the first item is (1-1/2)*(1-1/3), the second one has a chance of 1/2 * (1-1/3) and the last one is 1/3. The formula can be easily extended to a number N.

```
/// <summary>
/// Leet code #398. Random Pick Index
/// Given an array of integers with possible duplicates, randomly output
the
/// index of a given target number.
/// You can assume that the given target number must exist in the array.
///
/// Note:
/// The array size can be very large. Solution that uses too much extra
/// space will not pass the judge.
///
/// Example:
/// int[] nums = new int[] {1,2,3,3,3};
/// Solution solution = new Solution(nums);
/// pick(3) should return either index 2, 3, or 4 randomly. Each index
should have
/// equal probability of returning.
/// solution.pick(3);
/// pick(1) should return 0. Since in the array only nums[0] is equal to 1.
/// solution.pick(1);
/// </summary>
int LeetCode::pickRandom(vector<int>&nums, int target)
{
    int count = 0;
    int value = -1;
    for (size_t i = 0; i < nums.size(); i++)
    {
        if (nums[i] != target) continue;
        count++;
        if (rand() % count == 0)
        {
            value = i;
        }
    }
    return value;
}
```

```
/// <summary>
/// Leet code #384. Shuffle an Array
/// Shuffle a set of numbers without duplicates.
/// Example:
/// Init an array with set 1, 2, and 3.
/// int[] nums = {1,2,3};
/// Solution solution = new Solution(nums);
///
/// Shuffle the array [1,2,3] and return its result. Any permutation of
/// [1,2,3] must equally likely to be returned solution.shuffle();
///
/// Resets the array back to its original configuration [1,2,3].
/// solution.reset();
/// Returns the random shuffling of array [1,2,3].
/// solution.shuffle();
/// </summary>
vector<int> LeetCode::shuffle(vector<int> nums)
{
    for (size_t i = 0; i < nums.size(); i++) {
        int pos = rand() % (nums.size());
        swap(nums[pos], nums[i]);
    }
    return nums;
}
```

Bits Operation

```
/// <summary>
/// Leet code #137. Single Number II
/// Given an array of integers, every element appears three times except
for
/// one. Find that single one.
/// Note:
/// Your algorithm should have a linear runtime complexity. Could you
implement
/// it without using extra memory?
/// </summary>
int LeetCode::singleNumberII(vector<int>& nums)
{
    unsigned int result = 0;
    vector<int> bitCount(32);

    for (size_t i = 0; i < nums.size(); i++)
    {
        unsigned int x = nums[i];
        int index = 0;
        while (x != 0)
        {
            bitCount[31-index] += x % 2;
            x /= 2;
            index++;
        }
    }
    for (size_t i = 0; i < 32; i++)
    {
        bitCount[i] %= 3;
        result <<= 1;
        result |= bitCount[i];
    }
    return (int)result;
}

/// <summary>
/// Leet code #260. Single Number III
/// Given an array of numbers nums, in which exactly two elements appear
only
/// once and all the other elements
/// appear exactly twice. Find the two elements that appear only once.
///
/// Given nums = [1, 2, 1, 3, 2, 5], return [3, 5].
/// Note:
/// 1.The order of the result is not important. So in the above example,
[5, 3]
///   is also correct.
/// 2.Your algorithm should run in linear runtime complexity. Could you
implement
///   it using only constant space complexity?
/// </summary>
vector<int> LeetCode::singleNumberIII(vector<int>& nums)
{
    vector<int> result(2, 0);
    int sum = 0;
```

```
    // get the XOR for these two distinct numbers
    for (int num : nums) sum = sum ^ num;

    // get last different bit for the two numbers
    sum = sum &(-sum);

    // divide the whole list of numbers into two, one with the bit set,
    // another unset
    for (int num : nums)
    {
        if (num & sum)
        {
            result[0] ^= num;
        }
        else
        {
            result[1] ^= num;
        }
    }
    return result;
}


/// <summary>
/// Leet code # 371. Sum of Two Integers
///
/// Calculate the sum of two integers a and b, but you are not allowed to
use
/// the operator + and -.
/// Example:
/// Given a = 1 and b = 2, return 3.
/// </summary>
int LeetCode::getSum(int a, int b)
{
    int sum = a;
    int carry = 0;
    while (b != 0)
    {
        sum = a ^ b;
        carry = (a & b) << 1;
        a = sum;
        b = carry;
    }
    return sum;
}



/// <summary>
/// Leet code #477. Total Hamming Distance
/// The Hamming distance between two integers is the number of positions at
/// which the corresponding bits are different.
/// Now your job is to find the total Hamming distance between all pairs of
/// the given numbers.
/// Example:
/// Input: 4, 14, 2
/// Output: 6
///
```

```
/// Explanation: In binary representation, the 4 is 0100, 14 is 1110, and 2
/// is 0010 (just showing the four bits relevant in this case). So the
answer
/// will be:
/// HammingDistance(4, 14) + HammingDistance(4, 2) + HammingDistance(14, 2)
=
/// 2 + 2 + 2 = 6.
/// Note:
/// 1.Elements of the given array are in the range of 0  to 10^9
/// 2.Length of the array will not exceed 10^4.
/// </summary>
int LeetCode::totalHammingDistance(vector<int>& nums)
{
    int total = 0;
    for (size_t i = 0; i < 32; i++)
    {
        unsigned int bit = 1 << i;
        unsigned int zero = 0, one = 0;
        for (size_t j = 0; j < nums.size(); j++)
        {
            if ((nums[j] & bit) == 0) zero++;
            else one++;
        }
        total += one * zero;
    }
    return total;
}

/// <summary>
/// Leet code #421. Maximum XOR of Two Numbers in an Array Add to List
/// Given a non-empty array of numbers, a0, a1, a2, … , an-1, where 0 ≤ ai
< 231.
/// Find the maximum result of ai XOR aj, where 0 ≤ i, j < n.
///
/// Could you do this in O(n) runtime?
/// Example:
/// Input: [3, 10, 5, 25, 2, 8]
/// Output: 28
/// Explanation: The maximum result is 5 ^ 25 = 28.
/// </summary>
int LeetCode::findMaximumXOR(vector<int>& nums)
{
    int result = 0;
    int mask = 0;

    for (size_t bit_index = 0; bit_index < 32; bit_index++)
    {
        int bit_value = 1 << (31 - bit_index);
        mask |= bit_value;

        unordered_set<int> bits_set;
        for (size_t i = 0; i < nums.size(); i++)
        {
            bits_set.insert(nums[i] & mask);
        }

        int temp = result | bit_value;
        for (int bits : bits_set)
```

```
        {
            if (bits_set.find(temp ^ bits) != bits_set.end())
            {
                result = temp;
                break;
            }
        }
    }
    return result;
}
/// <summary>
/// Leet code #805. Split Array With Same Average
///
/// In a given integer array A, we must move every element of A to either
/// list B or list C. (B and C initially start empty.)
///
/// Return true if and only if after such a move, it is possible that the
/// average value of B is equal to the average value of C, and B and C are
/// both non-empty.
///
/// Example :
/// Input:
/// [1,2,3,4,5,6,7,8]
/// Output: true
/// Explanation: We can split the array into [1,4,5,8] and [2,3,6,7], and
/// both of them have the average of 4.5.
/// Note:
///
/// 1. The length of A will be in the range [1, 30].
/// 2. A[i] will be in the range of [0, 10000].
/// </summary>
bool LeetCode::splitArraySameAverage(vector<int>& A)
{
      int N = A.size();
      int sum = 0;
      for (int num : A) sum += num;
      vector<int> sum_map(sum + 1);
      sum_map[0] = 1;
      for (int num : A)
      {
            for (int s = sum - num; s >= 0; s--)
            {
                  sum_map[s + num] |= sum_map[s] << 1;
            }
      }
      for (int i = 1; i < N; i++)
      {
            if (sum * i % N == 0 && sum_map[sum * i / N] & (1 << i))
            {
                  return true;
            }
      }
      return false;
}

/// <summary>
/// Leet code #805. Split Array With Same Average
///
```

```
/// In a given integer array A, we must move every element of A to either
/// list B or list C. (B and C initially start empty.)
///
/// Return true if and only if after such a move, it is possible that the
/// average value of B is equal to the average value of C, and B and C are
/// both non-empty.
///
/// Example :
/// Input:
/// [1,2,3,4,5,6,7,8]
/// Output: true
/// Explanation: We can split the array into [1,4,5,8] and [2,3,6,7], and
/// both of them have the average of 4.5.
/// Note:
///
/// 1. The length of A will be in the range [1, 30].
/// 2. A[i] will be in the range of [0, 10000].
/// </summary>
bool LeetCode::splitArraySameAverageII(vector<int>& A)
{
	unordered_map<int, int> sum_map;
	int sum = 0;
	sum_map[0] = 1;
	for (int num : A)
	{
		unordered_map<int, int> next_sum = sum_map;
		for (auto itr : sum_map)
		{
			next_sum[itr.first + num] |= itr.second << 1;
		}
		sum_map = next_sum;
		sum += num;
	}

	for (size_t i = 1; i < A.size(); i++)
	{
		if ((sum * i % A.size() == 0) && (sum_map[sum* i / A.size()] &
(1 << i)))
		{
			return true;
		}
	}
	return false;
}
```

Stack and Queue

```
/// <summary>
/// Leet code #402. Remove K Digits
///
/// Given a non-negative integer num represented as a string, remove k
digits
/// from the number so that the new number is the smallest possible.
/// Note:
/// The length of num is less than 10002 and will be ≥ k.
/// The given num does not contain any leading zero.
/// Example 1:
/// Input: num = "1432219", k = 3
/// Output: "1219"
/// Explanation: Remove the three digits 4, 3, and 2 to form the new number
/// 1219 which is the smallest.
///
/// Example 2:
/// Input: num = "10200", k = 1
/// Output: "200"
/// Explanation: Remove the leading 1 and the number is 200. Note that the
/// output must not contain leading zeroes.
///
/// Example 3:
/// Input: num = "10", k = 2
/// Output: "0"
/// Explanation: Remove all the digits from the number and it is left with
/// nothing which is 0.
/// </summary>
string LeetCode::removeKdigits(string num, int k)
{
    string result;
    // pop up big leading digits from front
    for (size_t i = 0; i < num.size(); i++)
    {
        while (!result.empty() && (num[i] < result.back()) && k > 0)
        {
            result.pop_back();
            k--;
        }
        result.push_back(num[i]);
    }
    // pop up extra digits from end
    while (!result.empty() && k > 0)
    {
        result.pop_back();
        k--;
    }
    size_t i = 0;
    // pop up leading 0
    while (i < result.size() && result[i] == '0') i++;
    result = result.substr(i);
    if (result.empty()) result = "0";
    return result;
}

/// <summary>
```

```
/// Leet code #316. Remove Duplicate Letters
///
/// Given a string which contains only lowercase letters, remove duplicate
/// letters so that every letter appear once and only once. You must make
sure
/// your result is the smallest in lexicographical order among all possible
/// results.
///
/// Example:
///
/// Given "bcabc"
/// Return "abc"
///
/// Given "cbacdcbc"
/// Return "acdb"
/// </summary>
string LeetCode::removeDuplicateLetters(string s)
{
    unordered_map<char, int> letter_map;
    unordered_set<char> result_set;
    string result;
    for (size_t i = 0; i < s.size(); i++)
    {
        letter_map[s[i]]++;
    }
    size_t i = 0;
    while(i < s.size())
    {
        if (result.empty())
        {
            result.push_back(s[i]);
            result_set.insert(s[i]);
            i++;
        }
        else
        {
            if (result_set.count(s[i]) > 0)
            {
                letter_map[s[i]]--;
                i++;
            }
            else
            {
                char last_ch = result.back();
                // last character is greater than next character and it is
also duplicated, we throw it away
                if ((last_ch > s[i]) && (letter_map[last_ch] > 1))
                {
                    result.pop_back();
                    letter_map[last_ch]--;
                    result_set.erase(last_ch);
                }
                else
                {
                    result.push_back(s[i]);
                    result_set.insert(s[i]);
                    i++;
                }
```

```
            }
        }
    }
    return result;
}

/// <summary>
/// Leet code #84. Largest Rectangle in Histogram
/// Given n non-negative integers representing the histogram's bar height
/// where the width of each bar is 1,
/// find the area of largest rectangle in the histogram.
/// Above is a histogram where width of each bar is 1, given
/// height = [2,1,5,6,2,3].
/// The largest rectangle is shown in the shaded area, which has area = 10
unit.
/// For example,
/// Given heights = [2,1,5,6,2,3],
/// return 10.
/// </summary>
int LeetCode::largestRectangleAreaByStack(vector<int>& heights)
{
    int max_area = 0;

    stack<pair<int, int>> height_stack;
    for (size_t i = 0; i <= heights.size(); i++)
    {
        int height = (i == heights.size()) ? 0 : heights[i];
        if (height_stack.empty() || (height >= height_stack.top().second))
        {
            height_stack.push(make_pair(i, height));
        }
        else
        {
            int end = height_stack.top().first;
            pair<int, int> pair;
            while ((!height_stack.empty()) && (height <
height_stack.top().second))
            {
                pair = height_stack.top();
                height_stack.pop();
                max_area = max(max_area, (end - pair.first + 1) *
pair.second);
            }
            height_stack.push(make_pair(pair.first, height));
            height_stack.push(make_pair(i, height));
        }
    }
    return max_area;
}

/// <summary>
/// Leet code #239. Sliding Window Maximum
/// Given an array nums, there is a sliding window of size k which is
moving
/// from the very left of the array to the very right.
/// You can only see the k numbers in the window. Each time the sliding
window
/// moves right by one position.
```

```
/// For example,
/// Given nums = [1,3,-1,-3,5,3,6,7], and k = 3.
/// Window position                Max
/// ---------------               -----
/// [1  3  -1] -3  5  3  6  7       3
///  1 [3  -1  -3] 5  3  6  7       3
///  1  3 [-1  -3  5] 3  6  7       5
///  1  3  -1 [-3  5  3] 6  7       5
///  1  3  -1  -3 [5  3  6] 7       6
///  1  3  -1  -3  5 [3  6  7]      7
/// Therefore, return the max sliding window as [3,3,5,5,6,7].
/// Note:
/// You may assume k is always valid, ie: 1 ≤ k ≤ input array's size for
/// non-empty array.
/// Follow up:
/// Could you solve it in linear time?
/// Hint:
/// 1.How about using a data structure such as deque (double-ended queue)?
/// 2.The queue size need not be the same as the window’s size.
/// 3.Remove redundant elements and the queue should store only elements
///   that need to be considered.
/// </summary>
vector<int> LeetCode::maxSlidingWindow(vector<int>& nums, int k)
{
    vector<int> result;
    deque<int> max_window;
    for (size_t i = 0; i < nums.size(); i++)
    {
        if (max_window.empty())
        {
            max_window.push_back(nums[i]);
        }
        else
        {
            if (max_window.size() == k) max_window.pop_front();
            size_t count = 0;
            while (!max_window.empty() && max_window.back() < nums[i])
            {
                max_window.pop_back();
                count++;
            }
            for (size_t j = 0; j < count; j++)
            {
                max_window.push_back(nums[i]);
            }
            max_window.push_back(nums[i]);
        }
        if (max_window.size() == k)
        {
            result.push_back(max_window.front());
        }
    }
    return result;
}

/// <summary>
/// Leet code #295. Find Median from Data Stream
```

```
/// Median is the middle value in an ordered integer list. If the size of
the
/// list is even,
/// there is no middle value. So the median is the mean of the two middle
value.
/// Examples:
///
/// [2,3,4] , the median is 3
///
/// [2,3], the median is (2 + 3) / 2 = 2.5
///
/// Design a data structure that supports the following two operations:
/// void addNum(int num) - Add a integer number from the data stream to the
/// data structure.
/// double findMedian() - Return the median of all elements so far.
///
/// For example:
/// add(1)
/// add(2)
/// findMedian() -> 1.5
/// add(3)
/// findMedian() -> 2
/// Your MedianFinder object will be instantiated and called as such:
/// MedianFinder mf;
/// mf.addNum(1);
/// mf.findMedian();
/// </summary>
class MedianFinder
{
private:
    priority_queue<int, vector<int>, greater<int>> m_Large;
    priority_queue<int> m_Small;
public:
    // Default constructor.
    MedianFinder()
    {
    }

    // Adds a number into the data structure.
    void addNum(int num)
    {
        if ((m_Small.size() == 0) || (m_Small.top() > num))
        {
            m_Small.push(num);
        }
        else
        {
            m_Large.push(num);
        }
        if (m_Small.size() > m_Large.size() + 1)
        {
            m_Large.push(m_Small.top());
            m_Small.pop();
        }
        if (m_Large.size() > m_Small.size())
        {
            m_Small.push(m_Large.top());
            m_Large.pop();
```

```
        }
    }

    // Returns the median of current data stream
    double findMedian()
    {
        double value;
        if (m_Small.size() == m_Large.size() + 1)
        {
            value = (double)m_Small.top();
        }
        else
        {
            value = ((double)m_Small.top() + (double)m_Large.top()) / 2;
        }
        return value;
    }
};


/// <summary>
/// Leet code #480. Sliding Window Median
///
/// Median is the middle value in an ordered integer list. If the size of
/// the list is even, there is no middle value. So the median is the mean
/// of the two middle value.
/// Examples:
/// [2,3,4] , the median is 3
/// [2,3], the median is (2 + 3) / 2 = 2.5
/// Given an array nums, there is a sliding window of size k which is
/// moving from the very left of the array to the very right. You can
/// only see the k numbers in the window. Each time the sliding window
/// moves right by one position. Your job is to output the median array
/// for each window in the original array.
///
/// For example,
/// Given nums = [1,3,-1,-3,5,3,6,7], and k = 3.
/// Window position                Median
/// ---------------               -----
/// [1  3  -1] -3  5  3  6  7       1
/// 1 [3  -1  -3] 5  3  6  7       -1
/// 1  3 [-1  -3  5] 3  6  7       -1
/// 1  3  -1 [-3  5  3] 6  7       3
/// 1  3  -1  -3 [5  3  6] 7       5
/// 1  3  -1  -3  5 [3  6  7]      6
///
/// Therefore, return the median sliding window as [1,-1,-1,3,5,6].
/// Note:
/// You may assume k is always valid, ie: 1 ≤ k ≤ input array's size for
non-empty
/// array.
/// </summary>
vector<double> LeetCode::medianSlidingWindow(vector<int>& nums, int k)
{
    vector<double> result;
    map<int, int> low_half;
    map<int, int> high_half;
    int low_count = 0;
```

```
    int high_count = 0;
    for (size_t i = 0; i < nums.size(); i++)
    {
        // remove the first one
        if (i >= (size_t)k)
        {
            int value = nums[i - k];
            // remove a low half value
            if ((high_half.empty()) || (value < high_half.begin()->first))
            {
                low_half[value]--;
                if (low_half[value] == 0) low_half.erase(value);
                low_count--;
            }
            // remove a high half value
            else
            {
                high_half[value]--;
                if (high_half[value] == 0) high_half.erase(value);
                high_count--;
            }
        }

        // push the new one
        int value = nums[i];
        if ((high_half.empty()) || (value < high_half.begin()->first))
        {
            low_half[value]++;
            low_count++;
        }
        else
        {
            high_half[value]++;
            high_count++;
        }

        if (low_count < high_count)
        {
            low_half[high_half.begin()->first] = high_half.begin()->second;
            low_count += high_half.begin()->second;
            high_count -= high_half.begin()->second;
            high_half.erase(high_half.begin()->first);
        }
        else if ((!low_half.empty()) && (high_count + 2 *
low_half.rbegin()->second <= low_count))
        {
            high_half[low_half.rbegin()->first] = low_half.rbegin()-
>second;
            low_count -= low_half.rbegin()->second;
            high_count += low_half.rbegin()->second;
            low_half.erase(low_half.rbegin()->first);
        }

        if (low_count + high_count == k)
        {
            if (low_count == high_count)
            {
```

```
                result.push_back(((double)low_half.rbegin()->first +
(double)high_half.begin()->first) / 2);
            }
            else
            {
                result.push_back(low_half.rbegin()->first);
            }
        }
    }
    return result;
}
```

String

In this section, we will discuss some string specific solutions.

```
/// <summary>
/// Leet code #392. Is Subsequence
/// Given a string s and a string t, check if s is subsequence of t.
/// You may assume that there is only lower case English letters in both s
/// and t.
/// t is potentially a very long (length ~= 500,000) string, and s is a
short
/// string (<=100).
///
/// A subsequence of a string is a new string which is formed from the
original
/// string by deleting some (can be none) of the characters without
disturbing
/// the relative positions of the remaining characters. (ie, "ace" is a
/// subsequence of "abcde" while "aec" is not).
///
/// Example 1:
/// s = "abc", t = "ahbgdc"
/// Return true.
///
/// Example 2:
/// s = "axc", t = "ahbgdc"
/// Return false.
///
/// Follow up:
/// If there are lots of incoming S, say S1, S2, ... , Sk where k >= 1B,
and
/// you want to check one by one to see if T has its subsequence. In this
/// scenario, how would you change your code?
/// </summary>
bool LeetCode::isSubsequence(string s, string t)
{
    size_t i = 0, j = 0;
    while (i < s.size() && j < t.size())
    {
        if (s[i] == t[j])
        {
            i++; j++;
        }
        else
        {
            j++;
        }
    }
    if (i == s.size()) return true;
    else return false;
}


/// <summary>
/// Leet code #467. Unique Substrings in Wraparound String
///
/// Consider the string s to be the infinite wraparound string of
```

```
/// "abcdefghijklmnopqrstuvwxyz",
/// so s will look like this:
/// "...zabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcd....".
/// Now we have another string p. Your job is to find out how many unique
/// non-empty substrings of p are present in s. In particular, your input
is
/// the string p and you need to output the number of different non-empty
/// substrings of p in the string s.
/// Note: p consists of only lowercase English letters and the size of p
might
/// be over 10000.
///
/// Example 1:
/// Input: "a"
/// Output: 1
/// Explanation: Only the substring "a" of string "a" is in the string.
///
/// Example 2:
/// Input: "cac"
/// Output: 2
/// Explanation: There are two substrings "a", "c" of string "cac" in the
/// string s.
///
/// Example 3:
/// Input: "zab"
/// Output: 6
/// Explanation: There are six substrings "z", "a", "b", "za", "ab", "zab"
/// of string "zab" in the string s.
/// </summary>
int LeetCode::findSubstringInWraproundString(string p)
{
    unordered_map<char, int> str_map;
    int first = 0, last = first + 1;
    while (first < (int)p.size())
    {
        char ch = p[last - 1];
        str_map[ch] = max(str_map[ch], last - first);
        if ((last < (int)p.size()) && ((p[last - 1] - 'a' + 1) % 26 ==
(p[last] - 'a')))
        {
            last++;
        }
        else
        {
            first = last;
            last = first + 1;
        }
    }
    int count = 0;
    for (auto itr : str_map)
    {
        count += itr.second;
    }
    return count;
}
```

Trie Tree

The first one obviously is Trie Tree. The trie tree wiki page is here: https://en.wikipedia.org/wiki/Trie

The leetcode problem and code is as below. Basically trie can help you to look up a word which match the given character sequence quickly, and filter out the ones unqualified quickly.

Please notice that the following code may look like violate rule of encapsulation, but it is easy to implement the recursive functions.

```
/// <summary>
/// Leet code #208. Implement Trie(Prefix Tree)
/// Implement a trie with insert, search, and startsWith methods.
/// You may assume that all inputs are consist of lowercase letters a - z.
/// </summary>
struct TrieNode
{
    string word;
    int count = 0;
    map<const char, TrieNode *> char_map;

    /// <summary>
    /// Constructor of an empty TrieNode
    /// </summary>
    /// <returns></returns>
    TrieNode() {};

    /// <summary>
    /// Destructor of an TrieNode
    /// </summary>
    /// <returns></returns>
    ~TrieNode()
    {
        map<char, TrieNode*>::iterator iterator;
        for (iterator = char_map.begin(); iterator != char_map.end(); +
+iterator)
        {
            if (iterator->second != nullptr)
            {
                delete iterator->second;
            }
            char_map[iterator->first] = nullptr;
        }
        char_map.clear();
    }

    /// <summary>
    /// insert word into the TrieTree.
    /// </summary>
    /// <param name="word>The word</param>
    /// <returns></returns>
    void insert(string word, int i)
    {
        count++;
        if (i == word.size())
        {
```

```
            this->word = word;
            return;
        }
        TrieNode * node;
        if (char_map.find(word[i]) == char_map.end())
        {
            node = new TrieNode();
            char_map[word[i]] = node;
        }
        else
        {
            node = char_map[word[i]];
        }
        node->insert(word, i + 1);
    }

    /// <summary>
    /// get all the next level nodes
    /// </summary>
    /// <param name="word>The word</param>
    /// <returns></returns>
    vector<TrieNode *> getNodes()
    {
        vector<TrieNode *> result;
        map<char, TrieNode*>::iterator iterator;
        for (iterator = char_map.begin(); iterator != char_map.end(); +
+iterator)
        {
            if (iterator->second != nullptr)
            {
                result.push_back(iterator->second);
            }
        }
        return result;
    }

    /// <summary>
    /// search a word in the TrieTree.
    /// </summary>
    /// <param name="word">The word</param>
    /// <returns>true, if found</returns>
    bool search(string word, int i)
    {
        if (i == word.size())
        {
            if (this->word == word) return true;
            else return false;
        }
        if (char_map.find(word[i]) == char_map.end())
        {
            return false;
        }
        else
        {
            TrieNode* node = char_map[word[i]];
            return node->search(word, i + 1);
        }
    }
```

```
    /// <summary>
    /// Returns if there is any word in the trie.
    /// that starts with the given prefix.
    /// </summary>
    /// <param name="prefix">The prefix</param>
    bool startsWith(string prefix, int i)
    {
        if (i == prefix.size()) return true;
        if (char_map.find(prefix[i]) == char_map.end())
        {
            return false;
        }
        else
        {
            TrieNode* node = char_map[prefix[i]];
            return node->startsWith(prefix, i + 1);
        }
    }

    /// <summary>
    /// get match words from the specific prefix
    /// </summary>
    void getMatchWords(string prefix, int i, vector<string>& matchWords)
    {
        if (!this->word.empty())
        {
            matchWords.push_back(word);
        }
        if (i == prefix.size())
        {
            for (auto itr : char_map)
            {
                itr.second->getMatchWords(prefix, i, matchWords);
            }
        }
        else
        {
            if (char_map.find(prefix[i]) == char_map.end())
            {
                return;
            }
            else
            {
                TrieNode* node = char_map[prefix[i]];
                node->getMatchWords(prefix, i + 1, matchWords);
            }
        }
    }
};
```

In leet code, we can use Trie Tree in the following problems.

Problem	Level	Key points
208. Implement Trie(Prefix Tree)	M	implement trie tree.
211. Add and Search Word - Data structure design	M	same as above.

336. Palindrome Pairs	H	use trie to match the prefix and check remaining
212. Word Search II	H	trie + back tracking
425. Word Squares	H	trie can filter out unqualified choice.

KMP algorithm

The KMP algorithm is used to find the repeated pattern in the string.

https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm

```
/// <summary>
/// Leet code #28. Implement strStr()
/// Implement strStr().
/// Returns the index of the first occurrence of needle in haystack, or -1
/// if needle is not part of haystack.
/// </summary>
int LeetCode::strStrKmp(string haystack, string needle)
{
    if (haystack.size() < needle.size()) return -1;
    // Step 1: build kmp table
    vector<int> kmp_table;
    kmp_table.push_back(-1);
    kmp_table.push_back(0);
    size_t pos = 2;
    size_t count = 0;
    while (pos < needle.length())
    {
        if (needle[pos - 1] == needle[count])
        {
            count++;
            kmp_table.push_back(count);
            pos++;
        }
        else if (count > 0)
        {
            count = kmp_table[count];
        }
        else
        {
            kmp_table.push_back(0);
            pos++;
        }
    }

    // Step 2: search substring
    pos = 0;
    size_t index = 0;
    while (pos < haystack.length())
    {
        if (index == needle.length())
```

```
        {
            break;
        }
        if (haystack[pos + index] == needle[index])
        {
            index++;
        }
        else if (index == 0)
        {
            pos++;
        }
        else
        {
            pos = pos + index - kmp_table[index];
            index = kmp_table[index];
        }
    }
    if ((pos < haystack.length()) || (needle.length() == 0))
    {
        return pos;
    }
    else
    {
        return -1;
    }
}
```

In the following solution, we use KMP to remember the last repeated substring.

```
/// <summary>
/// Leet code #459. Repeated Substring Pattern
/// Given a non-empty string check if it can be constructed by taking a
/// substring of it
/// and appending multiple copies of the substring together. You may assume
/// the given string
/// consists of lowercase English letters only and its length will not
/// exceed 10000.
///
/// Example 1:
/// Input: "abab"
/// Output: True
///
/// Explanation: It's the substring "ab" twice.
/// Example 2:
/// Input: "aba"
/// Output: False
///
/// Example 3:
/// Input: "abcabcabcabc"
/// Output: True
/// Explanation: It's the substring "abc" four times. (And the substring
/// "abcabc" twice.)
///
/// </summary>
bool LeetCode::repeatedSubstringPattern(string str)
{
```

```
    vector<int> kmp_table(str.size() + 1, 0);

    size_t index = 0, pos = 2, size = str.size();
    while (pos <= size)
    {
        if (str[pos -1] == str[index])
        {
            index++;
            kmp_table[pos] = index;
            pos++;
        }
        else if (index > 0)
        {
            index = kmp_table[index];
            continue;
        }
        else
        {
            kmp_table[pos] = 0;
            pos++;
        }
    }
    int last = kmp_table.back();
    if ((last != 0) && (size % (size - last) == 0))
    {
        return true;
    }
    else
    {
        return false;
    }
}

In the following solution, we use KMP to find the mirror string from
leftmost.
/// <summary>
/// Leet code #214. Shortest Palindrome
///
/// Given a string S, you are allowed to convert it to a palindrome by
/// adding characters in front of it. Find and return the shortest
palindrome
/// you can find by performing this transformation.
///
/// For example:
/// Given "aacecaaa", return "aaacecaaa".
/// Given "abcd", return "dcbabcd".
/// Given an array of n positive integers and a positive integer s,
/// </summary>
string LeetCode::shortestPalindrome(string s)
{
    string result;
    vector<int> kmp_table(s.size());
    int count = 0;
    size_t pos = 2;
    while (pos < s.size())
    {
        if (s[pos - 1] == s[count])
        {
```

```
            count++;
            kmp_table[pos] = count;
            pos++;
        }
        else
        {
            if (count > 0) count = kmp_table[count];
            else pos++;
        }
    }
    int first = 0, last =s.size() - 1;
    while (first < last)
    {
        if (s[first] == s[last])
        {
            first++;
            last--;
        }
        else
        {
            if (first > 0)
            {
                first = kmp_table[first];
            }
            else last--;
        }
    }
    for (int i = s.size() - 1; i > first + last; i--)
    {
        result.push_back(s[i]);
    }
    result.append(s);
    return result;
}
```

String parsing and state machine

When we solve string parsing in interview, there are normally 3 ways.

1. Adhoc, by flags. This is simple and straightforward, but not systematic, you will expect some challenge from the interviewer. And it is also error prone.
2. State machine based, this is the standard way to do it in interview.
3. Regular expression parsing, if you want to show off your programming skill.

Let's look at the following example which will parse a number in the following format

[+/-][0-9][. [0-9]] [e[+/-][0-9]]

For each expression, you may start from a start state, and end by an end state. This is because you may have leading or tailing spaces. And for each token in the expression it is a new state, please watch if the token is option, it

means the previous state have a route to the next state and skip the current state.

For example, the expression above has 7 state (each means a toke), plus the start and end totally 9 states. Now your job is to figure out from each state, given what kind of input, it will enter the next state, if it received an unexpected input, then error out.

```
/// <summary>
/// Leet code #65. Valid Number
/// Validate if a given string is numeric.
/// Some examples:
/// "0" => true
/// " 0.1 " => true
/// "abc" => false
/// "1 a" => false
/// "2e10" => true
/// Note: It is intended for the problem statement to be ambiguous. You
should gather all requirements up front before implementing one.
/// </summary>
bool LeetCode::isValidNumberII(string s)
{
    typedef enum { start, sign, integer, decimal_start, decimal, exp_start,
exp_sign, exp_int, end } number_state;
    number_state state = start;
    for (size_t i = 0; i < s.size(); i++)
    {
        if (state == start)
        {
            if (isspace(s[i])) state = start;
            else if ((s[i] == '+') || (s[i] == '-')) state = sign;
            else if (isdigit(s[i])) state = integer;
            else if (s[i] == '.') state = decimal_start;
            else return false;
        }
        else if (state == sign)
        {
            if (isdigit(s[i])) state = integer;
            else if (s[i] == '.') state = decimal_start;
            else return false;
        }
        else if (state == integer)
        {
            if (isdigit(s[i])) state = integer;
            else if (s[i] == '.') state = decimal;
            else if (s[i] == 'e') state = exp_start;
            else if (isspace(s[i])) state = end;
            else return false;
        }
        else if (state == decimal_start)
        {
            if (isdigit(s[i])) state = decimal;
            else return false;
        }
        else if (state == decimal)
        {
            if (isdigit(s[i])) state = decimal;
```

```
            else if (s[i] == 'e') state = exp_start;
            else if (isspace(s[i])) state = end;
            else return false;
        }
        else if (state == exp_start)
        {
            if ((s[i] == '+') || (s[i] == '-')) state = exp_sign;
            else if (isdigit(s[i])) state = exp_int;
            else return false;
        }
        else if (state == exp_sign)
        {
            if (isdigit(s[i])) state = exp_int;
            else return false;
        }
        else if (state == exp_int)
        {
            if (isdigit(s[i])) state = exp_int;
            else if (isspace(s[i])) state = end;
            else return false;
        }
        else if (state == end)
        {
            if (isspace(s[i])) state = end;
            else return false;
        }
    }
    if ((state == start) || (state == decimal_start) || (state == sign) ||
(state == exp_start) || (state == exp_sign))
    {
        return false;
    }
    else
    {
        return true;
    }
}
```

Divide by delimiter

Another common way to parsing the string is to divide it by delimiter and parse the component respectively. In the following example, we check the IP address by the delimiter either ':' or '.', from here we check the IPv4 or IPv6 format.

```
/// <summary>
/// Check if this is a valid IPv4 address
/// </summary>
bool LeetCode::checkIPv4Address(vector<string> addressList)
{
    if (addressList.size() != 4)
    {
        return false;
    }
    for (size_t i = 0; i < addressList.size(); i++)
    {
        string address = addressList[i];
        if ((address.size() == 0) || (address.size() > 3)) return false;
```

```
        if ((address[0] == '0') && (address.size() > 1)) return false;
        for (char ch : address)
        {
            if (!isdigit(ch)) return false;
        }
        if (atoi(address.c_str()) > 255)
        {
            return false;
        }
    }
    return true;
}

/// <summary>
/// Check if this is a valid IPv6 address
/// </summary>
bool LeetCode::checkIPv6Address(vector<string> addressList)
{
    if (addressList.size() != 8)
    {
        return false;
    }
    for (size_t i = 0; i < addressList.size(); i++)
    {
        string address = addressList[i];
        if ((address.size() == 0) || (address.size() > 4)) return false;
        for (char ch : address)
        {
            if (!isxdigit(ch)) return false;
        }
    }
    return true;
}

/// <summary>
/// Leet code #468. Validate IP Address
/// In this problem, your job to write a function to check whether a input
/// string is a valid IPv4 address or IPv6 address or neither.
/// IPv4 addresses are canonically represented in dot-decimal notation,
which
/// consists of four decimal numbers,
/// each ranging from 0 to 255, separated by dots ("."), e.g.,172.16.254.1;
/// Besides, you need to keep in mind that leading zeros in the IPv4 is
illegal.
/// For example, the address 172.16.254.01 is illegal.
/// IPv6 addresses are represented as eight groups of four hexadecimal
digits,
/// each group representing 16 bits.
/// The groups are separated by colons (":"). For example, the address
/// 2001:0db8:85a3:0000:0000:8a2e:0370:7334 is a legal one.
/// Also, we could omit some leading zeros among four hexadecimal digits
and some
/// low-case characters in the address to upper-case ones,
/// so 2001:db8:85a3:0:0:8A2E:0370:7334 is also a valid IPv6 address(Omit
/// leading zeros and using upper cases).
/// However, we don't replace a consecutive group of zero value with a
single
/// empty group using two consecutive colons (::)
```

```
/// to pursue simplicity. For example, 2001:0db8:85a3::8A2E:0370:7334 is an
/// invalid IPv6 address.
/// Besides, you need to keep in mind that extra leading zeros in the IPv6
/// is also illegal. For example, the address
/// 02001:0db8:85a3:0000:0000:8a2e:0370:7334 is also illegal.
/// Note: You could assume there is no extra space in the test cases and
there may
/// some special characters in the input string.
/// Example 1:
/// Input: "172.16.254.1"
/// Output: "IPv4"
/// Explanation: This is a valid IPv4 address, return "IPv4".
/// Example 2:
/// Input: "2001:0db8:85a3:0:0:8A2E:0370:7334"
/// Output: "IPv6"
/// Explanation: This is a valid IPv6 address, return "IPv6".
/// Example 3:
/// Input: "256.256.256.256"
/// Output: "Neither"
/// Explanation: This is neither a IPv4 address nor a IPv6 address.
/// </summary>
string LeetCode::validIPAddress(string IP)
{
    vector<string> addressList;
    string word, type;
    for (size_t i = 0; i <= IP.size(); i++)
    {
        if (i == IP.size())
        {
            addressList.push_back(word);
            word.clear();
        }
        else if (IP[i] == '.')
        {
            if (type == "")
            {
                type = "IPv4";
            }
            else if (type != "IPv4")
            {
                type = "Neither";
                break;
            }
            addressList.push_back(word);
            word.clear();
        }
        else if (IP[i] == ':')
        {
            if (type == "")
            {
                type = "IPv6";
            }
            else if (type != "IPv6")
            {
                type = "Neither";
                break;
            }
            addressList.push_back(word);
```

```
            word.clear();
        }
        else
        {
            word.push_back(IP[i]);
        }
    }
    if ((type == "IPv4") && (checkIPv4Address(addressList)))
    {
        return type;
    }
    else if ((type == "IPv6") && (checkIPv6Address(addressList)))
    {
        return type;
    }
    else
    {
        return "Neither";
    }
}
```

This is a quite common interview question, we can reverse the string first, the reverse each word.

```
/// <summary>
/// Leet code #186. Reverse Words in a String II
///
/// Given an input string, reverse the string word by word. A word is
defined
/// as a sequence of non-space characters.
/// The input string does not contain leading or trailing spaces and the
words
/// are always separated by a single space.
/// For example,
/// Given s = "the sky is blue",
/// return "blue is sky the".
/// Could you do it in-place without allocating extra space?
/// </summary>
void LeetCode::reverseWordsII(string &s)
{
    std::reverse(s.begin(), s.end());
    // start from begining of the sentence
    size_t begin = 0;
    size_t end = 0;
    while (begin < s.size())
    {
        // either end hit end or hit a space, we got a word
        if ((!isspace(s[begin])) && ((end == s.size()) ||
(isspace(s[end]))))
        {
            std::reverse(s.begin() + begin, s.begin() + end);
            begin = end;
            end = begin + 1;
        }
        else if ((isspace(s[begin])) && (begin < s.size()))
        {
            begin++;
```

```
            end = begin + 1;
        }
        else if ((!isspace(s[end])) && (end < s.size()))
        {
            end++;
        }
    }
}
```

Count the character occurrence is one of the common pattern in string problem

```
/// <summary>
/// Leet code #76. Minimum Window Substring
/// Given a string S and a string T, find the minimum window in S which
will
/// contain all the characters in T in complexity O(n).
/// For example,
/// S = "ADOBECODEBANC"
/// T = "ABC"
/// Minimum window is "BANC".
/// Note:
/// If there is no such window in S that covers all characters in T, return
the
/// empty string "".
/// If there are multiple such windows, you are guaranteed that there will
always
/// be only one unique minimum window in S.
/// </summary>
/// This solution keep the first X position count for each character in the
/// substring, and the calculate the distance.
///
///
string LeetCode::minWindowII(string s, string t)
{
    vector<int> char_map(128, 0);
    for (size_t i = 0; i < t.size(); i++)
    {
        char_map[t[i]]++;
    }

    pair<int, int> min_window = make_pair(-1, -1);
    int begin = 0, count = t.size();

    for (size_t end = 0; end < s.size(); end++)
    {
        if (char_map[s[end]] > 0)
        {
            count--;
        }
        char_map[s[end]]--;
        // Do we have all the characters matched
        if (count > 0) continue;

        // recover the character count, until break the condition
        while (count == 0)
```

```
        {
            char_map[s[begin]]++;
            if (char_map[s[begin]] > 0)
            {
                count++;
                if (((min_window.first == -1) && (min_window.second == -1))
||
                    (min_window.second - min_window.first > (int)end -
begin))
                {
                    min_window.first = begin;
                    min_window.second = end;
                }
            }
            begin++;
        }
    }

    if ((min_window.first == -1) && (min_window.second == -1))
    {
        return "";
    }
    else
    {
        return s.substr(min_window.first, min_window.second -
min_window.first + 1);
    }
}
```

The following problem demonstrate another way to count strings.

```
/// <summary>
/// Leet code #527. Word Abbreviation
///
/// Given an array of n distinct non-empty strings, you need to
/// generate minimal possible abbreviations for every word
/// following rules below.
/// Begin with the first character and then the number of
/// characters abbreviated, which followed by the last character.
/// If there are any conflict, that is more than one words share
/// the same abbreviation, a longer prefix is used instead of
/// only the first character until making the map from word to
/// abbreviation become unique. In other words, a final
/// abbreviation cannot map to more than one original words.
/// If the abbreviation doesn't make the word shorter, then
/// keep it as original.
/// Example:
/// Input: ["like", "god", "internal", "me", "internet",
///         "interval", "intension", "face", "intrusion"]
/// Output: ["l2e","god","internal","me","i6t","interval",
///          "inte4n","f2e","intr4n"]
/// Note:
/// Both n and the length of each word will not exceed 400.
/// The length of each word is greater than 1.
/// The words consist of lowercase English letters only.
/// The return answers should be in the same order as the
/// original array.
```

```
/// </summary>
vector<string> LeetCode::wordsAbbreviation(vector<string>& dict)
{
    vector<string> result;
    vector<int> prefix;
    unordered_map<string, vector<int>> dup_set;
    for (string s : dict)
    {
        string abbr = makeAbbreviation(s, 1);
        dup_set[abbr].push_back(result.size());
        result.push_back(abbr);
        prefix.push_back(1);
    }
    for (size_t i = 0; i < result.size(); i++)
    {
        while (dup_set[result[i]].size() > 1)
        {
            string abbr = result[i];
            for (int k : dup_set[abbr])
            {
                prefix[k]++;
                result[k] = makeAbbreviation(dict[k], prefix[k]);
                dup_set[result[k]].push_back(k);
            }
            dup_set.erase(abbr);
        }
    }

    return result;
}
```

Array

Array is not a problem type, it is a scenario when the problem is in the context of array and it is not able to be attributed to other categories, so we put it under array.

The array problem normally involves the following topics:

1. Calculate the index or position in the array.
2. Calculate the aggregated value for a range.
3. Find the item by using cells in the array.

The following problem illustrate how to easily iterate a two-dimension array by using the variables to store the value properly.

```
/// <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;
}
```

The following problem tells you how to calculate the surplus in a cycle.

```
/// <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;
    }
}
```

The follow examples present how to calculate and store the temporary result in an array.

```
/// <summary>
/// Leet code #238. Product of Array Except Self
/// Given an array of n integers where n > 1, nums, return an array output such
/// that output[i] is equal to the product of all the elements of nums except
/// nums[i].
///
/// Solve it without division and in O(n).
/// For example, given [1,2,3,4], return [24,12,8,6].
///
/// Follow up:
/// Could you solve it with constant space complexity?
/// (Note: The output array does not count as extra space for the purpose of
/// space complexity analysis.)
/// </summary>
vector<int> LeetCode::productExceptSelf(vector<int>& nums)
{
    vector<int> result(nums.size());
    int product = 1;
    for (size_t i = 0; i < nums.size(); i++)
    {
        result[i] = product;
        product = product * nums[i];
    }
    product = 1;
    for (int i = nums.size() - 1; i >= 0; i--)
    {
        result[i] = result[i] * product;
        product *= nums[i];
    }
    return result;
}


/// <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;
}
```

The following two exmaples demonstrate how to find a duplicate item or a missing item by using the array space.

```
/// <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;
}
```

The idea of the following problem is to put number x in the index of -1

```
/// <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;
    }
```

Hash table

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. check 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

Basic Class

```
/// <summary>
/// Leet code #532. K-diff Pairs in an Array
///
/// Given an array of integers and an integer k, you need to find the
number
/// of unique k-diff pairs in the array. Here a k-diff pair is defined as
an
/// integer pair (i, j), where i and j are both numbers in the array and
their
/// absolute difference is k.
///
/// Example 1:
/// Input: [3, 1, 4, 1, 5], k = 2
/// Output: 2
/// Explanation: There are two 2-diff pairs in the array, (1, 3) and (3,
5).
/// Although we have two 1s in the input, we should only return the number
of
/// unique pairs.
/// Example 2:
/// Input:[1, 2, 3, 4, 5], k = 1
/// Output: 4
/// Explanation: There are four 1-diff pairs in the array, (1, 2), (2, 3),
/// (3, 4) and (4, 5).
/// Example 3:
/// Input: [1, 3, 1, 5, 4], k = 0
/// Output: 1
/// Explanation: There is one 0-diff pair in the array, (1, 1).
/// Note:
/// The pairs (i, j) and (j, i) count as the same pair.
/// The length of the array won't exceed 10,000.
/// All the integers in the given input belong to the range: [-1e7, 1e7].
/// </summary>
int LeetCode::findPairs(vector<int>& nums, int k)
{
    if (k < 0) return 0;
    int count = 0;
    unordered_map<int, int> num_map;
    for (size_t i = 0; i < nums.size(); i++)
    {
        num_map[nums[i]]++;
```

```
    }

    for (auto itr : num_map)
    {
        if (k == 0)
        {
            if (itr.second > 1) count++;
        }
        else
        {
            if (num_map.count(itr.first + k) > 0) count++;
        }
    }
    return count;
}

/// <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;
}
```

```
/// <summary>
/// Leet code #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:
/// 1.No scary math, just apply elementary math knowledge. Still remember
///   how to perform a long division?
/// 2.Try a long division on 4/9, the repeating part is obvious. Now try
///   4/333. Do you see a pattern?
/// 3.Be wary of edge cases! List out as many test cases as you can think
///   of and test your code thoroughly.
/// </summary>
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;
}
```

Advanced Class

```
/// <summary>
/// Leet code #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?
/// </summary>
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;
}

/// <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;
}
```

Binary Search

Binary search may be one of the biggest trap in the interview question, I call it as a trap, because first it is easy to write but also easy to have bug, second in many cases you may not realize the problem can be resolved by binary search, third binary search solution may be coupled with other irregular process patterns which are not obvious.

Standard binary algorithm

First let's look at the standard binary algorithm.

```
/// <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;
}

/// <summary>
/// Leet code #162. Find Peak Element
```

```
/// A peak element is an element that is greater than its neighbors.
/// Given an input array where num[i] ≠ num[i+1], find a peak element
/// and return its index.
/// The array may contain multiple peaks, in that case return the index
/// to any one of the peaks is fine.
///
/// You may imagine that num[-1] = num[n] = INT_MIN.
/// For example, in array [1, 2, 3, 1], 3 is a peak element and your
function
/// should return the index number 2.
///
/// click to show spoilers.
/// Note:
/// Your solution should be in logarithmic complexity.
/// </summary>
int LeetCode::findPeakElement(vector<int>& nums)
{
    int first = 0;
    int last = nums.size() - 1;
    int middle = last;
    while (first < last)
    {
        middle = first + (last - first) / 2;
        int middle_next = middle + 1;
        if (nums[middle_next] < nums[middle])
        {
            last = middle;
        }
        else
        {
            first = middle_next;
            middle = middle_next;
        }
    }
    return last;
}


/// <summary>
/// Leet code #240. Search a 2D Matrix II
/// Write an efficient algorithm that searches for a value in an m x n
matrix.
/// This matrix has the following properties:
/// Integers in each row are sorted in ascending from left to right.
/// Integers in each column are sorted in ascending from top to bottom.
///
/// For example,
/// Consider the following matrix:
/// [
///   [1,   4,  7, 11, 15],
///   [2,   5,  8, 12, 19],
///   [3,   6,  9, 16, 22],
///   [10, 13, 14, 17, 24],
///   [18, 21, 23, 26, 30]
/// ]
///
/// Given target = 5, return true.
/// Given target = 20, return false.
```

```
/// </summary>
bool LeetCode::searchMatrixII(vector<vector<int>>& matrix, int target)
{
    if ((matrix.size() == 0) || (matrix[0].size() == 0))
    {
        return false;
    }
    // starting from the end of first row
    int row = 0;
    int col = matrix[0].size() - 1;
    while ((row < (int)matrix.size()) && (col >= 0))
    {
        // if target is greater than the specified position, exclude the
row
        if (matrix[row][col] < target)
        {
            row++;
        }
        // if target is greater than the specified position, exclude the
column
        else if (matrix[row][col] > target)
        {
            col--;
        }
        else
        {
            return true;
        }
    }
    return false;
}


/// <summary>
/// Leet code # 33. Search in Rotated Sorted Array
///
/// Suppose a sorted array is rotated at some pivot unknown to you
beforehand.
/// (i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).
/// You are given a target value to search. If found in the array return
its index, otherwise return -1.
/// You may assume no duplicate exists in the array.
/// </summary>
int LeetCode::binarySearch(vector<int>& nums, int target)
{
    int first = 0, last = nums.size() - 1;
    while (first <= last)
    {
        size_t middle = (first + last) / 2;
        if (target == nums[middle])
        {
            return middle;
        }
        // because a range can be one item, so = is important.
        else if (nums[first] <= nums[middle])
        {
            // if the target is within ordered range
            if ((target >= nums[first]) && (target < nums[middle]))
```

```
            {
                last = middle - 1;
            }
            else // if not in ordered range then in another range.
            {
                first = middle + 1;
            }
        }
        else
        {
            if ((target > nums[middle]) && (target <= nums[last]))
            {
                first = middle + 1;
            }
            else
            {
                last = middle - 1;
            }
        }
    }
    return -1;
}

/// <summary>
/// Leet code #154. Find Minimum in Rotated Sorted Array II
///
/// Follow up for "Find Minimum in Rotated Sorted Array":
/// What if duplicates are allowed?
/// Would this affect the run-time complexity? How and why?
/// Suppose a sorted array is rotated at some pivot unknown to you
beforehand.
/// (i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).
/// Find the minimum element.
/// The array may contain duplicates.
/// </summary>
int LeetCode::findMinII(vector<int>& nums)
{
    if (nums.size() == 0) return 0;
    int first = 0;
    int last = nums.size() - 1;
    int middle = first;
    while (first < last)
    {
        middle = first + (last - first) / 2;
        if (nums[first] > nums[middle])
        {
            last = middle;
        }
        else if (nums[middle] > nums[last])
        {
            first = middle + 1;
            middle++;
        }
        else if (nums[middle] <nums[last])
        {
            last = middle;
        }
        else if (nums[middle] == nums[last])
```

```
        {
            last--;
        }
    }
    return nums[first];
}
```

The following problem is use binary search idea to find the kth item in two arrays.

```
/// <summary>
/// LeetCode #4. Median of Two Sorted Arrays
/// There are two sorted arrays nums1 and nums2 of size m and n
respectively.
/// Find the median of the two sorted arrays.
/// The overall run time complexity should be O(log(m + n)).
///
/// Example 1:
/// num1 = [1, 3]
/// nums2 = [2]
/// The median is 2.0
///
/// Example 2:
/// nums1 = [1, 2]
/// nums2 = [3, 4]
/// The median is (2 + 3)/2 = 2.5
/// </summary>
double LeetCode::findMedianSortedArrays(vector<int>& nums1, vector<int>&
nums2)
{
    double value;
    // transfer the problem to find the kth item (middle point) in two
arrays
    size_t size = nums1.size() + nums2.size();
    if (size % 2 == 1)
    {
        value = findKthNumber(nums1, nums2, 0, 0, size / 2 + 1);
    }
    else
    {
        value = findKthNumber(nums1, nums2, 0, 0, size / 2) +
findKthNumber(nums1, nums2, 0, 0, size / 2 + 1);
        value = value / 2;
    }
    return value;
}

/// <summary>
/// Find kth number in the two sorted array, it only do one iteration,
/// and if return true
/// it is found, the number of total numbers must be greater than the k
/// </summary>
double LeetCode::findKthNumber(vector<int> &nums1, vector<int> &nums2,
size_t s1, size_t s2, size_t k)
{
    // Basically each array should contribute k /2 items, and compare the
    // last value, and throw away the smaller part.
```

```
    double value;
    // if start position of nums1 greater or equal to its size,
    // then no more contribution from nums1
    if (s1 >= nums1.size())
    {
        value = nums2[s2 + k - 1];
    }
    else if (s2 >= nums2.size())
    {
        value = nums1[s1 + k - 1];
    }
    // k == 1 is a special case
    else if (k == 1)
    {
        value = min(nums1[s1], nums2[s2]);
    }
    else
    {
        // l is the contri
        size_t l = min(nums1.size() - s1, k / 2);
        size_t r = min(nums2.size() - s2, k / 2);
        if (nums1[s1 + l - 1] < nums2[s2 + r - 1])
        {
            s1 += l;
            k = k - l;
        }
        else
        {
            s2 += r;
            k = k - r;
        }
        value = findKthNumber(nums1, nums2, s1, s2, k);
    }
    return value;
}
```

Greedy

The greedy algorithm is to search an ordered array collect the non-conflicting (with the remaining) items as many as possible and group them. After all the items in the array are scanned, grouped and processed the problem is resolved.

The greedy algorithm is often coupled with schedule of intervals, in this case you should throw the interval into an auto sorted data structure, such as priority queue or ordered map (set).

The first solution is not optimal, we should sort by end time, not start time.

```
/// <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::minMeetingRooms(vector<Interval>& intervals)
{
    struct IntervalCompare
    {
        bool operator() (Interval &a, Interval &b)
        {
            if (a.start < b.start) return true;
            else if ((a.start == b.start) && (a.end < b.end)) return true;
            else return false;
        }
    };
    sort(intervals.begin(), intervals.end(), IntervalCompare());
    vector<int> ends;
    int max_rooms = 0;
    for (size_t i = 0; i < intervals.size(); i++)
    {
        int count = 1;
        for (size_t j = 0; j < i; j++)
        {
            if (intervals[i].start < intervals[j].end)
            {
                count++;
            }
        }
        max_rooms = max(max_rooms, count);
    }
    return max_rooms;
}

/// <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;
}
```

In some scenarios, there are no explicit interval, we need to make it out, think the range in an array is an interval.

```
/// <summary>
/// Leet code #330. Patching Array
///
/// Given a sorted positive integer array nums and an integer n, add/patch
/// elements to the array such that any number in range [1, n] inclusive
/// can be formed by the sum of some elements in the array.
/// Return the minimum number of patches required.
///
/// Example 1:
/// nums = [1, 3], n = 6
/// Return 1.
/// Combinations of nums are [1], [3], [1,3], which form possible sums of:
/// 1, 3, 4.
/// Now if we add/patch 2 to nums, the combinations are: [1], [2], [3],
[1,3],
/// [2,3], [1,2,3].
/// Possible sums are 1, 2, 3, 4, 5, 6, which now covers the range [1, 6].
/// So we only need 1 patch.
///
/// Example 2:
/// nums = [1, 5, 10], n = 20
/// Return 2.
/// The two patches can be [2, 4].
///
/// Example 3:
/// nums = [1, 2, 2], n = 5
/// Return 0.
/// </summary>
int LeetCode::minPatches(vector<int>& nums, int n)
{
    int result = 0;
    unsigned long long sum = 0;
    size_t index = 0;
    while (sum < (unsigned long long)n)
    {
        if (index < nums.size() && (unsigned long long) nums[index] <= sum
+ 1)
        {
            sum += nums[index];
            index++;
        }
        else
        {
            result++;
            sum += sum + 1;
        }
    }
    return result;
}

/// <summary>
/// Leet code #218. The Skyline Problem
///
/// A city's skyline is the outer contour of the silhouette formed
```

```
/// by all the buildings in that city when viewed from a distance.
/// Now suppose you are given the locations and height of all the buildings
/// as shown on a cityscape photo (Figure A), write a program to output
/// the skyline formed by these buildings collectively (Figure B).
///
/// Buildings   Skyline Contour
/// The geometric information of each building is represented by a triplet
of
/// integers [Li, Ri, Hi], where Li and Ri are the x coordinates of the
left
/// and right edge of the ith building, respectively, and Hi is its height.
/// It is guaranteed that 0 ≤ Li, Ri ≤ INT_MAX, 0 < Hi ≤ INT_MAX, and Ri -
Li > 0.
/// You may assume all buildings are perfect rectangles grounded on an
absolutely
/// flat surface at height 0.
///
/// For instance, the dimensions of all buildings in Figure A are recorded
as:
/// [ [2 9 10], [3 7 15], [5 12 12], [15 20 10], [19 24 8] ] .
/// The output is a list of "key points" (red dots in Figure B) in the
format of
/// [ [x1,y1], [x2, y2], [x3, y3], ... ] that uniquely defines a skyline. A
key point
/// is the left endpoint of a horizontal line segment. Note that the last
key point,
/// where the rightmost building ends, is merely used to mark the
termination of the
/// skyline, and always has zero height. Also, the ground in between any
two adjacent
/// buildings should be considered part of the skyline contour.
/// For instance, the skyline in Figure B should be represented as:
/// [ [2 10], [3 15], [7 12], [12 0], [15 10], [20 8], [24, 0] ].
/// Notes:
/// The number of buildings in any input list is guaranteed to be in the
range [0, 10000].
/// The input list is already sorted in ascending order by the left x
position Li.
/// The output list must be sorted by the x position.
/// There must be no consecutive horizontal lines of equal height in the
output skyline.
/// For instance, [...[2 3], [4 5], [7 5], [11 5], [12 7]...] is not
acceptable;
/// the three lines of height 5 should be merged into one in the final
output
/// as such: [...[2 3], [4 5], [12 7], ...]
/// </summary>
vector<pair<int, int>> LeetCode::getSkyline(vector<vector<int>>& buildings)
{
    vector<pair<int, int>> result;
    map<int, int> height_map = { { 0, 1 } };
    map<int, unordered_map<int, vector<int>>> edge_map;
    for (size_t i = 0; i < buildings.size(); i++)
    {
        // left side
        edge_map[buildings[i][0]][0].push_back(i);
        // right side
        edge_map[buildings[i][1]][1].push_back(i);
```

```
    }
    for (auto edge : edge_map)
    {
        int pos = edge.first;

        // process building left edges
        for (size_t i : edge.second[0])
        {
            height_map[buildings[i][2]]++;
        }

        // process building right edges
        for (size_t i : edge.second[1])
        {
            height_map[buildings[i][2]]--;
            if (height_map[buildings[i][2]] == 0)
height_map.erase(buildings[i][2]);
        }

        // skyline is heightest building
        int skyline = height_map.rbegin()->first;

        // push to result if not same height
        if (result.empty() || result.back().second != skyline)
        {
            result.push_back(make_pair(pos, skyline));
        }
    }
    return result;
}
```

Another way to do greedy is by Bread first search, starting the one without or least dependency.

```
/// <summary>
/// Leet code # 502. IPO
///
/// Suppose LeetCode will start its IPO soon. In order to sell a good price
/// of its shares to Venture Capital, LeetCode would like to work on some
/// projects to increase its capital before the IPO. Since it has limited
/// resources, it can only finish at most k distinct projects before the
IPO.
/// Help LeetCode design the best way to maximize its total capital after
/// finishing at most k distinct projects.
///
/// You are given several projects. For each project i, it has a pure
profit
/// Pi and a minimum capital of Ci is needed to start the corresponding
project.
/// Initially, you have W capital. When you finish a project, you will
obtain
/// its pure profit and the profit will be added to your total capital.
///
/// To sum up, pick a list of at most k distinct projects from given
projects
/// to maximize your final capital, and output your final maximized
capital.
///
```

```
/// Example 1:
///
/// Input: k=2, W=0, Profits=[1,2,3], Capital=[0,1,1].
///
/// Output: 4
/// Explanation: Since your initial capital is 0, you can only start the
project
/// indexed 0.
/// After finishing it you will obtain profit 1 and your capital becomes 1.
/// With capital 1, you can either start the project indexed 1 or the
project indexed 2.
/// Since you can choose at most 2 projects, you need to finish the project
indexed 2
/// to get the maximum capital.
/// Therefore, output the final maximized capital, which is 0 + 1 + 3 = 4.
/// Note:
/// 1.You may assume all numbers in the input are non-negative integers.
/// 2.The length of Profits array and Capital array will not exceed 50,000.
/// 3.The answer is guaranteed to fit in a 32-bit signed integer.
/// </summary>
int LeetCode::findMaximizedCapital(int k, int W, vector<int>& Profits,
vector<int>& Capital)
{
    priority_queue<pair<int, int>> capital_map;
    priority_queue<int> profit_map;

    for (size_t i = 0; i < Capital.size(); i++)
    {
        capital_map.push(make_pair(-Capital[i], Profits[i]));
    }

    for (int i = 0; i < k; i++)
    {
        while (!capital_map.empty() && -capital_map.top().first <= W)
        {
            profit_map.push(capital_map.top().second);
            capital_map.pop();
        }
        if (profit_map.empty())
        {
            break;
        }
        W += profit_map.top();
        profit_map.pop();
    }
    return W;
}


/// <summary>
/// Leet code #358. Rearrange String k Distance Apart
///
/// Given a non-empty string s and an integer k, rearrange the string such
/// that the same characters are at least distance k from each other.
/// All input strings are given in lowercase letters. If it is not possible
/// to rearrange the string, return an empty string "".
///
/// Example 1:
```

```
/// s = "aabbcc", k = 3
/// Result: "abcabc"
/// The same letters are at least distance 3 from each other.
///
/// Example 2:
/// s = "aaabc", k = 3
/// Answer: ""
/// It is not possible to rearrange the string.
///
/// Example 3:
/// s = "aaadbbcc", k = 2
/// Answer: "abacabcd"
/// Another possible answer is: "abcabcda"
/// The same letters are at least distance 2 from each other.
/// </summary>
string LeetCode::rearrangeString(string s, int k)
{
    vector<pair<int, int>> char_map(26, { 0, -1 });
    string result(s.size(), 0);

    for (char ch : s)
    {
        char_map[ch - 'a'].first++;
    }

    for (size_t i = 0; i < s.size(); i++)
    {
        pair<int, int> max_pos = make_pair(0, -1);
        for (int j = 0; j < 26; j++)
        {
            if (char_map[j].first > max_pos.first && char_map[j].second <=
(int)i)
            {
                max_pos = make_pair(char_map[j].first, j);
            }
        }
        if (max_pos.second == -1) return "";
        result[i] = max_pos.second + 'a';
        char_map[max_pos.second].first--;
        char_map[max_pos.second].second = i + k;
    }
    return result;
}


/// <summary>
/// Leet code #630. Course Schedule III
///
/// There are n different online courses numbered from 1 to n. Each course
/// has some duration(course length) t and closed on dth day. A course
/// should be taken continuously for t days and must be finished before or
/// on the dth day. You will start at the 1st day.
///
/// Given n online courses represented by pairs (t,d), your task is to
/// find the maximal number of courses that can be taken.
/// Example:
/// Input: [[100, 200], [200, 1300], [1000, 1250], [2000, 3200]]
/// Output: 3
```

```
/// Explanation:
/// There're totally 4 courses, but you can take 3 courses at most:
/// First, take the 1st course, it costs 100 days so you will finish it on
/// the 100th day, and ready to take the next course on the 101st day.
/// Second, take the 3rd course, it costs 1000 days so you will finish it
/// on the 1100th day, and ready to take the next course on the 1101st day.
/// Third, take the 2nd course, it costs 200 days so you will finish it on
/// the 1300th day.
/// The 4th course cannot be taken now, since you will finish it on the
/// 3300th day, which exceeds the closed date.
/// Note:
/// The integer 1 <= d, t, n <= 10,000.
/// You can't take two courses simultaneously.
/// </summary>
/// <hint>
/// You need to sort the end time, not the start time, it will make sure
/// that each meeting only is considered when it hasto be.
/// </hint>
int LeetCode::scheduleCourse(vector<vector<int>>& courses)
{
    struct dead_line_compare
    {
        bool operator() (vector<int>& a, vector<int>&b)
        {
            if (a[1] == b[1]) return (a[0] < b[0]);
            else return(a[1] < b[1]);
        }
    };

    sort(courses.begin(), courses.end(), dead_line_compare());

    int time = 0;
    priority_queue<int> course_duration;
    for (size_t i = 0; i < courses.size(); i++)
    {
        pair<int, int> schedule = make_pair(courses[i][0], courses[i][1]);

        // if start time later than now, we are greedy
        if (schedule.second >= time + schedule.first)
        {
            course_duration.push(schedule.first);
            time += schedule.first;
        }
        // if we can not make it, we replace the longest course
        else
        {
            if ((!course_duration.empty()) && (course_duration.top() >
schedule.first))
            {
                time -= (course_duration.top() - schedule.first);
                course_duration.pop();
                course_duration.push(schedule.first);
            }
        }
    }

    return course_duration.size();
}
```

Graph

The graph is normally used to process the items with dependencies, an item can be processed only if all its prerequisites have been processed.

There are 3 types of data structure can be used in graph, linked node, two-dimension array and hash table map. Normally we use the later two.

BFS or DFS

Most of the graph problems are using BFS, but it does not mean we cannot use DFS to resolve the graph problem.

```
/// <summary>
/// Leet code #332. Reconstruct Itinerary
/// </summary>
void LeetCode::findItinerary(vector<string> &path, unordered_map<string,
map<string, int>>& tickets,
    unordered_map<string, map<string, int>>& visited, int &count)
{
    if (count == 0) return;
    string city = path.back();
    for (map<string, int>::iterator itr = tickets[city].begin(); itr !=
tickets[city].end(); itr++)
    {
        // The visited map is to count how many times, the city is visited
        //  which should not go beyond the count from the input
        if (visited[city][itr->first] == itr->second) continue;
        visited[city][itr->first]++;
        path.push_back(itr->first);
        count--;
        findItinerary(path, tickets, visited, count);
        if (count != 0)
        {
            count++;
            visited[city][itr->first]--;
            path.pop_back();
        }
        else
        {
            break;
        }
    }
    return;
}

/// <summary>
/// Leet code #332. Reconstruct Itinerary
///
/// Given a list of airline tickets represented by pairs of departure and
arrival
/// airports [from, to], reconstruct the itinerary in order. All of the
tickets
/// belong to a man who departs from JFK. Thus, the itinerary must begin
with JFK.
///
/// Note:
```

```
/// 1.If there are multiple valid itineraries, you should return the
itinerary
/// that has the smallest lexical order when read as a single string. For
example,
/// the itinerary ["JFK", "LGA"] has a smaller lexical order than ["JFK",
"LGB"].
/// 2.All airports are represented by three capital letters (IATA code).
/// 3.You may assume all tickets form at least one valid itinerary.
///
/// Example 1:
/// tickets = [["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR",
"SFO"]]
/// Return ["JFK", "MUC", "LHR", "SFO", "SJC"].
///
/// Example 2:
/// tickets = [["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],
["ATL","SFO"]]
/// Return ["JFK","ATL","JFK","SFO","ATL","SFO"].
/// Another possible reconstruction is
["JFK","SFO","ATL","JFK","ATL","SFO"].
/// But it is larger in lexical order.
/// </summary>
vector<string> LeetCode::findItinerary(vector<pair<string, string>>
tickets)
{
    vector<string> result = { "JFK" };
    unordered_map<string, map<string, int>> route_map, visited;
    int count = 0;
    for (size_t i = 0; i < tickets.size(); i++)
    {
        route_map[tickets[i].first][tickets[i].second]++;
        count++;
    }
    findItinerary(result, route_map, visited, count);
    return result;
}
```

Best Meeting Point

To find out a best meeting point for multiple people, you need to use the method called weighted distance. Basically, the best meeting point for two persons are the middle point between them, but when you have multiple people and in a 2D grid you should use weighted distance. The weight is the number of people, to find out the fairest meeting point, you can use two pointers, and move from the light side to the heavy.

There is a catch in the weighted distance, in case there is any obstacle in the route, weighted distance may fail. This is why #317 cannot use this method.

```
/// <summary>
/// Leet code #296. Best Meeting Point
/// </summary>
int LeetCode::minTotalDistance(vector<int> & nums)
{
    if (nums.empty()) return 0;
    int i = 0, j = nums.size() - 1;
    int left = nums[i], right = nums[j], sum = 0;
    while (i < j)
```

```
    {
        if (left < right)
        {
            sum += left;
            i++;
            left += nums[i];
        }
        else
        {
            sum += right;
            j--;
            right += nums[j];
        }
    }
    return sum;
}

/// <summary>
/// Leet code #296. Best Meeting Point
///
/// A group of two or more people wants to meet and minimize the total
travel
/// distance. You are given a 2D grid of values 0 or 1, where each 1 marks
/// the home of someone in the group. The distance is calculated using
/// Manhattan Distance, where distance(p1, p2) = |p2.x - p1.x| + |p2.y -
p1.y|.
/// For example, given three people living at (0,0), (0,4), and (2,2):
/// 1 - 0 - 0 - 0 - 1
/// |   |   |   |   |
/// 0 - 0 - 0 - 0 - 0
/// |   |   |   |   |
/// 0 - 0 - 1 - 0 - 0
/// The point (0,2) is an ideal meeting point, as the total travel distance
of 2+2+2=6 is minimal. So return 6.
///
///        Hint:
/// 1.Try to solve it in one dimension first. How can this solution apply
to the two dimension case?
/// </summary>
int LeetCode::minTotalDistance(vector<vector<int>>& grid)
{
    if (grid.empty() || grid[0].empty()) return 0;
    vector<int> row(grid.size());
    vector<int> col(grid[0].size());
    for (size_t i = 0; i < grid.size(); i++)
    {
        for (size_t j = 0; j < grid[i].size(); j++)
        {
            if (grid[i][j] == 1)
            {
                row[i]++;
                col[j]++;
            }
        }
    }
    return minTotalDistance(row) + minTotalDistance(col);
}
```

```
/// <summary>
/// Leet code #902. Truck delivery
///
/// There are a number of houses in a line, the distance between two houses
/// are given in an array.
/// for example the distance between house(0) and house(1) is in
distance[0]
/// and the distance between house(n-2) and house(n-1) is given in
/// distance[n-2]. A delivery truck stops in front of the houses, and the
/// driver has a number of packages to deliver, each with a weight[i],
/// if no package for this house it is 0.
/// The truck can only stop in front of one house, the driver should
minimize
/// the total cost of the delivery.
///
/// Note:
/// 1. The cost is defined by multiply the distance the driver should walk
///     with the weight of package in hand, i.e.
///     It is weight[i] * distance[i].
/// 2. If the driver walk back with empty hand the cost is zero.
/// 3. The driver is only allowed to carry one package at a time.
/// </summary>
int LeetCode::minDeliveryCost(vector<int> distances, vector<int> weight)
{
    // at very beginning, we assume the truck stops at house0, so the
    // weight of total packages on left is 0 and the total weight
    // on right is from 1 to n-1.
    int weight_left = 0, weight_right = 0;
    int distance = 0, sum = 0, min_cost = INT_MAX;
    for (size_t i = 1; i < weight.size(); i++)
    {
        weight_right += weight[i];
        distance += distances[i];
        sum += weight[i] * distance;
    }
    min_cost = sum;
    for (size_t i = 1; i < weight.size(); i++)
    {
        // assume truck is at house i, house[i-1] is moved to left
        weight_left += weight[i - 1];
        // on the right side, the cost to each house deduct the cost of
weight * distance[i-1];
        sum -= weight_right * distances[i];
        // on the left side, the cost to each house add (weight *
distance[i-1]
        sum += weight_left * distances[i];
        // house[i] is no longer on right, it is in front of track
        weight_right -= weight[i];
        // calculate min_cost
        min_cost = min(sum, min_cost);
    }
    return min_cost;
}
```

Find the logic behind and beat 99%

But this is not a typical graph problem, because you can reduce 2 dimension to 1 dimension.

```
/// <summary>
/// Leet code #277. Find the Celebrity
///
/// Suppose you are at a party with n people (labeled from 0 to n - 1) and
/// among them, there may exist one celebrity. The definition of a
celebrity
/// is that all the other n - 1 people know him/her but he/she does not
know
/// any of them.
/// Now you want to find out who the celebrity is or verify that there is
/// not one.
/// The only thing you are allowed to do is to ask questions like:
/// "Hi, A. Do you know B?"
/// to get information of whether A knows B. You need to find out the
/// celebrity (or verify there is not one) by asking as few questions as
/// possible (in the asymptotic sense).
/// You are given a helper function bool knows(a, b) which tells you
whether
/// A knows B.
/// Implement a function int findCelebrity(n), your function should
minimize
/// the number of calls to knows.
/// Note: There will be exactly one celebrity if he/she is in the party.
/// Return the celebrity's label if there is a celebrity in the party.
/// If there is no celebrity, return -1.
/// </summary>
int LeetCode::findCelebrity(int n, vector<vector<bool>>& relation_map)
{
    int first = 0, last = first + 1;
    int temp = -1;
    // the first pass will catch one person he does not know any one behind
him
    while (last < n)
    {
        if (first == last)
        {
            last++;
        }
        else if (relation_map[first][last])
        {
            temp = first;
            first = last;
        }
        else
        {
            last++;
        }
    }
    // the second pass check if anyone in front of him knows him.
    for (int i = 0; i < first; i++)
    {
        if (relation_map[first][i]) return -1;
    }
```

```
    // The third pass check if anyone does not know him
    for (int i = 0; i < n; i++)
    {
        if (i == temp) continue;
        if (!relation_map[i][first]) return -1;
    }
    return first;
}
```

Shortest distance

When ask for shortest distance, the easy way is to Dijkstra's algorithm. We calculate the distance from the starting point and always pick the shortest distance to an unvisted node, but if the distance to a visted node has been modified with a shorter one, the path from that node should be checked again.

```
/// <summary>
/// Leet code #505. The Maze II
/// </summary>
void LeetCode::shortestDistance(vector<vector<int>>& maze,
vector<vector<int>>& visited, vector<int>& start,
    priority_queue<pair<int, vector<int>>> &process_queue)
{
    vector<vector<int>> next_list;
    int distance = visited[start[0]][start[1]];
    for (size_t i = 0; i < 4; i++)
    {
        int step = 0;
        vector<int> pos = start;
        if (i == 0)
        {
            while ((pos[0] > 0) && (maze[pos[0] - 1][pos[1]] == 0))
            {
                step++;
                pos[0]--;
            }
        }
        else if (i == 1)
        {
            while ((pos[0] < (int)maze.size() - 1) && (maze[pos[0] + 1]
[pos[1]] == 0))
            {
                step++;
                pos[0]++;
            }
        }
        else if (i == 2)
        {
            while ((pos[1] > 0) && (maze[pos[0]][pos[1] - 1] == 0))
            {
                step++;
                pos[1]--;
            }
        }
        else
        {
```

```
            while ((pos[1] < (int)maze[0].size() - 1) && (maze[pos[0]]
[pos[1] + 1] == 0))
            {
                step++;
                pos[1]++;
            }
        }
        if (pos == start) continue;
        if (visited[pos[0]][pos[1]] > distance + step)
        {
            visited[pos[0]][pos[1]] = distance + step;
            process_queue.push(make_pair(-visited[pos[0]][pos[1]], pos));
        }
    }
}

/// <summary>
/// Leet code #505. The Maze II
///
/// There is a ball in a maze with empty spaces and walls. The ball can
/// go through empty spaces by rolling up, down, left or right, but it
/// won't stop rolling until hitting a wall. When the ball stops, it could
/// choose the next direction.
///
/// Given the ball's start position, the destination and the maze, find the
/// shortest distance for the ball to stop at the destination. The distance
/// is defined by the number of empty spaces traveled by the ball from the
/// start position (excluded) to the destination (included). If the ball
/// cannot stop at the destination, return -1.
///
/// The maze is represented by a binary 2D array. 1 means the wall and 0
means
/// the empty space. You may assume that the borders of the maze are all
walls.
/// The start and destination coordinates are represented by row and column
/// indexes.
///
/// Example 1
/// Input 1: a maze represented by a 2D array
///
/// 0 0 1 0 0
/// 0 0 0 0 0
/// 0 0 0 1 0
/// 1 1 0 1 1
/// 0 0 0 0 0
/// Input 2: start coordinate (rowStart, colStart) = (0, 4)
/// Input 3: destination coordinate (rowDest, colDest) = (4, 4)
///
/// Output: 12
/// Explanation: One shortest way is :
/// left -> down -> left -> down -> right -> down -> right.
/// The total distance is 1 + 1 + 3 + 1 + 2 + 2 + 2 = 12.
///
/// Example 2
/// Input 1: a maze represented by a 2D array
///
/// 0 0 1 0 0
/// 0 0 0 0 0
```

```
/// 0 0 0 1 0
/// 1 1 0 1 1
/// 0 0 0 0 0
/// Input 2: start coordinate (rowStart, colStart) = (0, 4)
/// Input 3: destination coordinate (rowDest, colDest) = (3, 2)
/// Output: -1
/// Explanation: There is no way for the ball to stop at the destination.
///
/// Note:
/// 1.There is only one ball and one destination in the maze.
/// 2.Both the ball and the destination exist on an empty space,
///   and they will not be at the same position initially.
/// 3.The given maze does not contain border (like the red rectangle in the
///   example
///   pictures), but you could assume the border of the maze are all walls.
/// 4.The maze contains at least 2 empty spaces, and both the width and
height of
///   the maze won't exceed 100.
/// </summary>
int LeetCode::shortestDistance(vector<vector<int>>& maze, vector<int>&
start, vector<int>& destination)
{
    if (maze.empty() || maze[0].empty()) return -1;
    vector<vector<int>> visited(maze.size(), vector<int>(maze[0].size(),
INT_MAX));
    visited[start[0]][start[1]] = 0;
    priority_queue<pair<int, vector<int>>> process_queue;
    process_queue.push(make_pair(0, start));
    while (!process_queue.empty())
    {
        start = process_queue.top().second;
        process_queue.pop();
        if (start == destination) break;
        shortestDistance(maze, visited, start, process_queue);
    }
    int shortest_distance = visited[destination[0]][destination[1]];
    if (shortest_distance == INT_MAX) return -1;
    else return shortest_distance;
}
```

Union find

```
/// <summary>
/// Leet code #200. Number of Islands
/// Given a 2d grid map of '1's (land) and '0's (water), count the number
of
/// islands.
/// An island is surrounded by water and is formed by connecting adjacent
lands
/// horizontally or vertically. You may assume all four edges of the grid
are
/// all surrounded by water.
/// Example 1:
/// 11110
/// 11010
/// 11000
/// 00000
/// Answer : 1
```

```
/// Example 2 :
/// 11000
/// 11000
/// 00100
/// 00011
/// Answer : 3
/// </summary>
int LeetCode::numIslands(vector<vector<char>>& grid)
{
    size_t index = 0;
    vector<vector<int>> map(grid.size());
    unordered_map<int, int> islands;
    // search from top to down, then from left to right
    for (size_t i = 0; i < grid.size(); i++)
    {
        map[i] = vector<int>(grid[i].size());
        for (size_t j = 0; j < grid[i].size(); j++)
        {
            // if this is land
            if (grid[i][j] == '1')
            {
                // if adjacent to left land, use the id of left cell
                if ((i > 0) && (map[i - 1][j] != 0))
                {
                    map[i][j] = map[i - 1][j];
                }
                // if adjacent to up land use the if of up cell
                if ((j > 0) && (map[i][j - 1] != 0))
                {
                    // no left land
                    if (map[i][j] == 0)
                    {
                        map[i][j] = map[i][j - 1];
                    }
                    else if (map[i][j] != map[i][j - 1])
                    {
                        int src = map[i][j];
                        int dest = map[i][j - 1];
                        while (islands[src] != 0) src = islands[src];
                        while (islands[dest] != 0) dest = islands[dest];
                        // swap order is option here, but if we use hash
table
                        // better to keep it small. i.e. large id merge to
                        // small id
                        if (src > dest) swap(src, dest);
                        // must check duplicated to avoid endless loop.
                        if (src != dest) islands[dest] = src;
                    }
                }
                // isolate land, new id
                if (map[i][j] == 0)
                {
                    index++;
                    map[i][j] = index;
                    islands[map[i][j]] = 0;
                }
            }
        }
```

```
    }
    int result = 0;
    for (unordered_map<int, int>::iterator iterator = islands.begin();
iterator != islands.end(); iterator++)
    {
        if (iterator->second == 0)
        {
            result++;
        }
    }
    return result;
}
```

Sorting

Last year my son give me an exam problem from MIT, which give you a trace of sorting progress and ask what sorting mechanism it used. Here are some hints. To answer this question, we need to know what are the characteristics for the partially sort result. Assuming we sort from minimum to maximum.

Sorting Algorithm	Complexity	Worst Case	How the partial result looks like
Selection	O(n^2)	O(n^2)	Always the minimum is in partial result
Insertion	O(n^2)	O(n^2)	Partial result is always sorted, but minimum is not necessary in partial result.
Bubble	O(n^2)	O(n^2)	In every iteration, the larger number is exchange one level up.
Merge sort	O(nlog(n))	O(nlog(n))	You first get a sorted pair of two, then group of 4, then group of 8…
Quick Sort	O(nlog(n))	O(n^2)	The first number in the list will be put in middle, with left part less than it, right part greater than it, slowly you are building a BST.
Heap sort	O(nlog(n))	O(nlog(n))	close to quick sort, but only the left part build up the BST. Please remember if you have a small M groups of sorted group, heap sort can give you a O(n) solution.
Binary Tree sort	O(nlog(n))	O(n^2)	Almost same as quick sort
Bucket sort	O(n)*O(mlog(m))	O(n)*O(mlog(m))	bucket sort is normally used for a large data which should be sorted externally.

In the above list, you need to pay attention to **merge sort, heap sort, tree sort** and **bucket sort**

Bucket Sort

Bucket Sort is to divide the original data into multiple groups, the groups themselves are in order. After the data are in groups you can further do the next level bucket sort or simply select the minimum or maximum data in each group. Please look at the following example:

```
/// <summary>
/// Leet code #164. Maximum Gap
```

```
/// Given an unsorted array, find the maximum difference between the
successive
/// elements in its sorted form.
/// Try to solve it in linear time/space.
/// Return 0 if the array contains less than 2 elements.
/// You may assume all elements in the array are non-negative integers and
fit
/// in the 32-bit signed integer range.
/// </summary>
int LeetCode::maximumGap(vector<int>& nums)
{
    if (nums.size() < 2) return 0;

    int minimum = INT_MAX;
    int maximum = INT_MIN;
    for (size_t i = 0; i < nums.size(); i++)
    {
        if (nums[i] < minimum) minimum = nums[i];
        if (nums[i] > maximum) maximum = nums[i];
    }
    // The plus one is to make sure we do not have a slot_gap as 0.
    int slot_gap = (maximum - minimum) / nums.size() + 1;
    // In case the slot gap is not divisible by the range.
    vector<pair<int, int>> slot((maximum - minimum) / slot_gap + 1,
make_pair(-1, -1));
    for (size_t i = 0; i < nums.size(); i++)
    {
        int index = (nums[i] - minimum) / slot_gap;
        pair<int, int> pair = slot[index];
        if (pair.first == -1)
        {
            pair.first = nums[i];
            pair.second = nums[i];
        }
        else
        {
            if (nums[i] < pair.first)
            {
                pair.first = nums[i];
            }
            else if (nums[i] > pair.second)
            {
                pair.second = nums[i];
            }
        }
        slot[index] = pair;
    }
    int max_gap = INT_MIN;
    int previous = -1;
    pair<int, int> pair;
    if (slot.size() == 1)
    {
        pair = slot[0];
        max_gap = pair.second - pair.first;
    }
    else
    {
        for (size_t i = 0; i < slot.size(); i++)
```

```
        {
            pair = slot[i];
            if (pair.first == -1) continue;
            if ((previous != -1) && (pair.first - previous) > max_gap)
            {
                max_gap = pair.first - previous;
            }
            previous = pair.second;
        }
    }
    return max_gap;
}


/// <summary>
/// Leet code #274. H-Index
/// Given an array of citations (each citation is a non-negative integer)
of
/// a researcher, write a function to compute the researcher's h-index.
/// According to the definition of h-index on Wikipedia: "A scientist has
index h
/// if h of his/her N papers have at
/// least h citations each, and the other N − h papers have no more than h
citations each."
/// For example, given citations = [3, 0, 6, 1, 5], which means the
researcher has 5 papers
/// in total and each of
/// them had received 3, 0, 6, 1, 5 citations respectively. Since the
researcher has 3 papers
/// with at least 3
/// citations each and the remaining two with no more than 3 citations
each, his h-index is 3.
/// Note: If there are several possible values for h, the maximum one is
taken as the h-index.
/// Hint:
/// 1.An easy approach is to sort the array first.
/// 2.What are the possible values of h-index?
/// 3.A faster approach is to use extra space.
/// </summary>
int LeetCode::hIndex(vector<int>& citations)
{
    vector<int> count_map(citations.size() + 1);
    for (size_t i = 0; i < citations.size(); i++)
    {
        if (citations[i] >= (int)citations.size())
        {
            count_map[citations.size()]++;
        }
        else
        {
            count_map[citations[i]]++;
        }
    }
    int index;
    for (index = citations.size(); index >= 0; index--)
    {
        if (index < (int)citations.size()) count_map[index] +=
count_map[index + 1];
```

```
        if (count_map[index] >= index) break;
    }
    return index;
}
```

Dutch Flag Sort

Assume you have only 3 unique number with duplication in a long list, and you want to sort it. The solution is using 3 pointers, low, from left to right, keep track on the smallest number, high, from right to left, keep track on the largest number, mid is used to scan from left to right in case both end stuck on the middle number.

```
/// <summary>
/// Leet code #75. Sort Colors
/// Given an array with n objects colored red, white or blue, sort them so
/// that objects of the same color are adjacent, with the colors in the
order
/// red, white and blue.
/// Here, we will use the integers 0, 1, and 2 to represent the color red,
/// white, and blue respectively.
/// Note:
/// You are not suppose to use the library's sort function for this
problem.
/// Follow up:
/// A rather straight forward solution is a two-pass algorithm using
counting
/// sort.
/// First, iterate the array counting number of 0's, 1's, and 2's, then
overwrite
/// array with total number of 0's, then 1's and followed by 2's.
/// Could you come up with an one-pass algorithm using only constant space?
/// </summary>
void LeetCode::sortColors(vector<int>& nums)
{
    int low, high, mid;
    low = 0;
    high = nums.size() - 1;
    mid = low;
    while ((low <= high) && (mid <= high))
    {
        if (nums[low] == 0)
        {
            low++;
            mid = low;
        }
        else if (nums[high] == 2)
        {
            high--;
        }
        else if (nums[mid] == 0)
        {
            swap(nums[low], nums[mid]);
            low++;
        }
        else if (nums[mid] == 2)
        {
            swap(nums[mid], nums[high]);
```

```
            high--;
        }
        else
        {
            mid++;
        }
    }
}
```

A more shorter solution

```
void LeetCode::sortColors(vector<int>& nums)
{
    int low, high, mid;
    low = 0;
    high = nums.size() - 1;
    mid = low;
    while (mid <= high)
    {
        if (nums[mid] == 0)
        {
            swap(nums[low], nums[mid]);
            low++;
            mid++;
        }
        else if (nums[mid] == 2)
        {
            swap(nums[mid], nums[high]);
            high--;
        }
        else
        {
            mid++;
        }
    }
}
```

Wiggle Sort

```
/// <summary>
/// Leet code #280. Wiggle Sort
///
/// Given an unsorted array nums, reorder it in-place such that
/// nums[0] <= nums[1] >= nums[2] <= nums[3]....
/// For example, given nums = [3, 5, 2, 1, 6, 4],
/// one possible answer is [1, 6, 2, 5, 3, 4].
/// </summary>
void LeetCode::wiggleSort(vector<int>& nums)
{
    for (size_t i = 1; i < nums.size(); i++)
    {
        if (i % 2 == 1)
        {
            if (nums[i] < nums[i - 1]) swap(nums[i], nums[i - 1]);
        }
        else
        {
            if (nums[i] > nums[i - 1]) swap(nums[i], nums[i - 1]);
        }
```

```
    }
}

/// <summary>
/// Leet code #324. Wiggle Sort II Add to List
/// Given an unsorted array nums, reorder it such that nums[0]
/// < nums[1] > nums[2] < nums[3]....
/// Example:
/// (1) Given nums = [1, 5, 1, 1, 6, 4], one possible answer is
/// [1, 4, 1, 5, 1, 6].
/// (2) Given nums = [1, 3, 2, 2, 3, 1], one possible answer is
/// [2, 3, 1, 3, 1, 2].
/// Note:
/// You may assume all input has valid answer.
/// Follow Up:
/// Can you do it in O(n) time and/or in-place with O(1) extra space?
/// </summary>
void LeetCode::wiggleSortII(vector<int>& nums)
{
    int n = nums.size();

    // Find a median.
    vector<int>::iterator middle_ptr = nums.begin() + n / 2;
    nth_element(nums.begin(), middle_ptr, nums.end());
    int value = *middle_ptr;

    // Index-rewiring.
    #define A(i) nums[(1+2*(i)) % (n|1)]

    // 3-way-partition-to-wiggly in O(n) time with O(1) space.
    int low =  0, mid = 0, high = n - 1;
    while (mid <= high)
    {
        if (A(mid) > value)
        {
            swap(A(low), A(mid));
            low++; mid++;
        }
        else if (A(mid) < value)
        {
            swap(A(mid), A(high));
            high--;
        }
        else
        {
            mid++;
        }
    }
}
```

Top K with smallest sum

1. To solve top K with smallest sum, we normally use heap sort, the default heap sort data structure is priority queue. (map also works, but please be careful to handle the duplication in the map, normally we need a count in the map value.)
2. When try to map array to array, we keep track the index of the array2 for each element in array1. This tricks works in many problems.

```
/// <summary>
/// Leet code #373. Find K Pairs with Smallest Sums
/// You are given two integer arrays nums1 and nums2 sorted in ascending
/// order and an integer k.
/// Define a pair (u,v) which consists of one element from the first array
/// and one element from the second array.
/// Find the k pairs (u1,v1),(u2,v2) ...(uk,vk) with the smallest sums.
/// Example 1:
/// Given nums1 = [1,7,11], nums2 = [2,4,6],  k = 3
/// Return: [1,2],[1,4],[1,6]
/// The first 3 pairs are returned from the sequence:
/// [1,2],[1,4],[1,6],[7,2],[7,4],[11,2],[7,6],[11,4],[11,6]
///
/// Example 2:
/// Given nums1 = [1,1,2], nums2 = [1,2,3],  k = 2
/// Return: [1,1],[1,1]
/// The first 2 pairs are returned from the sequence:
/// [1,1],[1,1],[1,2],[2,1],[1,2],[2,2],[1,3],[1,3],[2,3]
///
/// Example 3:
/// Given nums1 = [1,2], nums2 = [3],  k = 3
/// Return: [1,3],[2,3]
/// All possible pairs are returned from the sequence:
/// [1,3],[2,3]
/// </summary>
vector<pair<int, int>> LeetCode::kSmallestPairs(vector<int>& nums1,
vector<int>& nums2, int k)
{
    vector<pair<int, int>> result;
    if (nums1.empty() || nums2.empty()) return result;
    vector<int> array1(nums1.size(), 0);
    priority_queue<pair<int, int>> priority_queue;
    for (size_t i = 0; i < nums1.size(); i++)
    {
        priority_queue.push(make_pair(-(nums1[i] + nums2[0]), i));
    }
    for (int i = 0; i < k; i++)
    {
        if (priority_queue.empty()) break;
        pair<int, int> value_pair = priority_queue.top();
        priority_queue.pop();
        result.push_back(make_pair(nums1[value_pair.second],
nums2[array1[value_pair.second]]));
        array1[value_pair.second]++;
        if (array1[value_pair.second] < (int)nums2.size())
        {
            priority_queue.push(make_pair(-(nums1[value_pair.second] +
nums2[array1[value_pair.second]]), value_pair.second));
        }
    }
    return result;
}
```

Reconstruct by order

```
/// <summary>
/// Leet code #406. Queue Reconstruction by Height
```

```
/// Suppose you have a random list of people standing in a queue. Each
person
/// is described by a pair of integers (h, k), where h is the height of the
/// person and k is the number of people in front of this person who have a
/// height greater than or equal to h. Write an algorithm to reconstruct
/// the queue.
///
/// Note:
/// The number of people is less than 1,100.
///
/// Example
/// Input:
/// [[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]
///
/// Output:
/// [[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]
/// </summary>
vector<pair<int, int>> LeetCode::reconstructQueue(vector<pair<int, int>>&
people)
{
    vector<pair<int, int>> result;
    struct heightCompare
    {
        bool operator() (pair<int, int>& a, pair<int, int>& b)
        {
            if ((a.first > b.first) || ((a.first == b.first) && (a.second <
b.second)))
            {
                return true;
            }
            else
            {
                return false;
            }
        }
    };

    sort(people.begin(), people.end(), heightCompare());
    for (size_t i = 0; i < people.size(); i++)
    {
        vector<pair<int, int>>::iterator itr = result.begin();
        result.insert(itr + people[i].second, people[i]);
    }
    return result;
}
```

Longest Increasing Subsequence

The solution to longest increasing subsequence is to use a array to remember the increasing sequence, every time a new number comes, try to locate it with the first item in the array which is greater than it, if it is located at the end (no one is greater than the new one), means the size of increasing subsequence grow, otherwise, use the smaller value to replace the bigger number in the array and give the more potential for the future growth. Let's look at the following examples:

```
/// <summary>
/// Leet code #300. Longest Increasing Subsequence Add to List
/// Given an unsorted array of integers, find the length of longest
increasing
/// subsequence.
/// For example,
/// Given [10, 9, 2, 5, 3, 7, 101, 18],
/// The longest increasing subsequence is [2, 3, 7, 101], therefore the
length is
/// 4.
/// Note that there may be more than one LIS combination, it is only
necessary for
/// you to return the length.
/// Your algorithm should run in O(n2) complexity.
/// Follow up: Could you improve it to O(n log n) time complexity?
/// </summary>
int LeetCode::lengthOfLIS(vector<int>& nums)
{
    vector<int> result;
    for (int num : nums)
    {
        vector<int>::iterator iterator = lower_bound(result.begin(),
result.end(), num);
        if (iterator == result.end())
        {
            result.push_back(num);
        }
        else
        {
            *iterator = num;
        }
    }

    return result.size();
}


/// <summary>
/// Leet code #354. Russian Doll Envelopes
/// You have a number of envelopes with widths and heights given as a pair
of integers (w, h).
/// One envelope can fit into another if and only if both the width and
height of one envelope is greater than the width and height of the other
envelope.
///
/// What is the maximum number of envelopes can you Russian doll? (put one
inside other)
/// Example:
/// Given envelopes = [[5,4],[6,4],[6,7],[2,3]], the maximum number of
envelopes you can Russian doll is 3 ([2,3] => [5,4] => [6,7]).
/// This method implement the idea of Longest Increasing Sequence:
/// 1. sort the width in asc order
/// 2. if width are same, sort the height in desc order
/// 3. in the sequence find the longest increase height sequence, because
width must already be in asc.
///    and same width are skipped.
/// This is O(nlogn), due to sort.
/// </summary>
```

```
int LeetCode::maxEnvelopesV1(vector<pair<int, int>>& envelopes)
{
    vector<int> collector;
    struct envelope_order
    {
        bool operator() (pair<int, int> a, pair<int, int> b)
        {
            return ((a.first < b.first) || ((a.first == b.first) &&
(a.second > b.second)));
        }
    };
    std::sort(envelopes.begin(), envelopes.end(), envelope_order());
    for (pair<int, int>envelope : envelopes)
    {
        vector<int>::iterator iterator = lower_bound(collector.begin(),
collector.end(), envelope.second);
        if (iterator == collector.end())
        {
            collector.push_back(envelope.second);
        }
        else if (*iterator > envelope.second)
        {
            *iterator = envelope.second;
        }
    }
    return (int)collector.size();
}

/// <summary>
/// Leet code #354. Russian Doll Envelopes
/// You have a number of envelopes with widths and heights given as a pair
of integers (w, h).
/// One envelope can fit into another if and only if both the width and
height of one envelope is greater than the width and height of the other
envelope.
///
/// What is the maximum number of envelopes can you Russian doll? (put one
inside other)
/// Example:
/// Given envelopes = [[5,4],[6,4],[6,7],[2,3]], the maximum number of
envelopes you can Russian doll is 3 ([2,3] => [5,4] => [6,7])
///
/// This method implement the idea of dynamic processing
/// 1. sort it first
/// 2. each envelope must hold itself, so the matrix start with 1
/// 3. from small to large envelope, if a large envelope can hold a smaller
one,
///    then it can hold all the ones inside the small ones.
/// </summary>
int LeetCode::maxEnvelopesV2(vector<pair<int, int>>& envelopes)
{
    struct envelope_order
    {
        bool operator() (pair<int, int> a, pair<int, int> b)
        {
            return ((a.first < b.first) || ((a.first == b.first) &&
(a.second > b.second)));
        }
```

```
    };
    std::sort(envelopes.begin(), envelopes.end(), envelope_order());
    vector<int> collector;
    int maxNumber = 0;
    for (size_t i = 0; i < envelopes.size(); i++)
    {
        collector.push_back(1);
        for (size_t j = 0; j < i; j++)
        {
            if ((envelopes[i].first > envelopes[j].first) &&
                (envelopes[i].second > envelopes[j].second))
            {
                collector[i] = max(collector[i], collector[j] + 1);
            }
        }
        maxNumber = max(maxNumber, collector[i]);
    }
    return maxNumber;
}
```

Sort and count

The following problem is to count the number of elements smaller than the current item on its right. When you see such count the number smaller or greater than itself on the left or right, you can consider two standard methods, using the BST or use merge sort. But if you are using C++ and if the number of element is not too big, you can also directly use vector insert. Or if you can predict the range of the number, using Binary index tree is also an option.

Please look at the following example:

```
/// <summary>
/// Leet code #315. Count of Smaller Numbers After Self
///
/// You are given an integer array nums and you have to return a new counts
array.
/// The counts array has the property where counts[i] is the number of
smaller
/// elements to the right of nums[i].
/// Example:
/// Given nums = [5, 2, 6, 1]
/// To the right of 5 there are 2 smaller elements (2 and 1).
/// To the right of 2 there is only 1 smaller element (1).
/// To the right of 6 there is 1 smaller element (1).
/// To the right of 1 there is 0 smaller element.
/// Return the array [2, 1, 1, 0].
/// </summary>
```

The first solution is a simple vector insertion, which violate the complexity requirement (it is a O(n^2), not O(n(log(n)), but surprisingly, due to optimized vector insertion, the performance is close to BST and quicker than merge sort for the leet code test case. This experiment means you may use this method in the real working.

```
 /// <summary>
/// Leet code #315. Count of Smaller Numbers After Self
///
/// You are given an integer array nums and you have to return a new counts
array.
/// The counts array has the property where counts[i] is the number of
smaller elements to the right of nums[i].
/// Example:
/// Given nums = [5, 2, 6, 1]
/// To the right of 5 there are 2 smaller elements (2 and 1).
/// To the right of 2 there is only 1 smaller element (1).
/// To the right of 6 there is 1 smaller element (1).
/// To the right of 1 there is 0 smaller element.
/// Return the array [2, 1, 1, 0].
/// </summary>
vector<int> LeetCode::countSmaller(vector<int>& nums)
{
    vector<int> num_map;
    vector<int> result(nums.size());
    for (int i = nums.size() - 1; i >= 0; i--)
    {
        vector<int> ::iterator itr = lower_bound(num_map.begin(),
num_map.end(), nums[i]);
        int index = itr - num_map.begin();
        result[i] = index;
        num_map.insert(itr, nums[i]);
    }
    return result;
}
```

The second solution is based on binary search tree, during the tree build out, every node should remember the number of its left children plus itself, i.e. the number of elements smaller than or equal to itself, so if the new node fall into the right branch it simply adds the count in the parent node.

```
vector<int> LeetCode::countSmallerII(vector<int>& nums)
{
    TreeNode * root = nullptr;
    vector<int> result(nums.size());
    for (int i = nums.size() - 1; i >= 0; i--)
    {
        countSmaller(root, nums[i], result[i]);
    }
    return result;
}

struct TreeNode {
    int val;
    int count;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), count(0), left(NULL), right(NULL) {}
};
```

```
/// <summary>
/// Insert BST node and update count
/// </summary>
void LeetCode::countSmallerII(TreeNode * &root, int value, int& count)
{
    if (root == nullptr)
    {
        root = new TreeNode(value);
        root->count = 1;
    }
    else
    {
        if (value <= root->val)
        {
            root->count++;
            countSmaller(root->left, value, count);
        }
        else
        {
            count += root->count;
            countSmaller(root->right, value, count);
        }
    }
}
```

The third solution is based on merge sort, we assume in each merge sort the smaller and right elements count within either left half or right half is already done, you only need to count for each element in left half, how many items in the right are smaller than it, because the right half is always on the right, so no need to judge position again, also due to right side is already in order, it is a linear O(n). (you do not need to recount the elements already counted in the left neighbors).

```
/// <summary>
/// Leet code #315. Count of Smaller Numbers After Self
/// Note: end is out of the boundary of the last element, e.g.
vector.size(),
/// not the index of last element.
/// </summary>
void LeetCode::mergeSortCountSmaller(vector<pair<int, int>>& nums, int
begin, int end, vector<int>&result)
{
    if (begin + 1 >= end) return;
    int mid = begin + (end - begin) / 2;
    mergeSortCountSmaller(nums, begin, mid, result);
    mergeSortCountSmaller(nums, mid, end, result);
    int left = begin, right = mid;
    int count = 0;
    // we may have right pointer stay at end position for a
    // while for the left elements to catch up
    while (left < mid  && right <= end)
    {
        while ((right < end) && (nums[left].first > nums[right].first))
right++;
        count = right - mid;
        result[nums[left].second] += count;
        left++;
```

```
    }
    sort(nums.begin() + begin, nums.begin() + end);
}

/// <summary>
/// Leet code #315. Count of Smaller Numbers After Self
///
/// You are given an integer array nums and you have to return a new counts
array.
/// The counts array has the property where counts[i] is the number of
smaller
/// elements to the right of nums[i].
/// Example:
/// Given nums = [5, 2, 6, 1]
/// To the right of 5 there are 2 smaller elements (2 and 1).
/// To the right of 2 there is only 1 smaller element (1).
/// To the right of 6 there is 1 smaller element (1).
/// To the right of 1 there is 0 smaller element.
/// Return the array [2, 1, 1, 0].
/// </summary>
vector<int> LeetCode::countSmallerIII(vector<int>& nums)
{
    vector<pair<int, int>> num_pairs;
    vector<int> result(nums.size());
    for (size_t i = 0; i < nums.size(); i++)
    {
        num_pairs.push_back(make_pair(nums[i], i));
    }
    mergeSortCountSmaller(num_pairs, 0, nums.size(), result);
    return result;
}
```

The fourth solution is based on binary index tree, assume each number is an integer, then when you have a new number you may update as many as 32 slot. The issue for this solution is that you need a large memory (4GB+) to store the binary index tree, or you can use a map to replace the vector which may further impact the performance. This solution is suit for the scenario when you know the range of the numbers, especially a small range, and a lot of duplicates.

```
But remember theoretically this is a O(nlog(n)) solution. (even someone can
argue it is a O(n) if you consider multiple 32 is a constant)

/// <summary>
/// Leet code #315. Count of Smaller Numbers After Self
/// </summary>
void LeetCode::addBIT(int index, vector<int>& accu_slot)
{
    while (index < (int)accu_slot.size())
    {
        accu_slot[index] += 1;
        index += (index & -index);
    }
}

/// <summary>
/// Leet code #315. Count of Smaller Numbers After Self
```

```
/// </summary>
int LeetCode::sumBIT(int index, vector<int>& accu_slot)
{
    int sum = 0;
    while (index != 0)
    {
        sum += accu_slot[index];
        index -= index & -index;
    }
    return sum;
}

/// <summary>
/// Leet code #315. Count of Smaller Numbers After Self
///
/// You are given an integer array nums and you have to return a new counts
array.
/// The counts array has the property where counts[i] is the number of
smaller
/// elements to the right of nums[i].
/// Example:
/// Given nums = [5, 2, 6, 1]
/// To the right of 5 there are 2 smaller elements (2 and 1).
/// To the right of 2 there is only 1 smaller element (1).
/// To the right of 6 there is 1 smaller element (1).
/// To the right of 1 there is 0 smaller element.
/// Return the array [2, 1, 1, 0].
/// </summary>
vector<int> LeetCode::countSmallerIV(vector<int>& nums)
{
    vector<int> result(nums.size());
    int max_num = INT_MIN;
    int min_num = INT_MAX;
    for (size_t i = 0; i < nums.size(); i++)
    {
        max_num = max(max_num, nums[i]);
        min_num = min(min_num, nums[i]);
    }
    // the min_num should be located at 1 and
    // the max_num should be located at max_num-min_num + 1
    vector<int> accu_slot(max_num - min_num + 2);

    for (int i = nums.size() - 1; i >= 0; i--)
    {
        addBIT(nums[i] - min_num + 1, accu_slot);
        result[i] = sumBIT(nums[i] - min_num, accu_slot);
    }
    return result;
}
/// <summary>
/// Leet code #327. Count of Range Sum
/// </summary>
int LeetCode::mergeSort(vector<long long>& sums, int begin, int end, int
lower, int upper)
{
    if (begin + 1 >= end) return 0;
    int middle = begin + (end - begin) / 2;
    int count = 0;
```

```
    count = mergeSort(sums, begin, middle, lower, upper) +
            mergeSort(sums, middle, end, lower, upper);
    int m = middle, n = middle;
    for (int i = begin; i < middle; i++)
    {
        while ((m < end) && (sums[m] - sums[i] < lower)) m++;
        while ((n < end) && (sums[n] - sums[i] <= upper)) n++;
        count += n - m;
    }
    inplace_merge(sums.begin() + begin, sums.begin() + middle, sums.begin()
+ end);
    return count;
}

/// <summary>
/// Leet code #327. Count of Range Sum
///
/// Given an integer array nums, return the number of range sums that lie
in [lower, upper] inclusive.
/// Range sum S(i, j) is defined as the sum of the elements in nums between
indices i and j (i ≤ j), inclusive.
///
/// Note:
/// A naive algorithm of O(n2) is trivial. You MUST do better than that.
/// Example:
/// Given nums = [-2, 5, -1], lower = -2, upper = 2,
/// Return 3.
/// The three ranges are : [0, 0], [2, 2], [0, 2] and their respective sums
are: -2, -1, 2.
/// </summary>
int LeetCode::countRangeSum(vector<int>& nums, int lower, int upper)
{
    vector<long long> sums;
    long long sum = 0;
    sums.push_back(0);
    for (size_t i = 0; i < nums.size(); i++)
    {
        sum += nums[i];
        sums.push_back(sum);
    }
    return mergeSort(sums, 0, sums.size(), lower, upper);
}

/// <summary>
/// Leet code #327. Count of Range Sum
///
/// Given an integer array nums, return the number of range sums that lie
in [lower, upper] inclusive.
/// Range sum S(i, j) is defined as the sum of the elements in nums between
indices i and j (i ≤ j), inclusive.
///
/// Note:
/// A naive algorithm of O(n2) is trivial. You MUST do better than that.
/// Example:
/// Given nums = [-2, 5, -1], lower = -2, upper = 2,
/// Return 3.
/// The three ranges are : [0, 0], [2, 2], [0, 2] and their respective sums
are: -2, -1, 2.
```

```
/// </summary>
int LeetCode::countRangeSumII(vector<int>& nums, int lower, int upper)
{
    vector<long long> sum_list;
    long long sum = 0;
    int result = 0;
    for (size_t i = 0; i < nums.size(); i++)
    {
        sum += nums[i];
        if ((sum >= lower) && (sum <= upper))
        {
            result++;
        }

        vector<long long> ::iterator start = lower_bound(sum_list.begin(),
sum_list.end(), sum - upper);
        vector<long long> ::iterator end = upper_bound(sum_list.begin(),
sum_list.end(), sum - lower);
        int start_index = start - sum_list.begin();
        int end_index = end - sum_list.begin();
        result += end_index - start_index;

        vector<long long> ::iterator itr = lower_bound(sum_list.begin(),
sum_list.end(), sum);
        sum_list.insert(itr, sum);
    }
    return result;
}
```

Dynamic Programming

Dynamic programming becomes one of the most popular problems in the interview question. Unlike backing tracking, dynamic programming has so many faces but we still can catch it by the following patterns.

Look back

This may be the most common style and the easiest one in dynamic programming. 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. Dynamic programming often use a one dimension or two-dimension array to resolve the problem, in each element of the array you remember the partial result worked out so far.

1. The following problem is to calculate maximum rob value based on the past accumulated value, every time you need to decide, steal or not steal the current house. The decision is made based on the calculate value in each choice.

The complexity for this solution is **O(n)**

```
/// <summary>
/// Leet code #198. House Robber
/// You are a professional robber planning to rob houses along a street.
Each
/// house has a certain amount of money stashed, the only constraint
stopping
/// you from robbing each of them is that adjacent houses have security
system
/// connected and it will automatically contact the police if two adjacent
/// houses were broken into on the same night.
/// Given a list of non-negative integers representing the amount of money
of
/// each house, determine the maximum amount of money you can rob tonight
/// without alerting the police.
/// </summary>
int LeetCode::rob(vector<int>& nums)
{
    if (nums.size() == 0) return 0;
    vector<int> matrix(nums.size());
    for (size_t i = 0; i < nums.size(); i++)
    {
        // You start with first house
        if (i == 0)
        {
            matrix[i] = nums[i];
        }
        // when at the second house, you simply compare the value of these
two
        // houses
        else if (i == 1)
        {
            matrix[i] = max(nums[i-1], nums[i]);
        }
        // when you have more than two houses, you need to think whether
        // you should steal the current house or the previous house
        else
        {
            matrix[i] = max(matrix[i-2] + nums[i], matrix[i-1]);
        }
    }
    return matrix[nums.size() -1];
}
```

2. The following problem is to calculate the minimum coins needed to get a specific amount. We build the result based on past result, if we deduct any coin from existing value.

```
/// <summary>
/// Leet code #322. Coin Change
/// You are given coins of different denominations and a total amount of
money
/// amount. Write a function to
/// compute the fewest number of coins that you need to make up that
amount.
/// If that amount of money cannot
```

```
/// be made up by any combination of the coins, return -1.
/// Example 1:
/// coins = [1, 2, 5], amount = 11
/// return 3 (11 = 5 + 5 + 1)
/// Example 2:
/// coins = [2], amount = 3
/// return -1.
/// Note:
/// You may assume that you have an infinite number of each kind of coin.
/// </summary>
int LeetCode::coinChange(vector<int>& coins, int amount)
{
    vector<int> matrix(amount + 1, -1);
    for (int i = 0; i <= amount; i++)
    {
        if (i == 0) matrix[i] = 0;
        for (size_t j = 0; j < coins.size(); j++)
        {
            // for each amount value i, we can deduct every possible coin
value
            // oins[j], if there is a previously calculated resul there,
            // then increase by one as the candidate result for amount i,
            // choose the minimum one
            if ((i >= coins[j]) && (matrix[i-coins[j]] != -1))
            {
                if (matrix[i] != -1)
                {
                    matrix[i] = min(matrix[i], matrix[i - coins[j]] + 1);
                }
                else
                {
                    matrix[i] = matrix[i - coins[j]] + 1;
                }
            }
        }
    }
    return matrix[amount];
}
```

Carry forward

However, in some case when you stand on some position, you are not able to look back, because you do not know where to look. In this case you need to carry the past result forward into the array.

The complexity for this solution is **O(n)**

```
/// <summary>
/// Leet code #55. Jump Game
/// Given an array of non-negative integers, you are initially positioned
at
/// the first index of the array.
/// Each element in the array represents your maximum jump length at that
```

```
/// position.
/// Determine if you are able to reach the last index.
/// For example:
/// A = [2,3,1,1,4], return true.
/// A = [3,2,1,0,4], return false.
/// </summary>
bool LeetCode::canJump(vector<int>& nums)
{
    if (nums.size() <= 1) return true;
    vector<int> collection;
    size_t index = 0;
    collection.push_back(0);
    // we look at every established position. from starting position of 0
    while (index < collection.size())
    {
        int last_stop = collection.back();
        // here we do a small optimization, only if we jump out of
        // farthest position we will care, because other wise, anyway we
will
        // such position later.
        for (int i = last_stop + 1; i <= collection[index] +
nums[collection[index]]; i++)
        {
            collection.push_back(i);
            if (i == nums.size() - 1)
            {
                return true;
            }
        }
        index++;
    }
    return false;
}

/// <summary>
/// Leet Code #403 Frog Jump
/// A frog is crossing a river.The river is divided into x units and at
/// each unit there may or may not exist a stone.The frog can jump on a
/// stone, but it must not jump into the water.
/// Given a list of stones' positions (in units) in sorted ascending order,
/// determine if the frog is able to cross the river by landing on the last
/// stone. Initially, the frog is on the first stone
/// and assume the first jump must be 1 unit.
/// If the frog has just jumped k units, then its next jump must be either
/// k - 1, k, or k + 1 units.
//  Note that the frog can only jump in the forward direction.
/// Note:
///   The number of stones is ≥ 2 and is < 1100.
///   Each stone's position will be a non-negative integer < 231.
///   The first stone's position is always 0.
/// Example 1 :
///   [0, 1, 3, 5, 6, 8, 12, 17]
/// There are a total of 8 stones.
///   The first stone at the 0th unit, second stone at the 1st unit,
///   third stone at the 3rd unit, and so on...
///  The last stone at the 17th unit.
/// Return true.The frog can jump to the last stone by jumping
/// 1 unit to the 2nd stone, then 2 units to the 3rd stone, then
```

```
/// 2 units to the 4th stone, then 3 units to the 6th stone,
/// 4 units to the 7th stone, and 5 units to the 8th stone.
/// Example 2:
///   [0, 1, 2, 3, 4, 8, 9, 11]
/// Return false.There is no way to jump to the last stone as
///   the gap between the 5th and 6th stone is too large.
/// </summary>
/// <param name="stones">The stones where they are located in</param>
/// <returns>true, if solution found</returns>
bool LeetCode::canCross(vector<int>& stones)
{
    size_t size = stones.size();
    // if any empty stones immediately
    if (size == 0)
    {
        return false;
    }
    unordered_map <int, unordered_set<int>> map;
    for (size_t i = 0; i < stones.size(); i++)
    {
        map[stones[i]] = unordered_set<int>();
    }
    // insert initial step
    map[0].insert(0);
    // iterate every stone
    for (size_t i = 0; i < stones.size(); i++)
    {
        for (unordered_set<int>::iterator iterator =
map[stones[i]].begin(); iterator != map[stones[i]].end(); ++iterator)
        {
            // for every step from this stone, we see where it goes, if
this is a new position, insert to the partial result
            int last_step = *iterator;
            for (int current_step = last_step - 1; current_step <=
last_step + 1; current_step++)
            {
                if (current_step > 0)
                {
                    int next_stone = stones[i] + current_step;
                    if (map.find(next_stone) != map.end())
                    {
                        map[next_stone].insert(current_step);
                    }
                }
            }
        }
    }
    if (map[stones.back()].size() != 0)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Multiple Iterations by formula

Sometimes, we can also walk through the input data array multiple times by apply some formula.

The following example requires a formula of best stock buy price and the sell price (profit).

The complexity for this solution is **O(n)**

```
/// <summary>
/// Leet code #123. Best Time to Buy and Sell Stock III
/// Say you have an array for which the ith element
/// is the price of a given stock on day i.
///   Design an algorithm to find the maximum profit.
/// You may complete at most two transactions.
/// Note:
///   You may not engage in multiple transactions at the same time
///   (ie, you must sell the stock before you buy again).
/// </summary>
int LeetCode::maxProfitTwoTxns(vector<int>& prices)
{
    if (prices.size() == 0)
    {
        return 0;
    }
    vector<int> balance(prices.size());
    for (size_t s = 0; s < 2; s++)
    {
        int best_buy = INT_MIN;
        int best_profit = 0;
        for (size_t i = 0; i < prices.size(); i++)
        {
            best_buy = max(best_buy, balance[i] - prices[i]);
            best_profit = max(best_profit, best_buy + prices[i]);
            balance[i] = best_profit;
        }
    }
    return balance[balance.size() - 1];
}
```

The following problem want to calculate the candy number by scan the array from left to right, then from right to left.

```
/// <summary>
/// Leet code # 135. Candy
///
/// There are N children standing in a line. Each child is assigned a
/// rating value.
/// You are giving candies to these children subjected to the
/// following requirements:
/// Each child must have at least one candy.
/// Children with a higher rating get more candies than their neighbors.
/// What is the minimum candies you must give?
/// </summary>
int LeetCode::candy(vector<int>& ratings)
{
    vector<int> candy(ratings.size());
    // passing from left to right
```

```
    for (int i = 0; i < (int)ratings.size(); i++)
    {
        if (i == 0)
        {
            candy[i] = 1;
        }
        else if (ratings[i] > ratings[i - 1])
        {
            candy[i] = max(candy[i - 1] + 1, 1);
        }
        else
        {
            candy[i] = 1;
        }
    }
    // passing from right to left
    for (int i = ratings.size() - 1; i >= 0; i--)
    {
        if (i == ratings.size() - 1)
        {
            candy[i] = max(candy[i], 1);
        }
        else if (ratings[i] > ratings[i + 1])
        {
            candy[i] = max(candy[i + 1] + 1, candy[i]);
        }
        else
        {
            candy[i] = max(candy[i], 1);
        }
    }
    int sumCandy = 0;
    for (size_t i = 0; i < candy.size(); i++)
    {
        sumCandy += candy[i];
    }
    return sumCandy;
}
```

Cross Apply with two array

Such problem normally has two input arrays, in many cases they are two strings, you try to find the answer by cross applying between them.

Assume the two input arrays having the size of m and n, so you will work out an two dimension array with the size of (m+1)*(n+1), (please note empty array also count one position). The in any specific position such as A[i][j], you can build it out from the previous result such as A[i-1][j], A[i][j-1] or A[i-1][j-1].

```
/// <summary>
/// Leet code #72. Edit Distance
/// Given two words word1 and word2, find the minimum number of steps
required to
/// convert word1 to
/// word2. (each operation is counted as 1 step.)
/// You have the following 3 operations permitted on a word:
```

```
/// a) Insert a character
/// b) Delete a character
/// c) Replace a character
/// </summary>
int LeetCode::minDistance(string word1, string word2)
{
    int distance = 0;
    vector<vector<int>> distance_map;
    // The matrix should be [s.size() + 1][p.size() + 1]
    // This is because the first row and first column
    // if left for empty string.
    for (size_t i = 0; i <= word1.size(); i++)
    {
        distance_map.push_back(vector<int>(word2.size()+1));
    }
    for (size_t i = 0; i <= word1.size(); i++)
    {
        for (size_t j = 0; j <= word2.size(); j++)
        {
            // when word1 is empty string, the distance is to insert every
            /// character in word2
            if (i == 0)
            {
                distance_map[i][j] = j;
            }
            // when word2 is empty string, the distance is to delete every
            // character of word1
            else if (j == 0)
            {
                distance_map[i][j] = i;
            }
            else
            {
                // if the character match, we inherit the edit size from
                // the subset
                if (word1[i - 1] == word2[j - 1])
                {
                    distance_map[i][j] = distance_map[i - 1][j - 1];
                }
                // otherwise, we increase the size by 1 from the subset,
                // by replacing a character
                else
                {
                    distance_map[i][j] = distance_map[i - 1][j - 1] + 1;
                }
                distance_map[i][j] = min(distance_map[i][j], distance_map[i
- 1][j] + 1);
                distance_map[i][j] = min(distance_map[i][j],
distance_map[i][j - 1] + 1);
            }
        }
    }
    return distance_map[word1.size()][word2.size()];
}


/// Leet code # 10. Regular Expression Matching
///
```

```
/// Implement regular expression matching with support for '.' and '*'.
///
/// '.' Matches any single character.
/// '*' Matches zero or more of the preceding element.
///
/// The matching should cover the entire input string (not partial).
/// The function prototype should be:
///
/// bool isMatchRegularExpression(const char *s, const char *p)
///
/// Some examples:
/// isMatch("aa","a")       -> false
/// isMatch("aa","aa")      -> true
/// isMatch("aaa","aa")     -> false
/// isMatch("aa", "a*")     -> true
/// isMatch("aa", ".*")     -> true
/// isMatch("ab", ".*")     -> true
/// isMatch("aab", "c*a*b") -> true
/// </summary>
bool LeetCode::isMatchRegularExpression(string s, string p)
{
    vector<vector<bool>> matrix;
    matrix.push_back(vector<bool>(p.size() + 1));

    matrix[0][0] = true;
    for (size_t i = 0; i < p.size(); i++)
    {
        if ((p[i] == '*') && (i >= 1))
            matrix[0][i + 1] = matrix[0][i - 1];
        else
            matrix[0][i + 1] = false;
    }

    for (size_t i = 0; i < s.size(); i++)
    {
        matrix.push_back(vector<bool>(p.size() + 1));
        matrix[i + 1][0] = false;
        for (size_t j = 0; j < p.size(); j++)
        {
            if ((s[i] == p[j]) || (p[j] == '.'))
            {
                matrix[i + 1][j + 1] = matrix[i][j];
            }
            else if ((p[j] == '*') && (j > 0))
            {
                // if character matches, and the pattern character is *,
                // it can match under 3 conditions.
                // 1. if without *, it matches,
                // 2. if without * and the character before
                //    (0 occurance for character),
                // 3. match until the last character in source,
                //    This means multiple occurance.
                if (((s[i] == p[j - 1]) || (p[j-1] == '.')))
                {
                    if (matrix[i + 1][j - 1] || matrix[i + 1][j] ||
matrix[i][j+1])
                    {
                        matrix[i + 1][j + 1] = true;
```

```
                    }
                    else
                    {
                        matrix[i + 1][j + 1] = false;
                    }
                }
                else
                {
                    if (matrix[i + 1][j - 1])
                    {
                        matrix[i + 1][j + 1] = true;
                    }
                    else
                    {
                        matrix[i + 1][j + 1] = false;
                    }
                }
            }
            else
            {
                matrix[i + 1][j + 1] = false;
            }
        }
    }

    return matrix[s.size()][p.size()];
}
```

Backpack problem

Backpack problem is that given a specific capacity bag, and a list of items, each has a size and value, you want to select some of the items (not all) to satisfy two conditions. **The size must be integer** but value can be any type.

1. The total items should be able to accommodate in the bag, i.e. the total size is less than bag size.
2. The total value of the items is maximum.

The solution is to build from bottom to top, starting from bag size as 1, to the whole bag size. On every specific bag size, you need to determine, if I select item X, then my total value is the maximum value of the items I can select with less size of X plus the value of X. In this way you choose what is the most optimized solution.

The formula is below,

Assume we are evaluating the size of S on item with size of Si and value of Vi, and we have already evaluated all the size smaller than S.

f(S) = max(f(S), f(S-Si) + Vi);

Let's look at the following example, the only special thing in the following example the size has two dimension, number of zero and number of one.

```
/// <summary>
```

```
/// Leet code #474. Ones and Zeroes
///
/// In the computer world, use restricted resource you have to generate
/// maximum benefit is what we always want to pursue.
/// For now, suppose you are a dominator of m 0s and n 1s respectively.
/// On the other hand,
/// there is an array with strings consisting of only 0s and 1s.
/// Now your task is to find the maximum number of strings that you can
/// form with given m 0s and n 1s.
/// Each 0 and 1 can be used at most once.
/// Note:
/// The given numbers of 0s and 1s will both not exceed 100
/// The size of given string array won't exceed 600.
/// Example 1:
/// Input: Array = {"10", "0001", "111001", "1", "0"}, m = 5, n = 3
/// Output: 4
///
/// Explanation: This are totally 4 strings can be formed by the using of 5
0s
/// and 3 1s, which are “10,”0001”,”1”,”0”
///
/// Example 2:
/// Input: Array = {"10", "0", "1"}, m = 1, n = 1
/// Output: 2
///
/// Explanation: You could form "10", but then you'd have nothing left.
/// Better form "0" and "1".
/// </summary>
int LeetCode::findMaxOneZeroForm(vector<string>& strs, int m, int n)
{
    if (strs.size() == 0) return 0;
    vector<pair<int, int>> str_count_list;
    for (size_t i = 0; i < strs.size(); i++)
    {
        pair<int, int> str_count = make_pair(0, 0);
        for (size_t j = 0; j < strs[i].size(); j++)
        {
            if (strs[i][j] == '0') str_count.first++;
            else if (strs[i][j] == '1') str_count.second++;
        }
        str_count_list.push_back(str_count);
    }

    vector<vector<int>> dp_map(vector<vector<int>>(m + 1, vector<int>(n +
1)));
    for (size_t i = 0; i < str_count_list.size(); i++)
    {
        for (int zero = m; zero >= 0; zero--)
        {
            for (int one = n; one >= 0; one--)
            {
                if ((zero >= str_count_list[i].first) && (one >=
str_count_list[i].second))
                {
                    dp_map[zero][one] = max(1 + dp_map[zero -
str_count_list[i].first][one - str_count_list[i].second], dp_map[zero]
[one]);
                }
```

```
            }
        }
    }
    return dp_map[m][n];
}
```

Two Dimension Dynamic Programming

Two-dimension dynamic processing is like the one dimension dynamic processing in the sense that you should divide your problem in to sub problems and solve the problem based on evaluating the result of sub problems.

Two-dimension dynamic processing is normally used to calculate the most optimal value on a range of data. This range can be a one dimension array. To get the value for a big range, normally you start from a small range first, in most of the case, this is a single item in the array, then the two adjacent items, and then 3, 4 5 adjacent. You can either divide a big range into left or right as sub range or look it as outer or inner scopes. In any case, the value of a list of adjacent items can be represented as a two-dimension array A[i][j], where i stands for the starting position and j is the end position.

To build this two-dimensions array up you start from A[i][i], then you do A[i][i+1], A[i][i+2], (in many case your code need to process A[i][i] and A[i][i+1] specially), until you do A[0][size-1]. And the final result is stored in A[0][size-1].

Examples in Leet Code

```
/// <summary>
/// Leet code #375. Guess Number Higher or Lower II
///
/// We are playing the Guess Game. The game is as follows:
/// I pick a number from 1 to n. You have to guess which number I picked.
/// Every time you guess wrong, I'll tell you whether the number I picked is higher or lower.
/// However, when you guess a particular number x, and you guess wrong, you pay $x.
/// You win the game when you guess the number I picked.
/// Example:
/// n = 10, I pick 8.
/// First round:  You guess 5, I tell you that it's higher. You pay $5.
/// Second round: You guess 7, I tell you that it's higher. You pay $7.
/// Third round:  You guess 9, I tell you that it's lower. You pay $9.
/// Game over. 8 is the number I picked.
/// You end up paying $5 + $7 + $9 = $21.
/// Given a particular n ≥ 1, find out how much money you need to have to guarantee a win.
/// Hint:
/// 1.The best strategy to play the game is to minimize the maximum loss you could possibly face.
///   Another strategy is to minimize the expected loss. Here, we are interested in the first scenario.
/// 2. Take a small example (n = 3). What do you end up paying in the worst case?
/// 3. Check out this article if you're still stuck.
```

```
/// 4.The purely recursive implementation of minimax would be worthless for even a small
n. You MUST use dynamic programming.
/// 5.As a follow-up, how would you modify your code to solve the problem of minimizing
the expected loss,
///   instead of the worst-case loss?
/// </summary>
int LeetCode::getMoneyAmount(int n)
{
    vector<vector<int>> matrix(n, vector<int>(n));
    for (int step = 0; step < n; step++)
    {
        for (int i = 0; i < n; i++)
        {
            if (step == 0)
            {
                matrix[i][i + step] = 0;
            }
            else if ((step == 1) && (i + step < n))
            {
                matrix[i][i + step] = i + 1;
            }
            else if (i + step < n)
            {
                int min_value = INT_MAX;
                for (int k = i + 1; k < i + step; k++)
                {
                    int left = matrix[i][k - 1];
                    int right = matrix[k + 1][i + step];
                    int value = max(left, right) + k + 1;
                    min_value = min(min_value, value);
                }
                matrix[i][i + step] = min_value;
            }
            else
            {
                break;
            }
        }
    }
    return matrix[0][n-1];
}

/// <summary>
/// Leet code #516. Longest Palindromic Subsequence
///
/// Given a string s, find the longest palindromic subsequence's length in s. You may
/// assume that the maximum length of s is 1000.
///
/// Example 1:
/// Input:
/// "bbbab"
/// Output: 4
/// One possible longest palindromic subsequence is "bbbb".
///
/// Example 2:
/// Input:
/// "cbbd"
/// Output: 2
```

```
/// One possible longest palindromic subsequence is "bb".
/// </summary>
int LeetCode::longestPalindromeSubseq(string s)
{
    int n = s.size();
    vector<vector<int>> matrix(n, vector<int>(n));
    for (int step = 0; step < n; step++)
    {
        for (int i = 0; i < n; i++)
        {
            if (step == 0)
            {
                matrix[i][i + step] = 1;
            }
            else if ((step == 1) && (i + step < n))
            {
                matrix[i][i + step] = (s[i] == s[i+step])? 2 : 1;
            }
            else if (i + step < n)
            {
                int max_value = INT_MIN;
                if (s[i] == s[i + step])
                {
                    max_value = max(max_value, 2 + matrix[i + 1][i + step - 1]);
                }
                max_value = max(max_value, matrix[i + 1][i + step]);
                max_value = max(max_value, matrix[i][i + step - 1]);
                matrix[i][i + step] = max_value;
            }
            else
            {
                break;
            }
        }
    }
    return matrix[0][n - 1];
}

/// <summary>
/// Leet code #312. Burst Balloons
/// Given n balloons, indexed from 0 to n-1. Each balloon is painted with a number on it
represented by array nums.
/// You are asked to burst all the balloons. If the you burst balloon i you will get nums[left]
* nums[i] * nums[right] coins.
/// Here left and right are adjacent indices of i. After the burst, the left and right then
becomes adjacent.
/// Find the maximum coins you can collect by bursting the balloons wisely.
/// Note:
/// (1) You may imagine nums[-1] = nums[n] = 1. They are not real therefore you can not
burst them.
/// (2) 0 ≤ n ≤ 500, 0 ≤ nums[i] ≤ 100
/// Example:
/// Given [3, 1, 5, 8]
/// Return 167
/// nums = [3,1,5,8] --> [3,5,8] -->   [3,8]   -->  [8]  --> []
/// coins =  3*1*5     +  3*5*8    +  1*3*8     + 1*8*1   = 167
/// </summary>
int LeetCode::maxBurstBalloonCoins(vector<int>& nums)
```

```
{
    vector<int> balloons;
    balloons.push_back(1);
    for (size_t i = 0; i < nums.size(); i++)
    {
        balloons.push_back(nums[i]);
    }
    balloons.push_back(1);
    vector<vector<int>> coins(balloons.size(), vector<int>(balloons.size(), 0));

    for (size_t gap = 2; gap < balloons.size(); gap++)
    {
        for (size_t i = 0; i < balloons.size() - gap; i++)
        {
            size_t first = i;
            size_t last = i + gap;
            if (gap == 2)
            {
                coins[first][last] = balloons[first] * balloons[first + 1] * balloons[last];
            }
            else
            {
                int max_coins = 0;
                for (size_t j = first + 1; j < last; j++)
                {
                    int sum_coins =
                        coins[first][j] + coins[j][last] +
                        balloons[first] * balloons[j] * balloons[last];
                    max_coins = max(max_coins, sum_coins);
                }
                coins[first][last] = max_coins;
            }
        }
    }
    return coins[0][balloons.size() - 1];
}


/// <summary>
/// Leet code #486. Predict the Winner
///
/// Given an array of scores that are non-negative integers. Player 1 picks
/// one of the numbers from either end of the array followed by the player
2
/// and then player 1 and so on. Each time a player picks a number, that
/// number will not be available for the next player. This continues until
/// all the scores have been chosen. The player with the maximum score
wins.
///
/// Given an array of scores, predict whether player 1 is the winner. You
/// can assume each player plays to maximize his score.
///
/// Example 1:
/// Input: [1, 5, 2]
/// Output: False
/// Explanation: Initially, player 1 can choose between 1 and 2.
/// If he chooses 2 (or 1), then player 2 can choose from 1 (or 2) and 5.
/// If player 2 chooses 5, then player 1 will be left with 1 (or 2).
```

```
/// So, final score of player 1 is 1 + 2 = 3, and player 2 is 5.
/// Hence, player 1 will never be the winner and you need to return False.
///
/// Example 2:
/// Input: [1, 5, 233, 7]
/// Output: True
/// Explanation: Player 1 first chooses 1. Then player 2 have to choose
/// between 5 and 7. No matter which number player 2 choose, player 1 can
/// choose 233.
/// Finally, player 1 has more score (234) than player 2 (12), so you need
/// to return True representing player1 can win.
///
/// Note:
/// 1.1 <= length of the array <= 20.
/// 2.Any scores in the given array are non-negative integers and will not
///   exceed 10,000,000.
/// 3.If the scores of both players are equal, then player 1 is still the
winner.
/// </summary>
bool LeetCode::predictTheWinner(vector<int>& nums)
{
    if (nums.empty()) return true;
    vector<vector<pair<int, int>>> sum(nums.size(), vector<pair<int,
int>>(nums.size(), pair<int, int>()));
    for (size_t step = 0; step < nums.size(); step++)
    {
        for (size_t i = 0; i < nums.size(); i++)
        {
            int first;
            int second;
            if (step == 0)
            {
                first = nums[i];
                second = 0;
                sum[i][i + step] = make_pair(first, second);
            }
            else if (i + step < nums.size())
            {
                first = max(nums[i] + sum[i + 1][i + step].second, nums[i +
step] + sum[i][i + step - 1].second);
                second = nums[i] + sum[i + 1][i + step].first + sum[i + 1]
[i + step].second - first;
                sum[i][i + step] = make_pair(first, second);
            }
        }
    }
    return (sum[0][nums.size() -1].first >= sum[0][nums.size() -
1].second);
}




/// <summary>
/// Leet code #471. Encode String with Shortest Length
/// </summary>
void LeetCode::findRepeatPattern(string s, size_t start, size_t end,
vector<vector<string>>& result)
```

```
{
    int length = end - start + 1;
    string str = s.substr(start, length);

    int count = 1;
    size_t next = start + length;
    while ((next + length <= s.size()) && (s.substr(next, length) == str))
    {
        count++;
        result[next][next + length - 1] = result[start][end];
        next += length;
        if (count > 1)
        {
            if (result[start][next - 1].empty())
            {
                result[start][next - 1] = s.substr(start, next - start);
            }
            string new_str = to_string(count) + "[" + result[start][end] +
"]";
            if (new_str.size() < result[start][next - 1].size())
            {
                result[start][next - 1] = new_str;
            }
        }
    }
    return;
}

/// <summary>
/// Leet code #471. Encode String with Shortest Length
///
/// Given a non-empty string, encode the string such that its encoded
/// length is the shortest.
/// The encoding rule is: k[encoded_string], where the encoded_string
/// inside the square brackets is being repeated exactly k times.
/// Note:
/// 1.k will be a positive integer and encoded string will not be empty or
///   have extra space.
/// 2.You may assume that the input string contains only lowercase English
///   letters. The string's length is at most 160.
/// 3.If an encoding process does not make the string shorter, then do not
///   encode it. If there are several solutions, return any of them is
fine.
///
/// Example 1:
/// Input: "aaa"
/// Output: "aaa"
/// Explanation: There is no way to encode it such that it is shorter than
/// the input string, so we do not encode it.
///
/// Example 2:
/// Input: "aaaaa"
/// Output: "5[a]"
/// Explanation: "5[a]" is shorter than "aaaaa" by 1 character.
///
/// Example 3:
/// Input: "aaaaaaaaaa"
/// Output: "10[a]"
```

```
/// Explanation: "a9[a]" or "9[a]a" are also valid solutions, both of them
/// have the same length = 5, which is the same as "10[a]".
///
/// Example 4:
/// Input: "aabcaabcd"
/// Output: "2[aabc]d"
/// Explanation: "aabc" occurs twice, so one answer can be "2[aabc]d".
///
/// Example 5:
/// Input: "abbbabbbcabbbabbbc"
/// Output: "2[2[abbb]c]"
/// Explanation: "abbbabbbc" occurs twice, but "abbbabbbc" can also be
encoded
/// to "2[abbb]c", so one answer can be "2[2[abbb]c]".
/// </summary>
string LeetCode::encode(string s)
{
    vector<vector<string>> result(s.size(), vector<string>(s.size()));

    for (size_t step = 0; step < s.size(); step++)
    {
        for (size_t i = 0; i < s.size(); i++)
        {
            if (i + step >= s.size())
            {
                break;
            }

            // we may process this range before.
            if (result[i][i + step].empty())
            {
                result[i][i + step] = s.substr(i, step + 1);
            }

            // less than 5 characters, no need to encode
            if (step >= 4)
            {
                // search for better option by using sub range
                for (size_t k = 0; k < step; k++)
                {
                    if (result[i][i + step].size() > result[i][i +
k].size() + result[i + k + 1][i + step].size())
                    {
                        result[i][i + step] = result[i][i + k] + result[i +
k + 1][i + step];
                    }
                }

            }
            findRepeatPattern(s, i, i + step, result);
        }
    }
    return result[0][s.size() - 1];
}
```

Dynamic Programming II

Lookback DP

The typical pattern for look back DP is when we calculate N-th result (or iteration), it depends on N-1th result (or iteration).

```
/// <summary>
/// Leet code #256. Paint House
///
/// There are a row of n houses, each house can be painted with one of the
/// three colors: red, blue or green. The cost of painting each house with
a
/// certain color is different. You have to paint all the houses such that
no
/// two adjacent houses have the same color.
/// The cost of painting each house with a certain color is represented by
a
/// n x 3 cost matrix. For example, costs[0][0] is the cost of painting
house
/// 0 with color red;
/// costs[1][2] is the cost of painting house 1 with color green, and so
on...
/// Find the minimum cost to paint all houses.
/// Note:
/// All costs are positive integers.
/// </summary>
/// <Analysis>
/// When we paint N-th house the result depends on the lowest cost we get
on
/// N-1th houses, we can evaluate every color in N-th house as long as that
/// we do not paint the N-1 th house in the same color.
/// This is a typical look back
/// </Analysis>
int LeetCode::minCost(vector<vector<int>>& costs)
{
    vector<vector<int>> sum;
    if (costs.size() == 0) return 0;
    for (size_t i = 0; i < costs.size(); i++)
    {
        sum.push_back(vector<int>(3));
        if (i == 0)
        {
            sum[i][0] = costs[i][0];
            sum[i][1] = costs[i][1];
            sum[i][2] = costs[i][2];
        }
        else
        {
            sum[i][0] = costs[i][0] + min(sum[i - 1][1], sum[i - 1][2]);
            sum[i][1] = costs[i][1] + min(sum[i - 1][0], sum[i - 1][2]);
            sum[i][2] = costs[i][2] + min(sum[i - 1][0], sum[i - 1][1]);
        }
    }
    return min(min(sum[costs.size() - 1][0], sum[costs.size() - 1][1]),
sum[costs.size() - 1][2]);
}
```

```
/// <summary>
/// Leet code #628. Maximum Product of Three Numbers
///
/// Given an integer array, find three numbers whose product is maximum
/// and output the maximum product.
///
/// Example 1:
/// Input: [1,2,3]
/// Output: 6
///
/// Example 2:
/// Input: [1,2,3,4]
/// Output: 24
///
/// Note:
/// 1.The length of the given array will be in range [3,10^4] and all
///   elements are in the range [-1000, 1000].
/// 2.Multiplication of any three numbers in the input won't exceed the
///   range of 32-bit signed integer.
/// </summary>
/// <Analysis>
/// The maximum product of N numbers depends on on miniunm or maximum
product
/// of N-1 numbers, the order of calculation does not matter, but make sure
not
/// include itself.
/// </Analysis>
int LeetCode::maximumProduct(vector<int>& nums)
{
    vector<pair<int, int>> product_array = vector<pair<int, int>>(3,
make_pair(1, 1));
    for (size_t i = 0; i < nums.size(); i++)
    {
        for (int k = (int)product_array.size() - 1; k >= 0; k--)
        {
            if (i <= (size_t)k)
            {
                product_array[k].first *= nums[i];
                product_array[k].second *= nums[i];
            }
            else if (k == 0)
            {
                product_array[k].first = min(product_array[k].first,
nums[i]);
                product_array[k].second = max(product_array[k].second,
nums[i]);
            }
            else
            {
                product_array[k].first = min(product_array[k].first,
min(nums[i] * product_array[k - 1].first, nums[i] * product_array[k -
1].second));
                product_array[k].second = max(product_array[k].second,
max(nums[i] * product_array[k - 1].first, nums[i] * product_array[k -
1].second));
            }
        }
    }
```

```
    return product_array[product_array.size() - 1].second;
}

/// <summary>
/// Leet code #660. Remove 9
/// Start from integer 1, remove any integer that contains 9 such as 9,
/// 19, 29...
///
/// So now, you will have a new integer sequence: 1, 2, 3, 4, 5, 6, 7,
/// 8, 10, 11, ...
///
/// Given a positive integer n, you need to return the n-th integer after
/// removing. Note that 1 will be the first integer.
///
/// Example 1:
/// Input: 9
/// Output: 10
/// Hint: n will not exceed 9 x 10^8.
/// </summary>
/// <Analysis>
/// First, we calculate how many digits we need to count until n-th number
which
/// qualify, we may over-count, then we calculate from most significant
digit
/// to the least significant digit.
/// The n digit number may include n-1 digits, n-2 digits until 1 digits,
consider
/// high digits are all zero. For n digits, the lower digits can also be
zero if
/// high digits are non-zero.
/// </Analysis>
int LeetCode::newInteger(int n)
{
    // remeber each digit of the result;
    vector<int> nums;
    // A[i] stands for how many number in  i dights without 9
    // S[i] stands for how many numbers totally until i digits without 9
    vector<int> A, S;
    // A[0] = 1, which is 0
    A.push_back(1);
    S.push_back(1);

    // we add zero in front, so count n + 1
    while (S.back() < n + 1)
    {
        A.push_back(8 * S.back());
        S.push_back(S.back() + A.back());
    }

    while (S.size() > 1)
    {
        S.pop_back();
        int digit = n / S.back();
        nums.push_back(digit);
        n -= digit * S.back();
    }

    int result = 0;
```

```
    for (size_t i = 0; i < nums.size(); i++)
    {
        result = result * 10 + nums[i];
    }
    return result;
}

/// <summary>
/// Leet code #279. Perfect Squares
/// Given a positive integer n, find the least number of perfect square
numbers
/// (for example, 1, 4, 9, 16, ...) which sum to n.
/// For example, given n = 12, return 3 because 12 = 4 + 4 + 4;
/// given n = 13, return 2 because 13 = 4 + 9.
/// </summary>
int LeetCode::numSquares(int n)
{
    vector<int> square_count(n+1, 0);
    vector<int> square;
    for (size_t i = 0; i <= (size_t)n; i++)
    {
        int r = square.size() * square.size();
        if ( r == i)
        {
            square_count[i] = 1;
            square.push_back(r);
        }
        else
        {
            for (size_t j = 1; j < square.size(); j++)
            {
                if (j == 1)
                {
                    square_count[i] = square_count[i - square[j]] + 1;
                }
                else
                {
                    square_count[i] = min(square_count[i], square_count[i -
square[j]] + 1);
                }
            }
        }
    }
    return square_count[n];
}
```

Carry forward

When you try to use carry forward, it should satisfy the following conditions.

1. When you are calculate i-th position, you already know on the limited future positions (j-th position where j > i) the i-th position will impact, so you avoid looking from 0 to j-1th position when you are in j-th position. This will reduce the O(N^2) complexity to O(N).
2. When you are at i-th position, and look back there may be too many combinations to calculate if you look backward, but if you push the

result to future and memorize the cheapest solution, it will save huge calculation effort.

So the carrying forward is not significantly different from looking back, just see which may to make the calculation easy. Carry forward will discard any duplicated, or expensive result at i-th position.

```
/// <summary>
/// Leet code #313. Super Ugly Number
/// Write a program to find the nth super ugly number.
/// Super ugly numbers are positive numbers whose all prime factors are in
/// the given prime list primes of size k. For example, [1, 2, 4, 7, 8, 13,
/// 14, 16, 19, 26, 28, 32]  is the sequence of the first 12 super ugly
/// numbers given primes = [2, 7, 13, 19] of size 4.
/// Note:
/// (1) 1 is a super ugly number for any given primes.
/// (2) The given numbers in primes are in ascending order.
/// (3) 0 < k ≤ 100, 0 < n ≤ 106, 0 < primes[i] < 1000.
/// </summary>
int LeetCode::nthSuperUglyNumber(int n, vector<int>& primes)
{
    vector<int> result;

    // the search track on the minimum value (the index in result list)
with
    // each prime as the factor.
    vector<int> search(primes.size(), 0);
    result.push_back(1);
    // loop until we have n result
    while (result.size() < (size_t)n)
    {
        int min_value = INT_MAX;
        int min_index = -1;

        for (size_t i = 0; i < primes.size(); i++)
        {
            int value = result[search[i]] * primes[i];

            if ((value < min_value) || (min_index == -1))
            {
                min_value = value;
                // not the real index, but the index in search list
matters, because
                // we update the pointer later
                min_index = i;
            }
        }
        if (min_index >= 0)
        {
            search[min_index]++;
            if (min_value > result.back())
            {
                result.push_back(min_value);
            }
        }
    }
    return result.back();
```

```
}

/// <summary>
/// Leet code #651. 4 Keys Keyboard
/// compare the new step with the existing stored set, if either position
/// (first first) or the step length (second value) win, we will insert the
/// new step, otherwise throw away.
/// In the same trip, we retire the loser in the set
/// </summary>
void LeetCode::insert_step(pair<int, int>&step, set<pair<int, int>>
&step_set)
{
    for (set<pair<int, int>>::iterator itr = step_set.begin(); itr !=
step_set.end();)
    {
        // new step lose, we return
        if (step.first <= itr->first && step.second <= itr->second)
        {
            return;
        }
        else if (step.first >= itr->first && step.second >= itr->second)
        {
            set<pair<int, int>>::iterator tmp = itr;
            tmp++;
            step_set.erase(itr);
            itr = tmp;
        }
        else
        {
            itr++;
        }
    }
    step_set.insert(step);
}

/// <summary>
/// Leet code #651. 4 Keys Keyboard
///
/// Key 1: (A): Prints one 'A' on screen.
/// Key 2: (Ctrl-A): Select the whole screen.
/// Key 3: (Ctrl-C): Copy selection to buffer.
/// Key 4: (Ctrl-V): Print buffer on screen appending it after what has
///        already been printed.
/// Now, you can only press the keyboard for N times (with the above four
/// keys), find out the maximum numbers of 'A' you can print on screen.
/// Example 1:
/// Input: N = 3
/// Output: 3
/// Explanation:
/// We can at most get 3 A's on screen by pressing following key sequence:
/// A, A, A
/// Example 2:
/// Input: N = 7
/// Output: 9
/// Explanation:
/// We can at most get 9 A's on screen by pressing following key sequence:
/// A, A, A, Ctrl A, Ctrl C, Ctrl V, Ctrl V
/// Note:
```

```
/// 1. 1 <= N <= 50
/// 2. Answers will be in the range of 32-bit signed integer.
/// </summary>
int LeetCode::maxA(int N)
{
    // the pair is (position, step);
    vector<set<pair<int, int>>> key_map(N+1);

    key_map[1].insert(make_pair(1, 1));
    for (int i = 1; i < N; i++)
    {
        for (auto step : key_map[i])
        {
            // handle A or Ctrl V
            pair<int, int> new_step = make_pair(step.first + step.second,
step.second);
            insert_step(new_step, key_map[i + 1]);
            if (i + 3 <= N)
            {
                // handle  Ctrl A, Ctrl C, Ctrl V
                new_step = make_pair(step.first + step.first, step.first);
                insert_step(new_step, key_map[i + 3]);
            }
        }
    }
    int result = INT_MIN;
    for (auto t : key_map[N])
    {
        result = max(result, t.first);
    }
    return result;
}
```

Two dimension DP

The two dimension DP is used to calculate a N-long vector by first calculate length of sub vector, (which is (0,0), (1,1), (2,2)… (n-1, n-1)), then length of 2 (which is (0,2), (1,3)…), until length of N. when you calculate length of X sub vector, you can always divide it into two more short sub vectors which is already calculated in the previous steps. In this case the two dimension DP is also known as build solution from bottom to top.

The two dimension DP normally has a symmetrical solution which is build solution from top to bottom, by using backtracking with divide and conquer method. This is to say if I want to calculate result(0, n), we can divide it into two small parts, result(0, i) and result(i, n), and call the function recursively, we need to remember result (i,j) once calculated.

```
/// <summary>
/// Leet code #664. Strange Printer
///
/// There is a strange printer with the following two special requirements:
/// The printer can only print a sequence of the same character each time.
///
/// At each turn, the printer can print new characters starting from and
/// ending at any places, and will cover the original existing characters.
```

```
/// Given a string consists of lower English letters only, your job is to
/// count the minimum number of turns the printer needed in order to print
/// it.
///
/// Example 1:
///
/// Input: "aaabbb"
/// Output: 2
/// Explanation: Print "aaa" first and then print "bbb".
///
/// Example 2:
/// Input: "aba"
/// Output: 2
/// Explanation: Print "aaa" first and then print "b" from the second
/// place of the string, which will cover the existing character 'a'.
///
/// Hint: Length of the given string will not exceed 100.
/// </summary>
int LeetCode::strangePrinter(string s)
{
    if (s.empty()) return 0;
    vector<vector<int>> dp(s.size(), vector<int>(s.size(), INT_MAX));
    for (size_t len = 0; len < s.size(); len++)
    {
        for (size_t i = 0; i < s.size(); i++)
        {
            if (i + len >= s.size()) break;
            if (len == 0)
            {
                dp[i][i + len] = 1;
                continue;
            }
            for (size_t j = i; j < i + len; j++)
            {
                int count = dp[i][j] + dp[j + 1][i + len];
                if (s[j] == s[i + len]) count--;

                dp[i][i + len] = min(dp[i][i + len], count);
            }
        }
    }
    return dp[0][s.size() - 1];
}
```

Cross Apply

The cross apply is straight forward, which only applies on two strings where if we edition one characters and make it become another, remember the dp vector is two dimension and the length is str.size() +1, considering empty string is an option.

```
/// <summary>
/// Leet code #583. Delete Operation for Two Strings
///
/// Given two words word1 and word2, find the minimum number of steps
required
/// to make word1 and word2 the same, where in each step you can delete one
```

```
/// character in either string.
/// Example 1:
/// Input: "sea", "eat"
/// Output: 2
/// Explanation: You need one step to make "sea" to "ea" and another step
to
/// make "eat" to "ea".
/// Note:
/// 1. The length of given words won't exceed 500.
/// 2. Characters in given words can only be lower-case letters.
/// </summary>
int LeetCode::minDeleteDistance(string word1, string word2)
{
    vector<vector<int>> dp(word1.size() + 1, vector<int>(word2.size() +
1));
    for (size_t i = 0; i < word1.size() + 1; i++)
    {
        for (size_t j = 0; j < word2.size() + 1; j++)
        {
            if (i == 0)
            {
                if (j == 0) dp[i][j] = 0;
                else dp[i][j] = dp[i][j - 1] + 1;
            }
            else if (j == 0)
            {
                dp[i][j] = dp[i - 1][j] + 1;
            }
            else
            {
                if (word1[i - 1] == word2[j - 1])
                {
                    dp[i][j] = dp[i - 1][j - 1];
                }
                else
                {
                    dp[i][j] = dp[i - 1][j - 1] + 2;
                }
                dp[i][j] = min(dp[i][j], dp[i - 1][j] + 1);
                dp[i][j] = min(dp[i][j], dp[i][j - 1] + 1);
            }
        }
    }
    return dp[word1.size()][word2.size()];
}
```

Binary Tree

Binary Tree is one of the common interview questions, but in general Binary tree questions can be divided into three categories based on their solution. They are:

1. Preorder, inroder, postorder traversal.
2. Level based traversal.
3. Find the predecessor and successor of a node.

The core algorithm in the above 3 scenarios are DFS (backtracking), BFS(queue based) and Stack explore.

DFS search

For DFS algorithm, we can use the recursive call to traverse the tree, we need to remember all the intermediate result by using the reference variable.

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);
}
```

DFS examples

Note:

1. A key coding trick in binary tree DFS search is that you should always differentiate the scenario on empty node, leaf nodes and parent nodes, otherwise it may have duplicated result.

```
/// <summary>
/// Leet code #113. Path Sum II
/// </summary>
void LeetCode::pathSum(TreeNode* root, int sum, vector<int>&path,
vector<vector<int>> &result)
{
    if (root == nullptr)
    {
        return;
    }
    path.push_back(root->val);
    if ((root->left == nullptr) && (root->right == nullptr))
    {
        if (root->val == sum) result.push_back(path);
    }
    else
```

```
    {
        if (root->left != nullptr) pathSum(root->left, sum - root->val,
path, result);
        if (root->right != nullptr) pathSum(root->right, sum - root->val,
path, result);
    }
    path.pop_back();
}

/// <summary>
/// Leet code #113. Path Sum II
/// Given a binary tree and a sum, find all root-to-leaf paths where each
/// path's sum equals the given sum.
/// For example:
/// Given the below binary tree and sum = 22,
///          5
///         / \
///        4   8
///       /   / \
///      11  13  4
///     /  \    / \
///    7    2  5   1
/// return
/// [
///   [5,4,11,2],
///   [5,8,4,5]
/// ]
/// </summary>
vector<vector<int>> LeetCode::pathSum(TreeNode* root, int sum)
{
    vector<int>path;
    vector<vector<int>> result;
    pathSum(root, sum, path, result);
    return result;
}


/// <summary>
/// Preorder traverse with the path passed on the way to check the sum.
/// </summary>
int LeetCode::pathSumIII(TreeNode* root, vector<int>& path, int sum)
{
    int count = 0;
    if (root == nullptr) return 0;
    path.push_back(root->val);
    int total = 0;
    // from bottom to search up is to make sure we do not
    // have duplicate
    for (int i = (int)path.size() - 1; i >= 0; i--)
    {
        total += path[i];
        if (total == sum) count++;
    }
    count += pathSumIII(root->left, path, sum);
    count += pathSumIII(root->right, path, sum);
    path.pop_back();
    return count;
}
```

```
/// <summary>
/// Leet code #437. Path Sum III
/// You are given a binary tree in which each node contains an integer
value.
/// Find the number of paths that sum to a given value.
/// The path does not need to start or end at the root or a leaf,
/// but it must go downwards (traveling only from parent nodes to child
nodes).
/// The tree has no more than 1,000 nodes and the values are in the range
/// -1,000,000 to 1,000,000.
/// Example:
/// root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8
///         10
///        /  \
///       5   -3
///      / \    \
///     3   2   11
///    / \   \
///   3  -2   1
///
/// Return 3. The paths that sum to 8 are:
///
/// 1.  5 -> 3
/// 2.  5 -> 2 -> 1
/// 3. -3 -> 11
/// </summary>
int LeetCode::pathSumIII(TreeNode* root, int sum)
{
    vector<int> path;
    return pathSumIII(root, path, sum);
}

/// <summary>
/// Leet code #250. Count Univalue Subtrees
/// </summary>
bool LeetCode::countUnivalSubtrees(TreeNode* root, int& value, int& count)
{
    if (root == nullptr) return true;
    int left_value = 0, left_count = 0;
    bool left_unival = true;
    if (root->left != nullptr)
    {
        left_unival = countUnivalSubtrees(root->left, left_value,
left_count);
        if (root->val != left_value)
        {
            left_unival = false;
        }
    }

    int right_value = 0, right_count = 0;
    bool right_unival = true;
    if (root->right != nullptr)
    {
        right_unival = countUnivalSubtrees(root->right, right_value,
right_count);
        if (root->val != right_value)
```

```
        {
            right_unival = false;
        }
    }
    count = left_count + right_count;
    value = root->val;
    if (left_unival && right_unival)
    {
        count++;
        return true;
    }
    else
    {
        return false;
    }
}

/// <summary>
/// Leet code #250. Count Univalue Subtrees
///
/// Given a binary tree, count the number of uni-value subtrees.
/// A Uni-value subtree means all nodes of the subtree have the same value.
///
/// For example:
/// Given binary tree,
///              5
///             / \
///            1   5
///           / \   \
///          5   5   5
///
/// return 4.
/// </summary>
int LeetCode::countUnivalSubtrees(TreeNode* root)
{
    int value = 0;
    int count = 0;
    countUnivalSubtrees(root, value, count);
    return count;
}

/// <summary>
/// Leet code #333. Largest BST Subtree
/// </summary>
bool LeetCode::checkBSTSubtree(TreeNode* root, int& min_val, int& max_val,
int& size)
{
    if (root == nullptr)
    {
        size = 0;
        return true;
    }
    int left_min = root->val, left_max = root->val, left_size = 0;
    bool left_BSTSubtree = true;
    if (root->left != nullptr)
    {
        left_BSTSubtree = checkBSTSubtree(root->left, left_min, left_max,
left_size);
```

```
        if (left_BSTSubtree) left_BSTSubtree = root->val > left_max;
    }
    int right_min = root->val, right_max = root->val, right_size = 0;
    bool right_BSTSubtree = true;
    if (root->right != nullptr)
    {
        right_BSTSubtree = checkBSTSubtree(root->right, right_min,
right_max, right_size);
        if (right_BSTSubtree) right_BSTSubtree = root->val < right_min;
    }
    min_val = min(left_min, right_min); max_val = max(right_min,
right_max);
    if (left_BSTSubtree && right_BSTSubtree)
    {
        size = 1 + left_size + right_size;
        return true;
    }
    else
    {
        size = max(left_size, right_size);
        return false;
    }
}

/// <summary>
/// Leet code #333. Largest BST Subtree
///
/// Given a binary tree, find the largest subtree which is a Binary Search
Tree
/// (BST), where largest means subtree with largest number of nodes in it.
/// Note:
/// A subtree must include all of its descendants.
/// Here's an example:
///
///     10
///    /  \
///   5   15
///  / \    \
/// 1   8    7
/// The Largest BST Subtree in this case is the highlighted one.
/// The return value is the subtree's size, which is 3.
/// Hint:
/// 1. You can recursively use algorithm similar to 98. Validate Binary
/// Search Tree at each node of the tree,
/// which will result in O(nlogn) time complexity.
/// Follow up:
/// Can you figure out ways to solve it with O(n) time complexity?
/// </summary>
int LeetCode::largestBSTSubtree(TreeNode* root)
{
    int min_val, max_val, size;
    checkBSTSubtree(root, min_val, max_val, size);
    return size;
}
```

Sum with and without self

The following two problems try to calculate greater value between the sum of the root value and the skip level sum or the sum of its two children.

Both of problems belongs to the DFS category. Please notice that the sum with self should also consider sum without self, get the maximum one. This is because it may happen on one side with the child root, on the side without the child root.

```
/// <summary>
/// Leet code #337. House Robber III
/// </summary>
void LeetCode::robIII(TreeNode* root, int& rob_with_root, int&
rob_without_root)
{
    rob_with_root = 0;
    rob_without_root = 0;
    if (root != nullptr)
    {
        int left_rob_with_root = 0, left_rob_without_root = 0;
        robIII(root->left, left_rob_with_root, left_rob_without_root);
        int right_rob_with_root = 0, right_rob_without_root = 0;
        robIII(root->right, right_rob_with_root, right_rob_without_root);
        rob_with_root = root->val + left_rob_without_root +
right_rob_without_root;
        rob_without_root = left_rob_with_root + right_rob_with_root;
        rob_with_root = max(rob_with_root, rob_without_root);
    }
}

/// <summary>
/// Leet code #337. House Robber III
/// The thief has found himself a new place for his thievery again.
/// There is only one entrance to this area,
/// called the "root." Besides the root, each house has one and only
/// one parent house. After a tour, the smart
/// thief realized that "all houses in this place forms a binary tree".
/// It will automatically contact the police
/// if two directly-linked houses were broken into on the same night.
/// Determine the maximum amount of money the thief can rob tonight
/// without alerting the police.
/// Example 1:
///     3
///    / \
///   2   3
///    \   \
///     3   1
/// Maximum amount of money the thief can rob = 3 + 3 + 1 = 7.
/// Example 2:
///        3
///     / \
///    4   5
///   / \   \
///  1   3   1
/// Maximum amount of money the thief can rob = 4 + 5 = 9.
/// </summary>
int LeetCode::robIII(TreeNode* root)
{
    int rob_with_root = 0;
    int rob_without_root = 0;
    robIII(root, rob_with_root, rob_without_root);
    return rob_with_root;
```

```
}

/// <summary>
/// Leet code #124. Binary Tree Maximum Path Sum
/// </summary>
void LeetCode::maxPathSum(TreeNode* root, int &max_path_sum,
int&max_path_loop)
{
    if (root == nullptr)
    {
        max_path_sum = 0;
        max_path_loop = INT_MIN;
    }
    else
    {
        int max_path_sum_left, max_path_loop_left;
        maxPathSum(root->left, max_path_sum_left, max_path_loop_left);
        int max_path_sum_right, max_path_loop_right;
        maxPathSum(root->right, max_path_sum_right, max_path_loop_right);

        max_path_sum = max(max_path_sum_left + root->val,
max_path_sum_right + root->val);
        max_path_sum = max(max_path_sum,  root->val);
        max_path_loop = max(max_path_loop_left, max_path_loop_right);
        max_path_loop = max(max_path_loop, root->val + max_path_sum_left +
max_path_sum_right);
        max_path_loop = max(max_path_loop, max_path_sum);
    }
}

/// <summary>
/// Leet code #124. Binary Tree Maximum Path Sum
/// Given a binary tree, find the maximum path sum.
/// For this problem, a path is defined as any sequence of nodes from some
/// starting node to any node in the tree along the parent-child
connections.
/// The path must contain at least
/// one node and does not need to go through the root.
/// For example:
/// Given the below binary tree,
///     1
///    / \
///   2   3
/// Return 6.
/// Explanation:
/// The max_path must come from the left direct path + self, the right
direct path + self
/// and the maximum left loop path and maximum right loop path.
/// </summary>
int LeetCode::maxPathSum(TreeNode* root)
{
    int max_path_loop = 0;
    int max_path_sum = 0;
    maxPathSum(root, max_path_sum, max_path_loop);
    return max_path_loop;
}
```

BFS search

BFS Search normally handle the level based traversal. As any algorithm in BFS search, you need a queue. The steps are as below.

1. Push the root to the queue.
2. For every node in the queue **now**, pop up and process the node. This means if during the processing, the new nodes may be added to the queue, but they are in the next level, they are the children, they should not be processed in this batch. This means you may need to save the size of the queue if you want to strictly differentiate the levels.
3. Loop #2 until no one is in the queue.

```
/// <summary>
/// Leet code #226. Invert Binary Tree
/// Invert a binary tree.
///      4
///    /   \
///   2     7
///  / \   / \
/// 1   3 6   9
/// to
///      4
///    /   \
///   7     2
///  / \   / \
/// 9   6 3   1
/// Trivia:
/// This problem was inspired by this original tweet by Max Howell:
/// Google: 90% of our engineers use the software you wrote (Homebrew),
/// but you can’t invert a binary tree on a whiteboard so fuck off.
/// </summary>
TreeNode* LeetCode::invertTree(TreeNode* root)
{
    queue<TreeNode *> node_queue;
    if (root != nullptr)
    {
        node_queue.push(root);
    }
    while (!node_queue.empty())
    {
        size_t size = node_queue.size();
        for (size_t i = 0; i < size; i++)
        {
            TreeNode * node = node_queue.front();
            node_queue.pop();
            swap(node->left, node->right);
            if (node->left != nullptr)
            {
                node_queue.push(node->left);
            }
            if (node->right != nullptr)
            {
                node_queue.push(node->right);
            }
        }
    }
    return root;
}
```

```
/// <summary>
/// Leet code 297. Serialize and Deserialize Binary Tree
/// Serialization is the process of converting a data structure or object
into a sequence of bits
/// so that it can be stored in a file or memory buffer, or transmitted
across a network connection link to
/// be reconstructed later in the same or another computer environment.
///
/// Design an algorithm to serialize and deserialize a binary tree. There
is no restriction on how your
/// serialization/deserialization algorithm should work. You just need to
ensure that a binary tree can be
/// serialized to a string and this string can be deserialized to the
original tree structure.
/// For example, you may serialize the following tree
///    1
///   / \
///  2   3
///     / \
///    4   5
///
/// as "[1,2,3,null,null,4,5]", just the same as how LeetCode OJ serializes
a binary tree.
/// You do not necessarily need to follow this format, so please be
creative and come up with different approaches yourself.
/// Note: Do not use class member/global/static variables to store states.
Your serialize and deserialize algorithms should be stateless.
/// </summary>
/// <summary>
/// Encodes a tree to a single string.
/// </summary>
/// <param name="root">the root</param>
/// <returns>The string</returns>
string LeetCode::serialize(TreeNode* root)
{
    string result = "";
    queue<TreeNode *> queue;
    if (root != nullptr) queue.push(root);
    while (!queue.empty())
    {
        TreeNode * node = queue.front();
        queue.pop();
        if (!result.empty()) { result.push_back(','); }
        if (node == nullptr)
        {
            result.append("null");
        }
        else
        {
            result.append(std::to_string(node->val));
            queue.push(node->left);
            queue.push(node->right);
        }
    }
    while (true)
    {
```

```
        if ((result.size() > 4) && (result.substr(result.size() - 4) ==
"null"))
        {
            result.erase(result.size() - 4);
        }
        else if ((result.size() > 1) && (result.substr(result.size() - 1)
== ","))
        {
            result.erase(result.size() - 1);
        }
        else
        {
            break;
        }
    }
    return "[" + result + "]";
}

/// <summary>
/// Decodes your encoded data to tree.
/// </summary>
/// <param name="data">the string data</param>
/// <returns>The root</returns>
TreeNode* LeetCode::deserialize(string data)
{
    queue<TreeNode *> input_queue;
    queue<TreeNode *> output_queue;
    string number;
    for (size_t i = 0; i < data.size(); i++)
    {
        if ((data[i] == '[') || (data[i] == ',') || data[i] == ']' ||
isspace(data[i]))
        {
            if (number.size() != 0)
            {
                if (number == "null")
                {
                    input_queue.push(nullptr);
                }
                else
                {
                    input_queue.push(new TreeNode(std::stoi(number)));
                }
                number.clear();
            }
        }
        else
        {
            number.push_back(data[i]);
        }
    }
    TreeNode *root = nullptr;
    TreeNode *node = nullptr;
    while (!input_queue.empty())
    {
        if (output_queue.empty())
        {
            root = input_queue.front();
```

```
            input_queue.pop();
            node = root;
            output_queue.push(node);
        }
        else
        {
            node = output_queue.front();
            output_queue.pop();
            if (node != nullptr)
            {
                if (!input_queue.empty())
                {
                    node->left = input_queue.front();
                    input_queue.pop();
                }
                if (!input_queue.empty())
                {
                    node->right = input_queue.front();
                    input_queue.pop();
                }
                output_queue.push(node->left);
                output_queue.push(node->right);
            }
        }
    }
    return root;
}
```

Predecessor and Successor

To find the predecessor, the common steps are below.

1. Search from top to bottom, push the path to the stack,
2. If the target node is found, pop it out of stack
3. check if it has left child, if the answer is yes, **greedily** look for its right child and push the path into stack, until there is no right child, the stack top is the predecessor.
4. If we want to find the next predecessor, we simply pop up a node from stack, repeat step #3.

The similar way can be used to find successor, just switch the left child and right child in the steps above.

```
/// <summary>
/// Leet code #272. Closest Binary Search Tree Value II
/// </summary>
vector<int> LeetCode::getPredecessor(stack<TreeNode*> left_stack, int k)
{
    vector<int> result;
    while (!left_stack.empty())
    {
        TreeNode * node = left_stack.top();
        left_stack.pop();
        result.push_back(node->val);

        if (result.size() == k) break;
        node = node->left;
        while (node != nullptr)
```

```
        {
            left_stack.push(node);
            node = node->right;
        }
    }
    return result;
}

/// <summary>
/// Leet code #272. Closest Binary Search Tree Value II
/// </summary>
vector<int> LeetCode::getSuccessor(stack<TreeNode*> right_stack, int k)
{
    vector<int> result;
    while (!right_stack.empty())
    {
        TreeNode * node = right_stack.top();
        right_stack.pop();
        result.push_back(node->val);

        if (result.size() == k) break;
        node = node->right;
        while (node != nullptr)
        {
            right_stack.push(node);
            node = node->left;
        }
    }
    return result;
}

/// <summary>
/// Leet code #272. Closest Binary Search Tree Value II
///
/// Given a non-empty binary search tree and a target value, find k values
in the BST that are closest to the target.
/// Note:
/// Given target value is a floating point.
/// You may assume k is always valid, that is: k ≤ total nodes.
/// You are guaranteed to have only one unique set of k values in the BST
that are closest to the target.
/// Follow up:
/// Assume that the BST is balanced, could you solve it in less than O(n)
runtime (where n = total nodes)?
/// Hint:
/// 1.Consider implement these two helper functions:
/// i.getPredecessor(N), which returns the next smaller node to N.
/// ii.getSuccessor(N), which returns the next larger node to N.
/// 2.Try to assume that each node has a parent pointer, it makes the
problem much easier.
/// 3.Without parent pointer we just need to keep track of the path from
the root to the current node using a stack.
/// 4.You would need two stacks to track the path in finding predecessor
and successor node separately.
/// </summary>
vector<int> LeetCode::closestKValues(TreeNode* root, double target, int k)
{
    vector<int> result;
```

```
    stack<TreeNode*> left_stack;
    stack<TreeNode*> right_stack;
    TreeNode *node = root;
    while (node != nullptr)
    {
        if (target <= node->val)
        {
            right_stack.push(node);
            node = node->left;
        }
        else if (target > node->val)
        {
            left_stack.push(node);
            node = node->right;
        }
    }
    vector<int> left_values = getPredecessor(left_stack, k);
    vector<int> right_values = getSuccessor(right_stack, k);
    int left_index = 0, right_index = 0;
    while (left_index < (int)left_values.size() || right_index <
(int)right_values.size())
    {
        if (result.size() == k) break;
        if (left_index == left_values.size())
        {
            result.push_back(right_values[right_index]);
            right_index++;
        }
        else if (right_index == right_values.size())
        {
            result.push_back(left_values[left_index]);
            left_index++;
        }
        else if (abs(left_values[left_index] - target) <
abs(right_values[right_index] - target))
        {
            result.push_back(left_values[left_index]);
            left_index++;
        }
        else
        {
            result.push_back(right_values[right_index]);
            right_index++;
        }
    }
    return result;
}
```

Complex Example

#99 Recover binary tree

https://leetcode.com/problems/recover-binary-search-tree

```
/// <summary>
/// Find the two disordered nodes in the binary search tree
/// </summary>
void LeetCode::recoverTree(TreeNode* root, TreeNode* &min_node, TreeNode*
&max_node,
```

```
    TreeNode* &first, TreeNode* &second)
{
    if (root == nullptr) return;
    TreeNode* left_min = root;
    TreeNode* left_max = root;
    TreeNode* right_min = root;
    TreeNode* right_max = root;
    if ((root->left == nullptr) && (root->right == nullptr))
    {
        min_node = root;
        max_node = root;
        return;
    }
    if (root->left != nullptr)
    {
        recoverTree(root->left, left_min, left_max, first, second);
    }
    if (root->right != nullptr)
    {
        recoverTree(root->right, right_min, right_max, first, second);
    }

    if (left_max->val > root->val)
    {
        first = left_max;
        second = root;
    }
    if (root->val > right_min->val)
    {
        if (left_max->val <= root->val) first = root;
        second = right_min;
    }

    min_node = left_min;
    if (root->val < min_node->val) min_node = root;
    if (right_min->val < min_node->val) min_node = right_min;

    max_node = left_max;
    if (root->val > max_node->val) max_node = root;
    if (right_max->val > max_node->val) max_node = right_max;
}

/// <summary>
/// Leet code #99. Recover Binary Search Tree
/// Two elements of a binary search tree (BST) are swapped by mistake.
/// Recover the tree without changing its structure.
/// Note:
/// A solution using O(n) space is pretty straight forward. Could you
devise a constant space solution?
/// </summary>
void LeetCode::recoverTree(TreeNode* root)
{
    TreeNode *min_node = nullptr, *max_node = nullptr;
    TreeNode *first = nullptr, *second = nullptr;
    recoverTree(root, min_node, max_node, first, second);
    if ((first != nullptr) && (second != nullptr))
    {
        swap(first->val, second->val);
```

```
    }
}

/// <summary>
/// Find the two disordered nodes in the binary search tree
/// </summary>
void LeetCode::recoverTreeII(TreeNode* root, TreeNode* &prev,
                             TreeNode* &first, TreeNode* &second)
{
    if (root == nullptr) return;
    if (root->left != nullptr)
    {
        recoverTreeII(root->left, prev, first, second);
    }
    if ((prev != nullptr) && (prev->val > root->val))
    {
        if (first == nullptr)
        {
            first = prev;
        }
        second = root;
    }
    prev = root;
    if (root->right != nullptr)
    {
        recoverTreeII(root->right, prev, first, second);
    }
}

/// <summary>
/// Leet code #99. Recover Binary Search Tree
/// Two elements of a binary search tree (BST) are swapped by mistake.
/// Recover the tree without changing its structure.
/// Note:
/// A solution using O(n) space is pretty straight forward. Could you
devise a constant space solution?
/// </summary>
void LeetCode::recoverTreeII(TreeNode* root)
{
    TreeNode *prev = nullptr, *first = nullptr, *second = nullptr;
    recoverTreeII(root, prev, first, second);
    if ((first != nullptr) && (second != nullptr))
    {
        swap(first->val, second->val);
    }
}
```

Comparison between C++ and Python

Unless you are a frontend programmer, the most three popular programming languages are C++, Java and Python. Because C++ and Java are very similar, so this chapter will focus on the comparison between C++ and Python. It only list the major features, so should not be considered as a full programming guide

DataType

Category	C++	Python
Data type	int (signed 32 bits), unsigned int, size_t (unsigned int), long long (64 bits for windows platform), unsigned long log, float and double bool = true / false char * or string (STL)	numeric, string and bool bool = True / False
variable	Must be declared before use, Must be assigned as a default value before use, unless it is a class object has default value. Can not change type after declared	No need to declare Must give default value before use Can change type after declared
math	+, -, * /, %, pow(), log()	+, -, *, /, **, log()
bool	true / false <, >, <=, >=, ==,=!, and, or, not	True / False <, >, <=, >=, ==, !=, &&, ||, !
Array operation	indexing [ ] membership : length size() (STL)	indexing [ ] concatenation + repetition * membership in length len slicing [:]
Array	vector<int> a_list = {1, 2, 3}; my_list.find(item) reverse(a_list.begin(), a_list.end()) sort(a_list.begin(), a_list.end())	a_list = [1, 2, True, 3] a_list.append(item) a_list.insert(i, item) a_list.pop() a_list.pop(i) a_list.sort() a_list.reverse() dele a_list[i] a_list.index(item) a_list.count(item) a_list.remove(item) a_list(range(5, 10))
Hashtable	unordered_map<int, int>, unordered_set<int>	
Sorted table	map<int, int>, set<int>	

Pointer	TreeNode *, ListNode *	No pointer, every object is a reference
Passing value or reference	Why passing reference: 1. avoid copy class object. 2. return many variables (update input) When not passing reference: 1. Any pointer, such as linked list, tree, avoid getting lost.	
sort	sort(array.begin(), array.end());	
reverse	reverse(array.begin(), array.end());	

customized comparator	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());	