- LeetCode and LintCode Problems Summary
  - Time Complexity Analysis
  - DFS
    - Generate Parentheses
    - Validate Binary Search Tree
    - Symmetric Tree
    - Sum Root to Leaf Numbers
    - Path Sum (Classic) / Path Sum II
    - Minimum Depth of Binary Tree
    - Maximum Depth of Binary Tree
    - Same Tree
    - Recover Binary Search Tree (Hard)
    - Number of Islands
    - Flatten Binary Tree to Linked List (Hard)
    - Convert Sorted List to Binary Search Tree (Find the middle node of a linkedlist within a range)
    - Convert Sorted Array to Binary Search Tree
    - Construct Binary Tree from Preorder and Inorder Traversal
    - Construct Binary Tree from Inorder and Postorder Traversal
    - Clone Graph (Graph, Check Again!)
    - Binary Tree Paths (Classic)
    - House Robber III
    - Balanced Binary Tree
    - Binary Tree Maximum Path Sum (Hard) (Check Again!)
  - BFS
    - Populating Next Right Pointers in Each Node I / II (Hard)
    - Binary Tree Right Side View
    - Surrounded Regions
  - Backtracking / Search - Advanced (DFS / BFS)
    - Palindrome Partitioning (Classic!!!)
    - Combinations
    - Combination Sum / Combination Sum II / Combination Sum III (Classic!!!)
    - Letter Combinations of a Phone Number
    - N-Queens / N-Queens II (Classic!!!)
    - Subsets / Subsets II
    - Permutations / Permutations II
    - Word Ladder I / Word Ladder II
    - Topological Sorting / Course Schedule / Course Schedule II
    - Minimum Height Trees
    - Word Search
    - Word Search II
    - Restore IP Address
    - Word Break II
  - Linked List
    - Merge Two Sorted Lists (Classic)
    - Merge k Sorted Lists
    - Linked List Cycle / Linked List Cycle II
    - Rotate List
    - Insertion Sort List
    - Partition list
    - Sort List

- Remove Duplicates from Sorted List / Remove Duplicates from Sorted List II
- Palindrome Linked List
- Reverse Linked List
- Reverse Linked List II
- Remove Linked List Elements
- Intersection of Two Linked Lists

Array / String

- H-Index / H-Index II
- Maximum Gap
- Minimum Window Substring
- Longest Palindromic Substring
- Continuous Subarray Sum
- Continuous Subarray Sum II
- Submatrix Sum
- Bounding Box Sum
- Longest Substring Without Repeating Characters
- Longest Substring with At Most K Distinct Characters
- Subarray Sum (Classic)
- Minimum Size Subarray Sum
- Remove Duplicates from Sorted Array
- Partition Array (Classic!!! Must Check Again!!!)
    - QuickSort Implementation
- Sort Colors
- Kth Largest Element in an Array
- Sort Letters by Case
- Nuts & Bolts Problem
- Trapping Rain Water
- Container With Most Water
- Median of Two Sorted Arrays
- Maximum Subarray / Minimum Subarray
- Count and Say
- Interleaving Positive and Negative Numbers
- Product of Array Except Self
- Largest Number
- Next Permutation
- Gas Station
- Rotate Array
- Reverse Words in a String
- Largest Number
- Rotate Image
- Spiral Matrix
- Group Anagrams
- Compare Version Numbers
- ZigZag Conversion
- Implement strStr()
- Word Pattern
- Shortest Palindrome (Hard!!! Check Again!!!)
- Candy
- Merge Intervals
- Insert Interval

- Binary Search
  - Subarray Sum II
  - Search for a Range
  - Search in Rotated Sorted Array (Classic!!!)
  - Find Minimum in Rotated Sorted Array
  - Find Minimum in Rotated Sorted Array II
  - Find the Duplicate Number
  - Find Peak Element
  - Find Peak Element II
  - H-Index II (Classic!!!)
  - Sqrt(x)
  - Search a 2D Matrix
- Binary Tree
  - Binary Tree Paths
  - Lowest Common Ancestor
  - Inorder Successor in Binary Search Tree
  - Binary Search Tree Iterator
  - Binary Tree Preorder Traversal
  - Binary Tree Inorder Traversal
  - Kth Smallest Element in a BST
  - Search Range in Binary Search Tree
  - Binary Tree Postorder Traversal
  - Binary Tree Zigzag Level Order Traversal
  - Convert Sorted Array to Binary Search Tree With Minimal Height
  - Segment Tree Build / Segment Tree Build II
  - Segment Tree Query / Segment Tree Query II
  - Segment Tree Modify
  - Count of Smaller Number before itself
  - Interval Minimum Number
  - Interval Sum / Interval Sum II
  - Range Sum Query - Mutable
  - Count Complete Tree Nodes
- Dynamic Programming
  - Stone Game
  - Jump Game / Jump Game II
  - Climbing Stairs
  - House Robber, House Robber II (Check Again!!!)
  - Minimum Path Sum, Unique Paths, Unique Paths II
  - Unique Binary Search Trees, Unique Binary Search Trees II
  - Decode Ways
  - Perfect Squares
  - Longest Increasing Subsequence (O(NlogN) solution, Check again! Classic)
  - Longest Common Substring (Classic)
  - Longest Common Subsequence
  - Maximum Subarray
  - Maximum Product Subarray
  - Triangle (Classic)
  - Ugly Number II (Check Again!!!)
  - Best Time to Buy and Sell Stocks I/II/III/IV (Hard, Check Again!!!)
  - Maximal Square
  - Longest Valid Parentheses (Hard, Check Again!!!)

- Wildcard Matching (Hard, Rolling Array, Check Again!!!)
- Regular Expression Matching
- Maximal Rectangle (Hard, Check Again!!!)
- Palindrome Partitioning II (Hard, Classic, Check Again!!!)
- Scramble String (Hard, Check Again!!!)
- Interleaving String (Classic)
- Edit Distance (Classic)
- Word Break
- Dungeon Game
- Distinct Subsequences (Classic)
- Coin Change (Hard!!!)
- Coins in a Line
- Coins in a Line II
- Coins in a Line III
- Backpack
- Minimum Adjustment Cost
- k Sum
- Coin Change

Data Structure (Hash / Heap / Stack / Queue)

- Longest Consecutive Sequence
- Largest Rectangle in Histogram
- Stack Sorting
- Max Tree
- Heapify
- Number of Airplanes in the Sky
- Insert Interval
- Number of Islands
- Add and Search Word - Data structure design
- Trapping Rain Water II
- Sliding Window Median, Data Stream Median
- Sliding Window Maximum
- Building Outline
- Kth Smallest Number in Sorted Matrix
- Valid Parentheses
- Animal Shelter
- Kth Smallest Sum In Two Sorted Arrays
- Delete Digits
- Contains Duplicate II
- Contains Duplicate III
- Simplify Path
- Evaluate Reverse Polish Notation
- Basic Calculator / Basic Calculator II

Math & Bit-Manipulation

- Gray Code (Classic)
- Plus One
- A + B Problem
- Add Binary
- Count 1 in Binary
- Pow(x, n)
- Update Bits
- Tailing Zeros
- Number of Digit One

- Palindrome Number
- Single Number
- Single Number II
- Single Number III
- Missing Number
- Binary Representation
- Fast Power
- Fraction to Recurring Decimal
- Divide Two Integers
- Multiply Strings
- String to Integer (atoi)
- Excel Sheet Column Title
- Excel Sheet Column Number
- Integer to Roman
- Roman to Integer
- Integer to English Words
- Count Primes
- Reverse Bits
- Counting Bits

Design

- Peeking Iterator
- Binary Search Tree Iterator
- Min Stack
- LRU Cache (Classic!!!)

LeetCode and LintCode Problems Summary

Time Complexity Analysis

1. $f(n) = n + 2f(\frac{n}{2}) \rightarrow f(n) = n\log(n)$
2. $f(n) = n + f(\frac{n}{2}) \rightarrow f(n) = 2n$
3. $f(n) = 1 + f(\frac{n}{2}) \rightarrow f(n) = \log(n)$
4. Two pointers: `i` and `j` moving forward, but in `for-for` loop. It's `O(2n)`.

DFS

Generate Parentheses

Idea: Number of left parentheses and right parentheses as parameters separately into dfs function.

```cpp
void dfs(int nleft, int nright, string& crtrst, vector<string>& finalrst)
{
    if (nleft > nright || nleft < 0 || nright < 0)
        return;
    if (nleft == 0 && nright == 0)
    {
        finalrst.push_back(crtrst);
        return;
    }
    crtrst += '(';
    dfs(nleft - 1, nright, crtrst, finalrst);
    crtrst.pop_back();
    crtrst += ')';
    dfs(nleft, nright - 1, crtrst, finalrst);
    crtrst.pop_back();
}
vector<string> generateParenthesis(int n)
{
    vector<string> finalrst;
    string crtrst = "";
    if (n == 0)
        return finalrst;
    dfs(n, n, crtrst, finalrst);
    return finalrst;
}
```

Validate Binary Search Tree

Idea 1: DFS each node, to compute the `minval` and `maxval` of the subtree rooted at this node. The condition is `v0 > lmax && v0 < rmin`

Idea 2: Divide & Conquer: Use **ResultType** to return multiple values. In this problem, **ResultType** includes `isBST`, `minVal` and `maxVal`. The condition is that: both subtrees are BST, and `left's maxVal < root->val < right's minVal`

```
class Node
{
public:
    Node() : maxval(INT_MIN), minval(INT_MAX), isvalid(true) {}
    Node(int x1, int x2, bool x3) : maxval(x1), minval(x2), isvalid(x3) {}
    int maxval, minval;
    bool isvalid;
};
inline int max3(int a, int b, int c)
{
    return max(a, max(b, c));
}
inline int min3(int a, int b, int c)
{
    return min(a, min(b, c));
}
Node helper(TreeNode* root)
{
    if (root == NULL)
        return Node();
    Node leftrst = helper(root->left);
    Node rightrst = helper(root->right);
    Node rst;
    rst.maxval = max3(leftrst.maxval, rightrst.maxval, root->val);
    rst.minval = min3(leftrst.minval, rightrst.minval, root->val);
    if (root->left && root->right)
    {
        rst.isvalid = leftrst.isvalid && rightrst.isvalid
            && root->val > leftrst.maxval && root->val < rightrst.minval;
        return rst;
    }
    else if (root->left == NULL && root->right == NULL)
    {
        rst.isvalid = true;
        return rst;
    }
    else if (root->left && root->right == NULL)
    {
        rst.isvalid = leftrst.isvalid && root->val > leftrst.maxval;
        return rst;
    }
    else
    {
        rst.isvalid = rightrst.isvalid && root->val < rightrst.minval;
        return rst;
    }
}
bool isValidBST(TreeNode *root)
{
    if (root == NULL)
        return true;
    Node rst = helper(root);
    return rst.isvalid;
}
```

Symmetric Tree

Idea: DFS two nodes sharing the same father node. The condition is

1. two nodes NULL or
2. two nodes have the same values and `left node's left == right node's right && left node's right == right node's left`

Sum Root to Leaf Numbers

Idea: Use `crtnum`, `finalsum` to do the DFS. If the current node is a leaf node, add the `crtnum` to `finalsum`. Leaf node is the stop condition (**Leaf Node Stop Condition**), so must add `if (root->left)` or `if (root->right)` to the DFS function.

```
void dfs(TreeNode* root, int crtrst, int& finalrst)
{
    int val = root->val;
    crtrst = crtrst * 10 + val;
    if (root->left == NULL && root->right == NULL)
    {
        finalrst += crtrst;
        return;
    }
    if (root->left)
        dfs(root->left, crtrst, finalrst);
    if (root->right)
        dfs(root->right, crtrst, finalrst);
}
int sumNumbers(TreeNode* root)
{
    int finalrst = 0;
    int crtrst = 0;
    if (root == NULL)
        return 0;
    dfs(root, crtrst, finalrst);
    return finalrst;
}
```

Path Sum (Classic) / Path Sum II

Idea: Similar to Sum Root to Leaf Numbers, use **Leaf Node Stop Condition** to do DFS. For Path Sum II, we should carefully `push_back` and `pop_back` the current node before return. **DFS template will always be like this:** `finalrst, crtrst, crtstatus, push_back(), pop_back()`.

```cpp
// Path Sum I
bool hasPathSum(TreeNode* root, int sum)
{
    if (root == NULL)
        return false;
    sum -= root->val;
    if (root->left == NULL && root->right == NULL)
        return sum == 0;
    if (root->left && hasPathSum(root->left, sum) ||
        root->right && hasPathSum(root->right, sum))
        return true;
    return false;
}
// Path Sum II
void dfs(TreeNode* root, vector<int>& crtrst,
    vector<vector<int>>& finalrst, int sum)
{
    sum -= root->val;
    crtrst.push_back(root->val);
    if (root->left == NULL && root->right == NULL)
    {
        if (sum == 0)
            finalrst.push_back(crtrst);
        crtrst.pop_back();
        return;
    }
    if (root->left)
        dfs(root->left, crtrst, finalrst, sum);
    if (root->right)
        dfs(root->right, crtrst, finalrst, sum);
    crtrst.pop_back();
}
vector<vector<int>> pathSum(TreeNode* root, int sum)
{
    vector<vector<int>> finalrst;
    vector<int> crtrst;
    if (root == NULL)
        return finalrst;
    dfs(root, crtrst, finalrst, sum);
    return finalrst;
}
```

Minimum Depth of Binary Tree

Idea: **Leaf Node Stop Condition**: if leaf node, then return `1`; else return `min(x1,x2)+1`; x1,x2 are minDepth of left and right subtree.

Maximum Depth of Binary Tree

Idea: very similar to Minimum Depth of Binary Tree.

Same Tree

Idea: Very similar to Symmetric Tree. The condition is

1. two nodes NULL or
2. two nodes have the same values and `left's left == right's left && left's right == right's right`

Recover Binary Search Tree (Hard)

Idea: **Template: Inorder Traversal**. Inorder traverse the tree, find the two error nodes by: `if (pre && pre->val > root->val)`: (1) First node: `pre`, (2) Second node: `root`.
A special case: if the two nodes are of father-son relation, then we will not find the second node which satisfies the condition. So, once the first node found, we store `pre` and `root`. If the second node found, we replace the previously-stored `root` with the current `root`.

```
// A Template for inorder tree traversal
void dfs(TreeNode* root, TreeNode*& pre)
{
    if (root == NULL)
        return;
    dfs(root->left, pre);
    // Process Current node:
    // do something on root;
    pre = root;
    dfs(root->right, pre);
}
```

Number of Islands

Idea: traverse each pixel, if it's `1`, then `sum++` and set its neighbors `2`.

Flatten Binary Tree to Linked List (Hard)

Idea: if current node is leaf, return itself. Otherwise, make its right points to its left, DFS on left, and return the last pointer. This pointer's right points to the original right.

```
typedef pair<TreeNode*, TreeNode*> Node;
Node helper(TreeNode* root)
{
    if (root->left == NULL && root->right == NULL)
        return Node(root, root);
    else if (root->left && root->right == NULL)
    {
        Node x = helper(root->left);
        root->right = x.first;
        root->left = NULL;
        return Node(root, x.second);
    }
    else if (root->left == NULL && root->right)
    {
        Node x = helper(root->right);
        root->right = x.first;
        return Node(root, x.second);
    }
    else
    {
        Node x = helper(root->left);
        Node y = helper(root->right);
        x.second->right = y.first;
        root->right = x.first;
        root->left = NULL;
        return Node(root, y.second);
    }
}
void flatten(TreeNode* root)
{
    if (root == NULL)
        return;
    Node x = helper(root);
}
```

Convert Sorted List to Binary Search Tree (Find the middle node of a linkedlist within a range)

Idea: Find the mid node `mid` of a range `[head,tail)` of a linked list, set `mid` as the root of the BST. Its left child is the DFS result of `[head,mid)`, and its right child is the DFS result of `[mid->next,tail)`.

```
// find the middle node of a linked list within range [p,q)
ListNode* find_mid_node(ListNode* p, ListNode* q)
{
    if (p == q)
        return NULL;
    ListNode *mid, *temp;
    mid = temp = p;
    while (temp != q && temp->next != q)
    {
        mid = mid->next;
        temp = temp->next->next;
    }
    return mid;
}
```

A simpler solution:

1. Find middle of a linked list, noted as `mid`
2. Set `mid`'s `pre`'s next `= NULL`, then, convert its left part as the left tree, and right part as the right tree, and `mid` as root;

```
TreeNode* sortedListToBST(ListNode* head)
{
    if (head == NULL)
        return NULL;
    ListNode dummy(0);
    dummy.next = head;
    ListNode *p = head, *q = head, *pre = &dummy;
    while (q && q->next && q->next->next)
    {
        pre = p;
        p = p->next;
        q = q->next->next;
    }
    pre->next = NULL;
    TreeNode* root = new TreeNode(p->val);
    root->left = sortedListToBST(dummy.next);
    root->right = sortedListToBST(p->next);
    return root;
}
```

Convert Sorted Array to Binary Search Tree

Idea: very similar to Convert Sorted List to Binary Search Tree. In array, we use `[s,e]` to represent a range, because computing `mid` can be `mid = s + (e-s)/2;`. And stop condition is `if (e < s)`.

Construct Binary Tree from Preorder and Inorder Traversal

Idea: The first element of preorder array is the root, find this element in inorder array, then we know the elements on the left of this element in the inorder array is the left subtree, and the elements on the right of this element in the inorder array is the right subtree. Note: `[s,e]` results in stop condition is `if (s > e)`

```
TreeNode* helper(vector<int>& P, int s1, int e1, vector<int>& I, int s2, int e2)
{
    if (s2 > e2 || s1 > e1)
        return NULL;
    int rootval = P[s1];
    int index = s2;
    for (; index <= e2 && I[index] != rootval; index++);
    int nleft = index - 1 - s2 + 1, nright = e2 - (index + 1) + 1;
    TreeNode *root = new TreeNode(rootval);
    root->left = helper(P, s1 + 1, s1 + nleft, I, s2, index - 1);
    root->right = helper(P, s1 + nleft + 1, s1 + nleft + nright, I, index + 1, e2);
    return root;
}
TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder)
{
    int n1 = preorder.size(), n2 = inorder.size();
    if (n1 != n2 || n1 == 0)
        return NULL;
    return helper(preorder, 0, n1 - 1, inorder, 0, n2 - 1);
}
```

Construct Binary Tree from Inorder and Postorder Traversal

Idea: Very similar to Construct Binary Tree from Preorder and Inorder Traversal.

```
TreeNode* helper(vector<int>& inorder, int s1, int e1,
    vector<int>& postorder, int s2, int e2)
{
    if (s1 > e1 || s2 > e2)
        return NULL;
    int rootval = postorder[e2];
    TreeNode* root = new TreeNode(rootval);
    int index = find(inorder.begin() + s1, inorder.begin() + e1 + 1, rootval)
                - inorder.begin();
    root->left = helper(inorder, s1, index - 1, postorder, s2, index - 1 - s1 + s2);
    root->right = helper(inorder, index + 1, e1, postorder, index - s1 + s2, e2 - 1);
    return root;
}
TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder)
{
    int n1 = inorder.size(), n2 = postorder.size();
    if (n1 != n2 || n1 == 0)
        return NULL;
    return helper(inorder, 0, n1 - 1, postorder, 0, n2 - 1);
}
```

Clone Graph (Graph, Check Again!)

Idea: use `unordered_map<int,*> to store the cloned node`.

```
UndirectedGraphNode *cloneGraph(UndirectedGraphNode *node)
{
    if (node == NULL)
        return NULL;
    typedef UndirectedGraphNode UGN;
    unordered_map<int, UGN*> hashMap;
    unordered_set<UGN*> S;
    // Step 1. Clone nodes.
    queue<UGN*> Q;
    Q.push(node);
    UGN* cp = new UGN(node->label);
    hashMap[node->label] = cp;
    int n1 = 1, n2 = 0;
    while (!Q.empty())
    {
        UGN* x = Q.front();
        Q.pop();
        n1--;
        for (int i = 0; i < x->neighbors.size(); i++)
        {
            UGN* nb = x->neighbors[i];
            if (hashMap.count(nb->label))
                continue;
            hashMap[nb->label] = new UGN(nb->label);
            Q.push(nb);
            n2++;
        }
        if (n1 == 0)
            swap(n1, n2);
    }
    // Step 2. Clone neighbors.
    UGN* rst = hashMap[node->label];
    Q.push(node);
    S.insert(node);
    n1 = 1, n2 = 0;
    while (!Q.empty())
    {
        UGN* x = Q.front();
        UGN* cpx = hashMap[x->label];
        Q.pop();
        n1--;
        for (int i = 0; i < x->neighbors.size(); i++)
        {
            UGN* nb = x->neighbors[i];
            UGN* cpnb = hashMap[nb->label];
            cpx->neighbors.push_back(cpnb);
            if (S.count(nb))
                continue;
            S.insert(nb);
            Q.push(nb);
            n2++;
        }
        if (n1 == 0)
            swap(n1, n2);
    }
    return rst;
}
```

A more concise DFS version:

```cpp
unordered_map<UndirectedGraphNode*, UndirectedGraphNode*> hash;
UndirectedGraphNode *cloneGraph(UndirectedGraphNode *node)
{
    if (node == NULL)
        return node;
    if (hash.find(node) == hash.end())
    {
        hash[node] = new UndirectedGraphNode(node->label);
        for (auto x : node->neighbors)
        {
            (hash[node]->neighbors).push_back(cloneGraph(x));
        }
    }
    return hash[node];
}
```

Binary Tree Paths (Classic)

> **Summary**
>
> 1. Leaf Node Stop condition
> 2. If Processing is in current root, then restore it to its original value. (Note: Each branch needs restoration!)

House Robber III

Idea: Divide and Conquer. For each node, there are two results, selected `rst[1]` or not selected `rst[0]`.

1. `rst[0] = max(leftrst[0], leftrst[1]) + max(rightrst[0], rightrst[1])`
2. `rst[1] = root->val + leftrst[0] + rightrst[0]`.

```cpp
vector<int> helper(TreeNode* root)
{
    if (root == NULL)
        return vector<int>(2, 0);
    vector<int> rst(2);
    vector<int> leftrst = helper(root->left);
    vector<int> rightrst = helper(root->right);
    rst[0] = max(leftrst[0], leftrst[1]) + max(rightrst[0], rightrst[1]);
    rst[1] = root->val + leftrst[0] + rightrst[0];
    return rst;
}
int rob(TreeNode* root)
{
    vector<int> rst = helper(root);
    return max(rst[0], rst[1]);
}
```

Balanced Binary Tree

Idea: Divide and Conquer. Define two variables `isbalanced` and `depth` stored into a `struct`. And check whether `abs(lefttree's depth - righttree's depth) <= 1 && lefttree isbalanced && righttree isbalanced`.

```cpp
class Node
{
public:
    Node() : depth(0), isbalanced(true) {}
    Node(int x1, bool x2) : depth(x1), isbalanced(x2) {}
    int depth;
    bool isbalanced;
};
Node helper(TreeNode* root)
{
    if (root == NULL)
        return Node();
    Node leftrst = helper(root->left);
    Node rightrst = helper(root->right);
    Node rst;
    rst.depth = max(leftrst.depth, rightrst.depth) + 1;
    rst.isbalanced = leftrst.isbalanced && rightrst.isbalanced
        && abs(leftrst.depth - rightrst.depth) <= 1;
    return rst;
}
bool isBalanced(TreeNode* root)
{
    if (root == NULL)
        return true;
    Node rst = helper(root);
    return rst.isbalanced;
}
```

Binary Tree Maximum Path Sum (Hard) (Check Again!)

Idea: Define two variables, which are stored into a `struct` or `class`.

1. The max path sum of a tree `maxPathSum`,
2. The max path sum from root of a tree `maxRootPathSum`.
   So we have a recursive equation about a root node and its children. Note the cases if subtree's result is less than 0.

```
class Node
{
public:
    Node() : maxPathSum(INT_MIN), maxRootPathSum(INT_MIN) {}
    Node(int x1, int x2) : maxPathSum(x1), maxRootPathSum(x2) {}
    int maxPathSum;
    int maxRootPathSum;
};
inline int max3(int a, int b, int c)
{
    return max(a, max(b, c));
}
Node Helper(TreeNode* root)
{
    if (root == NULL)
        return Node();
    Node leftrst = Helper(root->left);
    Node rightrst = Helper(root->right);
    Node rst;
    // maxPathSum locates in
    // 1. left tree, or
    // 2. right tree or
    // 3. a path passing both trees, Note that in this case,
    //    if subtree's result is < 0, drop it.
    rst.maxPathSum = max3
    (
        leftrst.maxPathSum,
        rightrst.maxPathSum,
        root->val + max(0, leftrst.maxRootPathSum)
            + max(0, rightrst.maxRootPathSum)
    );
    // maxRootPathSum locates in
    // 1. left tree, or
    // 2. right tree
    // Note that the root itself must be included,
    //    and if subtree's result is < 0, drop it.
    rst.maxRootPathSum = root->val
        + max3(0, leftrst.maxRootPathSum, rightrst.maxRootPathSum);
    return rst;
}
int maxPathSum(TreeNode* root)
{
    if (root == NULL)
        return 0;
    Node rst = Helper(root);
    return rst.maxPathSum;
}
```

BFS

Populating Next Right Pointers in Each Node I / II (Hard)

Idea: Level-order traversal, standard template. But O(n) space. So should solve it using level-order traversal + linked list.

Binary Tree Right Side View

Idea:

1. Standard Level-order traversal. This solution is trivial.
2. DFS. Define the `maxDepth` the tree is traversed, and `crtDepth` at each node, if `crtDepth > maxDepth` then it means at `crtDepth`, this node is first visited. Of course, keep `root->right` before `root->left` traversed.

```
void dfs(TreeNode* root, int crtDepth, int& maxDepth, vector<int>& rst)
{
    if (crtDepth > maxDepth)
    {
        rst.push_back(root->val);
        maxDepth = crtDepth;
    }
    if (root->left == NULL && root->right == NULL)
        return;
    if (root->right)
        dfs(root->right, crtDepth + 1, maxDepth, rst);
    if (root->left)
        dfs(root->left, crtDepth + 1, maxDepth, rst);
}
vector<int> rightSideView(TreeNode* root)
{
    vector<int> rst;
    if (root == NULL)
        return rst;
    int maxDepth = INT_MIN, crtDepth = 0;
    dfs(root, crtDepth, maxDepth, rst);
    return rst;
}
```

Surrounded Regions

Idea: BFS seems faster, while another solution is Union-Find.

Backtracking / Search - Advanced (DFS / BFS)

Palindrome Partitioning (Classic!!!)

Idea: DFS string `s`: divide `s` into two substrings `s1`, `s2`. If `s1` is palindrome, then DFS `s2`. otherwise continue.

```cpp
void dfs(string& s, int start, vector<string>& crtrst,
    vector<vector<string>>& finalrst, const vector<vector<int>>& P)
{
    int n = s.size();
    if (start == n)
    {
        finalrst.push_back(crtrst);
        return;
    }
    for (int i = start; i < n; i++)
    {
        bool isPalindrome = P[start][i];
        if (!isPalindrome)
            continue;
        crtrst.push_back(s.substr(start, i - start + 1));
        dfs(s, i + 1, crtrst, finalrst, P);
        crtrst.pop_back();
    }
}
void gen_is_palindrome(string& s, vector<vector<int>>& P)
{
    int n = P.size();
    for (int i = 0; i < n; i++)
        P[i][i] = true;
    for (int i = 0; i < n - 1; i++)
        P[i][i + 1] = s[i] == s[i + 1];
    int i = 0, j = 2, k = 3;
    while (j < n)
    {
        P[i][j] = P[i + 1][j - 1] && s[i] == s[j];
        i++;
        j++;
        if (j == n)
        {
            i = 0;
            j = k++;
        }
    }
}
vector<vector<string>> partition(string s)
{
    vector<vector<string>> finalrst;
    vector<string> crtrst;
    int n = s.size();
    if (n == 0)
        return finalrst;
    vector<vector<int>> P(n, vector<int>(n, 0));
    gen_is_palindrome(s, P);
    dfs(s, 0, crtrst, finalrst, P);
    return finalrst;
}
```

Combinations

Idea: For C_n^k, we use `k` to denote how many numbers have been added. Once `k == 0`, we push the `crtrst` into `finalrst`;

```cpp
void dfs(int s, int e, int k, vector<int>& crtrst,
    vector<vector<int>>& finalrst)
{
    if (k == 0)
    {
        finalrst.push_back(crtrst);
        return;
    }
    for (int i = s; i <= e; i++)
    {
        crtrst.push_back(i);
        dfs(i + 1, e, k - 1, crtrst, finalrst);
        crtrst.pop_back();
    }
}
vector<vector<int>> combine(int n, int k)
{
    vector<vector<int>> finalrst;
    vector<int> crtrst;
    if (n == 0 || k == 0 || k > n)
        return finalrst;
    dfs(1, n, k, crtrst, finalrst);
    return finalrst;
}
```

Combination Sum / Combination Sum II / Combination Sum III (Classic!!!)

Idea: DFS Template

> DFS: define a `crtrst`, `finalrst`. The stop condition is when the condition satisfied, `crtrst` will be added to `finalrst`.

```cpp
// This is a general template:
void dfs(const vector<int>& cds, int s, int tar,
    vector<int>& crtrst, vector<vector<int>>& finalrst)
{
    // Stop Contidion:
    if (tar == 0)
    {
        finalrst.push_back(crtrst);
        return;
    }
    // If Not to Stop Condition:
    for (int i = s; i < cds.size(); i++)
    {
        // tar - cds[i] < 0 means we do not need to calculate the rest elements since they must be larger than the current one.
        if (tar - cds[i] < 0)
            break;
        // repeated numbers need to be counted once.
        if (i > s && cds[i] == cds[i - 1])
            continue;
        crtrst.push_back(cds[i]);
        dfs(cds, s, tar - cds[i], crtrst, finalrst);
        crtrst.pop_back();
    }
}

vector<vector<int>> CombinationSum(vector<int>& cds, int tar)
{
    // Given an array of candidates,
    // Search the combinations such that their sum is tar.
    // 1. Define crtrst, finalrst:
    vector<vector<int>> finalrst;
    vector<int> crtrst;
    sort(cds.begin(), cds.end()); // make the results ascending order
    // 2. DFS: dfs the cds with starting index s, searching it
    // to every possible corner, and add the satisfied rst to finalrst;
    int s = 0;
    dfs(cds, s, tar, crtrst, finalrst);
}
```

```
void dfs(int n, int k, int s, vector<int>& crtrst,
    vector<vector<int>>& finalrst, vector<int>& visited)
{
    if (n == 0 && k == 0)
    {
        finalrst.push_back(crtrst);
        return;
    }
    if (n == 0 && k != 0 || n != 0 && k == 0)
        return;
    for (int i = s; i <= 9; i++)
    {
        if (visited[i])
            continue;
        if (n - i < 0)
            break;
        crtrst.push_back(i);
        visited[i] = 1;
        dfs(n - i, k - 1, i + 1, crtrst, finalrst, visited);
        visited[i] = 0;
        crtrst.pop_back();
    }
}
vector<vector<int>> combinationSum3(int k, int n)
{
    vector<vector<int>> finalrst;
    vector<int> crtrst;
    vector<int> visited(10, 0);
    if (k == 0 || n < k)
        return finalrst;
    dfs(n, k, 1, crtrst, finalrst, visited);
    return finalrst;
}
```

Note: Follow N-Queens' idea, if we don't use loop in dfs, then we should write code like this:

```cpp
void dfs(const vector<int>& cds, int crtindex, int tar,
    vector<int>& crtrst, vector<vector<int>>& finalrst)
{
    if (tar == 0)
    {
        finalrst.push_back(crtrst);
        return;
    }
    if (crtindex >= cds.size())
        return;
    if (tar - cds[crtindex] < 0)
        return;
    // There will be 2 cases for the search: select the current element, or not select
    // Case 1: Select
    crtrst.push_back(cds[crtindex]);
    dfs(cds, crtindex + 1, tar - cds[crtindex], crtrst, finalrst);
    crtrst.pop_back();
    // Case 2: Not Select, must note that, if we choose not to select the current elem
ent,
    // then we should not select the following same elements, since it will cause dupl
icate answers.
    int i = crtindex + 1;
    while (i < cds.size() && cds[i] == cds[crtindex]) i++; // find the next first one
!= crt element
    dfs(cds, i, tar, crtrst, finalrst);
}
```

Letter Combinations of a Phone Number

Idea: At each string's character, find its mapped alphabet characters.

```cpp
void dfs(string& digits, int k, vector<string>& keyboard,
    string& crtrst, vector<string>& finalrst)
{
    if (k == digits.size())
    {
        finalrst.push_back(crtrst);
        return;
    }
    int index = digits[k] - '0';
    for (int i = 0; i < keyboard[index].size(); i++)
    {
        char c = keyboard[index][i];
        crtrst += c;
        dfs(digits, k + 1, keyboard, crtrst, finalrst);
        crtrst.pop_back();
    }
}
vector<string> letterCombinations(string digits)
{
    vector<string> keyboard = { " ", "", "abc", "def", "ghi",
        "jkl", "mno", "pqrs", "tuv", "wxyz" };
    vector<string> finalrst;
    string crtrst;
    int n = digits.size();
    if (n == 0)
        return finalrst;
    dfs(digits, 0, keyboard, crtrst, finalrst);
    return finalrst;
}
```

N-Queens / N-Queens II (Classic!!!)

Idea: Define `dfs(crtrst, finalrst, row)` as the meaning that, given the existing `crtrst`, search all results starting from `row`.

这个问题和Combination Sum略有不同，Combination Sum在做DFS时，是从start开始把之后的所有数都遍历了一次，其隐含的意义包括当前这个数不选，结果如何。而本问题中，每一个行都必须有一个状态，不可跳过，所以就必须只处理此行，没有做循环。

```cpp
bool isValid(vector<string>& crtrst, int u, int v)
{
    int n = crtrst.size();
    // up:
    for (int i = 0; i < u; i++)
        if (crtrst[i][v] == 'Q')
            return false;
    // upper-left:
    for (int i = u - 1, j = v - 1; i >= 0 && j >= 0; i--, j--)
        if (crtrst[i][j] == 'Q')
            return false;
    // upper-right:
    for (int i = u - 1, j = v + 1; i >= 0 && j < n; i--, j++)
        if (crtrst[i][j] == 'Q')
            return false;
    return true;
}
void dfs(vector<string>& crtrst, vector<vector<string>>& finalrst, int sr)
{
    int n = crtrst.size();
    if (sr == n)
    {
        finalrst.push_back(crtrst);
        return;
    }
    for (int j = 0; j < n; j++)
    {
        bool valid = isValid(crtrst, sr, j);
        if (!valid)
            continue;
        crtrst[sr][j] = 'Q';
        dfs(crtrst, finalrst, sr + 1);
        crtrst[sr][j] = '.';
    }
}
vector<vector<string>> solveNQueens(int n)
{
    vector<vector<string>> finalrst;
    if (n == 0)
        return finalrst;
    vector<string> crtrst(n, string(n, '.'));
    int startRow = 0;
    dfs(crtrst, finalrst, startRow);
    return finalrst;
}
```

The code above can be further optimized, using hash table to record whether two diagonals and vertical line can be filled with queen.

```cpp
void dfs(int r, vector<string>& crtrst, vector<vector<string>>& finalrst,
    vector<int>& hashV, vector<int>& hashD1, vector<int>& hashD2)
{
    int n = crtrst.size();
    if (r == n)
    {
        finalrst.push_back(crtrst);
        return;
    }
    for (int j = 0; j < n; j++)
    {
        if (hashV[j] || hashD1[j - r + n - 1] || hashD2[j + r])
            continue;
        crtrst[r][j] = 'Q';
        hashV[j] = hashD1[j - r + n - 1] = hashD2[j + r] = 1;
        dfs(r + 1, crtrst, finalrst, hashV, hashD1, hashD2);
        hashV[j] = hashD1[j - r + n - 1] = hashD2[j + r] = 0;
        crtrst[r][j] = '.';
    }
}
vector<vector<string>> solveNQueens(int n)
{
    vector<vector<string>> finalrst;
    if (n == 0)
        return finalrst;
    vector<string> crtrst(n, string(n, '.'));
    vector<int> hashV(n, 0), hashD1(2 * n - 1, 0), hashD2(2 * n - 1, 0);
    dfs(0, crtrst, finalrst, hashV, hashD1, hashD2);
    return finalrst;
}
```

Subsets / Subsets II

Idea: Also search every status. Note its stop condition

1. If using loop idea: 求一个array的subsets，即是将数组中每个数分别放入 `crtrst`，然后再看之后的元素。这种想法很容易犯一个错误，即认为停止条件是 `startIndex == n`. 如果这样的话，就会漏掉路径中非叶子节点状态。举个例子：如求 `[1,2,3]` 的subsets，将 `1` 放入后，会进一步分别放 `2` 或 `3`。但是只有放 `3` 之后才会触发停止条件，导致 `[1,2]` 这个解被漏掉。实际上，只要进入dfs函数体时就应该把 `crtrst` 放入 `finalrst`.

```cpp
void dfs(vector<int>& nums, vector<int>& crtrst, vector<vector<int>>& finalrst, int idx)
{
    finalrst.push_back(crtrst);
    for (int i = idx; i < nums.size(); i++)
    {
        crtrst.push_back(nums[i]);
        dfs(nums, crtrst, finalrst, i + 1);
        crtrst.pop_back();
    }
}
vector<vector<int>> subsets(vector<int> &nums)
{
    vector<vector<int>> finalrst;
    vector<int> crtrst;
    sort(nums.begin(), nums.end());
    dfs(nums, crtrst, finalrst, 0);
    return finalrst;
}
```

2. If not using loop idea: 求一个array的subsets，即是把当前索引的数要么放入 `crtrst` 要么忽略（选或不选），然后dfs后面的元素。此时的停止条件则是 `startIndex == n`.

```cpp
void dfs(const vector<int>& nums, int s, vector<int>& crtrst, vector<vector<int>>& finalrst)
{
    int n = nums.size();
    if (s == n)
    {
        finalrst.push_back(crtrst);
        return;
    }
    crtrst.push_back(nums[s]);
    dfs(nums, s + 1, crtrst, finalrst);
    crtrst.pop_back();
    dfs(nums, s + 1, crtrst, finalrst);
}
```

For Subsets II, just jump the consecutive same elements.

```cpp
// For-Loop idea:
void dfs1(const vector<int>& S, int s, vector<int>& crtrst,
    vector<vector<int>>& finalrst)
{
    finalrst.push_back(crtrst);
    for (int i = s; i < S.size(); i++)
    {
        if (i > s && S[i] == S[i - 1])
            continue;
        crtrst.push_back(S[i]);
        dfs1(S, i + 1, crtrst, finalrst);
        crtrst.pop_back();
    }
}
// Non-For-Loop idea:
void dfs2(const vector<int>& S, int s, vector<int>& crtrst,
    vector<vector<int>>& finalrst)
{
    int n = S.size();
    if (s == n)
    {
        finalrst.push_back(crtrst);
        return;
    }
    crtrst.push_back(S[s]);
    dfs2(S, s + 1, crtrst, finalrst);
    crtrst.pop_back();
    int i = s + 1;
    while (i < n && S[i] == S[s]) i++;
    dfs2(S, i, crtrst, finalrst);
}
```

Permutations / Permutations II

Idea: Use `visited` array to indicate whether a number has been selected. For duplicated number, we must use `last` and `lastExisted` two variables to check whether current number can be selected.

```cpp
// Permutations I
void dfs(vector<int>& A, vector<int>& crtrst,
    vector<vector<int>>& finalrst, vector<int>& visited)
{
    if (crtrst.size() == A.size())
    {
        finalrst.push_back(crtrst);
        return;
    }
    for (int i = 0; i < A.size(); i++)
    {
        if (visited[i])
            continue;
        crtrst.push_back(A[i]);
        visited[i] = 1;
        dfs(A, crtrst, finalrst, visited);
        visited[i] = 0;
        crtrst.pop_back();
    }
}
vector<vector<int>> permute(vector<int>& nums)
{
    vector<vector<int>> finalrst;
    vector<int> crtrst;
    int n = nums.size();
    if (n == 0)
        return finalrst;
    vector<int> visited(n, 0);
    dfs(nums, crtrst, finalrst, visited);
    return finalrst;
}
// Permutations II
void dfs(vector<int>& A, vector<int>& crtrst,
    vector<int>& visited, vector<vector<int>>& finalrst)
{
    if (crtrst.size() == A.size())
    {
        finalrst.push_back(crtrst);
        return;
    }
    int last = -1;
    bool lastExisted = false;
    for (int i = 0; i < A.size(); i++)
    {
        if (visited[i])
            continue;
        if (lastExisted && A[i] == last)
            continue;
        last = A[i];
        lastExisted = true;
        crtrst.push_back(A[i]);
        visited[i] = 1;
        dfs(A, crtrst, visited, finalrst);
        visited[i] = 0;
        crtrst.pop_back();
    }
}
vector<vector<int>> permuteUnique(vector<int>& nums)
{
    vector<int> crtrst;
```

```
61.     vector<vector<int>> finalrst;
62.     int n = nums.size();
63.     if (n == 0)
64.         return finalrst;
65.     sort(nums.begin(), nums.end());
66.     vector<int> visited(n, 0);
67.     dfs(nums, crtrst, visited, finalrst);
68.     return finalrst;
69. }
```

Word Ladder I / Word Ladder II

Idea:

1. 对于Problem 1, 使用BFS搜索整个graph，到达 `endWord` 时即可得到距离；但此法较慢，一种快速的方法是：从 `beginWord` 和 `endWord` 分别BFS，保持两者遍历过的节点数量几乎一致。当两个访问过节点集合有重复时，返回距离。但此法对于Problem 2很难实现（也可以实现，不过会很复杂）。
2. 对于Problem 2, 首先用普通的BFS从 `endWord` 往 `beginWord` 搜索，记录下当前节点到 `endWord` 的距离。然后再从 `beginWord` 做DFS: DFS当前节点时，就把当前节点放入 `crtrst`，然后对于其所有的邻居，如果发现他们到 `endWord` 的距离比目前距离小1，则进一步DFS这个节点；待发现当前节点是 `endWord` 时，把 `crtrst` 放入 `finalrst` 中。

```cpp
void bfs(string beginWord, string endWord,
    unordered_set<string>& wordList, unordered_map<string, int>& HashMap)
{ // Compute the dist from current node to endWord, stored in HashMap.
    HashMap[endWord] = 0;
    if (beginWord == endWord)
    {
        HashMap[beginWord] = 0;
        return;
    }
    if (beginWord.size() != endWord.size())
    {
        HashMap[beginWord] = INT_MAX;
        return;
    }
    int len = 1;
    queue<string> Q;
    Q.push(endWord);
    wordList.erase(endWord);
    int n0 = 1, n1 = 0;
    while (!Q.empty())
    {
        string tmp = Q.front();
        Q.pop();
        n0--;
        for (int i = 0; i < tmp.size(); i++)
        {
            char origChar = tmp[i];
            for (char c = 'a'; c <= 'z'; c++)
            {
                if (c == origChar)
                    continue;
                tmp[i] = c;
                if (tmp == beginWord)
                {
                    HashMap[beginWord] = len;
                    return;
                }
                if (wordList.count(tmp))
                {
                    wordList.erase(tmp);
                    HashMap[tmp] = len;
                    Q.push(tmp);
                    n1++;
                }
            }
            tmp[i] = origChar;
        }
        if (n0 == 0)
        {
            swap(n0, n1);
            len++;
        }
    }
    if (HashMap.find(beginWord) == HashMap.end())
        HashMap[beginWord] = INT_MAX;
}

void dfs(string beginWord, const string& endWord,
    vector<string>& crtrst, vector<vector<string>>& finalrst,
    unordered_set<string>& wdList, unordered_map<string, int>& HashMap)
```

```
{
    crtrst.push_back(beginWord);
    int d = HashMap[beginWord];
    if (beginWord == endWord)
    {
        finalrst.push_back(crtrst);
        crtrst.pop_back();
        return;
    }
    for (int i = 0; i < beginWord.size(); i++)
    {
        char origChar = beginWord[i];
        for (char c = 'a'; c <= 'z'; c++)
        {
            if (c == origChar)
                continue;
            beginWord[i] = c;
            if (wdList.count(beginWord) && HashMap[beginWord] == d - 1)
            {
                dfs(beginWord, endWord, crtrst, finalrst, wdList, HashMap);
            }
        }
        beginWord[i] = origChar;
    }
    crtrst.pop_back();
}

vector<vector<string>> findLadders(string beginWord, string endWord,
    unordered_set<string> &wordList)
{
    unordered_map<string, int> HashMap;
    unordered_set<string> copyWordList = wordList;
    bfs(beginWord, endWord, wordList, HashMap);
    vector<vector<string>> finalrst;
    vector<string> crtrst;
    int Dist = HashMap[beginWord];
    if (Dist == INT_MAX)
        return finalrst;
    dfs(beginWord, endWord, crtrst, finalrst, copyWordList, HashMap);
    return finalrst;
}
```

Topological Sorting / Course Schedule / Course Schedule II

Idea: **TopoSort**.

- BFS: straight-forward, use a `HashMap` to store the in-degrees of each node. For those nodes with 0 in-degree, push them into a queue. When visiting a node from the queue, reduce its neighbor's in-degree by 1, if in-degree reaches 0, push it into the queue, until the queue empty. Note the case there is a loop: **If the number of nodes poped from the queue is equal to the graph's nodes, then it means no loop; otherwise, there is loop in the graph.**

```cpp
vector<DirectedGraphNode*> topSort(vector<DirectedGraphNode*> graph)
{
    unordered_map<DirectedGraphNode*, int> HashMap;
    for (int i = 0; i < graph.size(); i++)
        HashMap[graph[i]] = 0;
    for (int i = 0; i < graph.size(); i++)
    {
        for (int j = 0; j < graph[i]->neighbors.size(); j++)
        {
            HashMap[graph[i]->neighbors[j]]++;
        }
    }
    queue<DirectedGraphNode*> Q;
    int n0 = 0, n1 = 0;
    for (auto it = HashMap.begin(); it != HashMap.end(); it++)
    {
        if (it->second == 0)
        {
            Q.push(it->first);
            n0++;
        }
    }
    vector<DirectedGraphNode*> rst;
    while (!Q.empty())
    {
        DirectedGraphNode* tmp = Q.front();
        Q.pop();
        rst.push_back(tmp);
        n0--;
        for (int i = 0; i < tmp->neighbors.size(); i++)
        {
            HashMap[tmp->neighbors[i]]--;
            if (HashMap[tmp->neighbors[i]] == 0)
            {
                Q.push(tmp->neighbors[i]);
                n1++;
            }
        }
        if (n0 == 0)
        {
            swap(n0, n1);
        }
    }
    return rst;
}
```

- DFS: define dfs as: dfs all of the `crtnode`'s unvisited neighbors and then put `crtnode` into a stack. Because all of `crtnode`'s neighbors must be put into the stack before `crtnode`, then `crtnode` must be topoSorted before its neighbors. **A Post-Order DFS can get a reversed order of topoSort**. A detailed tutorial can be found here TopoSort in DFS. 通俗解释：DFS某个顶点的做法为：把其所有的neighbors全部DFS掉，然后再把当前点压入`vector`，同时删掉当前点（即已经处理过了）。那么由于一个点的neighbors（其实也就是孩子们）都被搞定了，自己也就没有存在的必要了。此顺序必然是拓扑排序的逆序。

```cpp
void dfs(DirectedGraphNode* crtnode, unordered_set<DirectedGraphNode*>& S,
    vector<DirectedGraphNode*>& rst)
{
    if (S.count(crtnode) == 0)
        return;
    for (int i = 0; i < crtnode->neighbors.size(); i++)
    {
        dfs(crtnode->neighbors[i], S, rst);
    }
    S.erase(crtnode);
    rst.push_back(crtnode);
}
vector<DirectedGraphNode*> topSort(vector<DirectedGraphNode*> graph)
{
    unordered_set<DirectedGraphNode*> S(graph.begin(), graph.end()); // S unvisited
    vector<DirectedGraphNode*> rst;
    for (int i = 0; i < graph.size(); i++)
    {
        dfs(graph[i], S, rst);
    }
    reverse(rst.begin(), rst.end());
    return rst;
}
```

Minimum Height Trees

Idea: Topo-sort. From the nodes of degree one, remove them level-by-level, until there are one or two nodes left.

```cpp
vector<int> findMinHeightTrees(int n, vector<pair<int, int>>& edges)
{
    vector<int> rst;
    if (n <= 2)
    {
        for (int i = 0; i < n; i++)
            rst.push_back(i);
        return rst;
    }
    vector<unordered_set<int>> E(n);
    for (int i = 0; i < edges.size(); i++)
    {
        int x = edges[i].first, y = edges[i].second;
        E[x].insert(y);
        E[y].insert(x);
    }
    queue<int> Q;
    int n1 = 0, n2 = 0;
    for (int i = 0; i < n; i++)
    {
        if (E[i].size() != 1)
            continue;
        Q.push(i);
        n1++;
    }
    while (!Q.empty())
    {
        int x = Q.front();
        Q.pop();
        n1--;
        n--;
        for (auto it = E[x].begin(); it != E[x].end(); it++)
        {
            int y = *it;
            E[y].erase(x);
            if (E[y].size() == 1)
            {
                Q.push(y);
                n2++;
            }
        }
        E[x].clear();
        if (n1 == 0)
        {
            if (n <= 2)
            {
                while (!Q.empty())
                {
                    rst.push_back(Q.front());
                    Q.pop();
                }
                return rst;
            }
            swap(n1, n2);
        }
    }
    return rst;
}
```

Word Search

Idea: DFS.

```
bool dfs(vector<vector<char>>& bd, int i, int j,
    vector<vector<int>>& visited, string& wd, int k)
{
    int shift[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
    int m = bd.size(), n = bd[0].size();
    if (bd[i][j] != wd[k])
        return false;
    if (k == wd.size() - 1) // 注意这里，不可写作k == wd.size()，因为可能无法触发
        return true;
    visited[i][j] = true;
    for (int d = 0; d < 4; d++)
    {
        int u = i + shift[d][0];
        int v = j + shift[d][1];
        if (u < 0 || u >= m || v < 0 || v >= n || visited[u][v])
            continue;
        bool x = dfs(bd, u, v, visited, wd, k + 1);
        if (x)
        {
            visited[i][j] = false;
            return true;
        }
    }
    visited[i][j] = false;
    return false;
}
bool exist(vector<vector<char>>& board, string word)
{
    int m = board.size();
    int n = m ? board[0].size() : m;
    if (m * n == 0)
        return false;
    if (word.size() == 0)
        return false;
    if (m * n < word.size())
        return false;
    vector<vector<int>> visited(m, vector<int>(n, 0));
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < n; j++)
        {
            bool rst = dfs(board, i, j, visited, word, 0);
            if (rst)
                return true;
        }
    }
    return false;
}
```

Word Search II

Idea: Trie.

1. Build a Trie, to store all words.
2. DFS the board, if find word exists, add it to the result.

Note:

1. There may be duplicate words in the `finalrst`, so once we find a word, we set it non-exist in the Trie.
2. If a word found, we will continue search, not return!!!

```
class TrieNode
{
public:
    TrieNode() : count(0) { memset(children, 0, sizeof(TrieNode*) * 26); }
    int count;
    TrieNode* children[26];
};
void insert(TrieNode* root, string& s)
{
    TrieNode *p = root;
    for (int i = 0; i < s.size(); i++)
    {
        char c = s[i];
        if (p->children[c - 'a'] == NULL)
            p->children[c - 'a'] = new TrieNode();
        p = p->children[c - 'a'];
    }
    p->count++;
}
TrieNode* buildTrie(vector<string>& words)
{
    TrieNode *root = new TrieNode();
    for (int i = 0; i < words.size(); i++)
        insert(root, words[i]);
    return root;
}
void destroy(TrieNode* root)
{
    if (root == NULL)
        return;
    for (int i = 0; i < 26; i++)
        destroy(root->children[i]);
    delete root;
}
void dfs(vector<vector<char>>& bd, int i, int j, int m, int n,
    TrieNode* root, string& crtrst, vector<string>& finalrst, vector<vector<int>>& vis
ited)
{
    if (i >= m || i < 0 || j >= n || j < 0 || visited[i][j])
        return;
    char c = bd[i][j];
    visited[i][j] = 1;
    crtrst += c;
    if (root->children[c - 'a'] == NULL)
    {
        crtrst.pop_back();
        visited[i][j] = 0;
        return;
    }
    if (root->children[c - 'a']->count)
    {
        finalrst.push_back(crtrst);
        root->children[c - 'a']->count = 0;
    }
    dfs(bd, i + 1, j, m, n, root->children[c - 'a'], crtrst, finalrst, visited);
    dfs(bd, i - 1, j, m, n, root->children[c - 'a'], crtrst, finalrst, visited);
    dfs(bd, i, j + 1, m, n, root->children[c - 'a'], crtrst, finalrst, visited);
    dfs(bd, i, j - 1, m, n, root->children[c - 'a'], crtrst, finalrst, visited);
    visited[i][j] = 0;
    crtrst.pop_back();
```

```
}
vector<string> findWords(vector<vector<char>>& board, vector<string>& words)
{
    TrieNode* root = buildTrie(words);
    vector<string> finalrst;
    string crtrst;
    int m = board.size();
    int n = m ? board[0].size() : m;
    if (m * n == 0)
        return finalrst;
    vector<vector<int>> visited(m, vector<int>(n, 0));
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < n; j++)
        {
            dfs(board, i, j, m, n, root, crtrst, finalrst, visited);
        }
    }
    destroy(root);
    return finalrst;
}
```

Restore IP Address

Idea: DFS.

1. 把当前的字母 `c` 放入当前小节 `value`；
2. 如果 `value` 数值上大于255或者开头为0且多于1位，则 `value` 不合法，返回；
3. 如果当前 `value` 合法，则
    - 当前 `value` 作为一节，dfs剩下部分；
    - 继续看下一个字母
4. 如果直到整个字符串都扫完且剩下点数为-1，则说明目前结果合法。

```
void dfs(string& s, int k, string value, int ndot,
    string& crtrst, vector<string>& finalrst)
{
    int n = s.size();
    if (ndot == -1 && k == n)
    {
        finalrst.push_back(crtrst);
        return;
    }
    if (ndot == -1 && k != n || ndot != -1 && k == n)
        return;
    char c = s[k];
    value += c;
    int val = stoi(value);
    if (val > 255 || (value.size() > 1 && value[0] == '0'))
        return;
    int origsz = crtrst.size();
    crtrst += value;
    if (ndot > 0)
        crtrst += '.';
    dfs(s, k + 1, "", ndot - 1, crtrst, finalrst);
    crtrst.resize(origsz);
    dfs(s, k + 1, value, ndot, crtrst, finalrst);
}
vector<string> restoreIpAddresses(string s)
{
    int n = s.size();
    vector<string> finalrst;
    string crtrst;
    string value = "";
    if (n == 0)
        return finalrst;
    dfs(s, 0, value, 3, crtrst, finalrst);
    return finalrst;
}
```

Word Break II

Idea: Memorization. A traditional DFS cannot pass, because it will cause duplicate computation.

```cpp
vector<string> dfs(string& s, unordered_set<string>& dict, int maxLen)
{
    auto it = map.find(s);
    if (it != map.end())
        return it->second;
    vector<string> rst;
    for (int i = 1; i <= s.size(); i++)
    {
        string s1 = s.substr(0, i);
        if (s1.size() > maxLen)
            break;
        if (dict.count(s1) == 0)
            continue;
        string s2 = s.substr(i);
        if (s2.empty())
            rst.push_back(s1);
        vector<string> tmprst = dfs(s2, dict, maxLen);
        for (int j = 0; j < tmprst.size(); j++)
            rst.push_back(s1 + " " + tmprst[j]);
    }
    map.insert({ s, rst });
    return rst;
}
vector<string> wordBreak(string s, unordered_set<string>& wordDict)
{
    int n = s.size();
    if (n == 0)
        return vector<string>(0);
    int maxLen = 0;
    for (auto& x : wordDict)
        maxLen = max(maxLen, (int)x.size());
    return dfs(s, wordDict, maxLen);
}
unordered_map<string, vector<string>> map;
```

Linked List

> Summary
>
> 1. **Pre Pointer is a good idea**
> 2. **Dummy Node**

Merge Two Sorted Lists (Classic)

```cpp
ListNode* merge2SortedLists(ListNode* l1, ListNode* l2)
{
    ListNode dummy(0);
    ListNode* pre = &dummy;
    ListNode *p1 = l1, *p2 = l2;
    while (p1 && p2)
    {
        if (p1->val <= p2->val)
        {
            pre->next = p1;
            pre = p1;
            p1 = p1->next;
        }
        else
        {
            pre->next = p2;
            pre = p2;
            p2 = p2->next;
        }
    }
    pre->next = p1 ? p1 : p2;
    return dummy.next;
}
```

Merge k Sorted Lists

Idea: Divide and Conquer

Linked List Cycle / Linked List Cycle II

Idea:

1. Problem I: Fast and Slow Pointers.
2. Problem II: Fast and Slow Pointers, after `p` and `q` intersects, we set `p` as `head` and move again.

```cpp
// Problem I: in this version, p and q are initialized as head the same.
bool hasCycle(ListNode *head)
{
    if (head == NULL)
        return false;
    ListNode *p1 = head, *p2 = head;
    while (p1 && p2 && p2->next)
    {
        p1 = p1->next;
        p2 = p2->next->next;
        if (p1 == p2)
            return true;
    }
    return false;
}
// Problem II: in this version, p and q are initialized differently
ListNode *detectCycle(ListNode *head)
{
    if (head == NULL)
        return NULL;
    ListNode *p = head, *q = head->next;
    while (p && q && q->next && p != q)
    {
        p = p->next;
        q = q->next->next;
    }
    if (p != q)
        return NULL;
    q = q->next;
    p = head;
    while (p != q)
    {
        p = p->next;
        q = q->next;
    }
    return p;
}
```

Rotate List

Idea: Note that we must compute the length of the list first and then judge right shift by mod.

```
void computeLength(ListNode* head, int& len, ListNode*& tail)
{
    ListNode* p = head;
    len = 0;
    while (p != NULL)
    {
        len++;
        tail = p;
        p = p->next;
    }
}
ListNode* rotateRight(ListNode* head, int k)
{
    if (head == NULL)
        return NULL;
    int len = 0;
    ListNode *tail = NULL;
    computeLength(head, len, tail);
    k = k % len;
    if (k == 0)
        return head;
    ListNode dummy(0);
    dummy.next = head;
    ListNode *p = head, *q = head, *pre = &dummy;
    for (int i = 0; i < k; i++)
        p = p->next;
    while (p != NULL)
    {
        p = p->next;
        pre = q;
        q = q->next;
    }
    tail->next = head;
    pre->next = NULL;
    return q;
}
```

Insertion Sort List

Idea: Explanation in the code.

```
ListNode* insertionSortList(ListNode* head)
{
    if (head == NULL || head->next == NULL)
        return head;
    ListNode dummy(0);
    dummy.next = head;
    ListNode *q = head->next, *qpre = head;
    while (q)
    {
        // Step 1. Every time, from dummy.next to qpre, it is must be sorted.
        // if we find qpre <= q, we know it needs no insertation, so go ahead
        // until we find qpre > q.
        while (q && qpre->val <= q->val)
        {
            qpre = q;
            q = q->next;
        }
        // if until end, we find qpre <= q, then finish.
        if (q == NULL)
            break;
        // Step 2. Otherwise, we will loop from the beginning to find the element > q.
        // and it is sure to find such an element, because we know at least qpre > q.
        ListNode *ppre = &dummy, *p = ppre->next;
        while (p->val <= q->val)
        {
            ppre = p;
            p = p->next;
        }
        // Step 3. Insert: change the corresponding pointers.
        qpre->next = q->next;
        ppre->next = q;
        q->next = p;
        q = qpre->next;
    }
    return dummy.next;
}
```

Partition list

Idea: Similar to Insertion Sort List.
Steps:

1. Find the 1st node `>= x`, noted as `p`.
2. Loop the nodes after `p`, if it is smaller than `x`, then insert just before `p`.

```
ListNode* partition(ListNode* head, int x)
{
    if (head == NULL)
        return NULL;
    ListNode dummy(0);
    dummy.next = head;
    ListNode *p = head, *pre = &dummy;
    // Find the first node >= x
    while (p && p->val < x)
    {
        pre = p;
        p = p->next;
    }
    if (p == NULL)
        return dummy.next;
    // q is the first node after p.
    ListNode *q = p->next, *qpre = p;
    while (q)
    {
        // if q >= x, continue
        while (q && q->val >= x)
        {
            qpre = q;
            q = q->next;
        }
        if (q == NULL)
            break;
        // insert q just before p
        qpre->next = q->next;
        pre->next = q;
        q->next = p;
        pre = q;
        q = qpre->next;
    }
    return dummy.next;
}
```

Sort List

Idea: Merge sort. Merge step in linked list is constant space. But in array, it is `O(n)` because we need to copy.

```
ListNode* merge(ListNode* p1, ListNode* p2)
{
    if (p1 == NULL || p2 == NULL)
        return p1 == NULL ? p2 : p1;
    ListNode dummy(0);
    ListNode *pre = &dummy;
    while (p1 && p2)
    {
        if (p1->val < p2->val)
        {
            pre->next = p1;
            pre = p1;
            p1 = p1->next;
        }
        else
        {
            pre->next = p2;
            pre = p2;
            p2 = p2->next;
        }
    }
    if (p1 || p2)
    {
        ListNode *tmp = p1 ? p1 : p2;
        pre->next = tmp;
    }
    return dummy.next;
}
ListNode* sortList(ListNode* head)
{
    if (head == NULL || head->next == NULL)
        return head;
    ListNode *p = head, *q = head, *pre = NULL;
    while (q != NULL && q->next != NULL)
    {
        pre = p;
        p = p->next;
        q = q->next->next;
    }
    pre->next = NULL;
    ListNode* p1 = sortList(head);
    ListNode* p2 = sortList(p);
    return merge(p1, p2);
}
```

Remove Duplicates from Sorted List / Remove Duplicates from Sorted List II

Idea: Check `p` and `p->next` equal or not, once we find they are equal, we loop until we find the one non-equal to it. I and II are very similar.

```cpp
// Problem I:
ListNode* deleteDuplicates(ListNode* head)
{
    if (head == NULL)
        return NULL;
    ListNode dummy(0);
    dummy.next = head;
    ListNode *p = head, *pre = &dummy;
    while (p && p->next)
    {
        if (p->val == p->next->val)
        {
            pre = p;
            int val = p->val;
            p = p->next;
            while (p && p->val == val)
            {
                ListNode* tmp = p;
                p = p->next;
                delete tmp;
            }
            pre->next = p;
        }
        else
        {
            pre = p;
            p = p->next;
        }
    }
    return dummy.next;
}
// Problem II:
ListNode* deleteDuplicates(ListNode* head)
{
    if (head == NULL)
        return NULL;
    ListNode dummy(0);
    dummy.next = head;
    ListNode *p = head, *pre = &dummy;
    while (p && p->next)
    {
        if (p->val == p->next->val)
        {
            int val = p->val;
            while (p && p->val == val)
            {
                ListNode *tmp = p;
                p = p->next;
                delete tmp;
            }
            pre->next = p;
        }
        else
        {
            pre = p;
            p = p->next;
        }
    }
    return dummy.next;
}
```

Palindrome Linked List

Idea:

1. Split the linked list into two half.
2. Reverse the second half.
3. Compare.

```
ListNode* reverseList(ListNode* head); // See "Reverse Linked List"
bool isPalindrome(ListNode* head)
{
    if (head == NULL || head->next == NULL)
        return true;
    ListNode *p = head, *q = head;
    while (q && q->next && q->next->next)
    {
        p = p->next;
        q = q->next->next;
    }
    q = p->next;
    p->next = NULL;
    q = reverseList(q);
    p = head;
    while (p && q && p->val == q->val)
    {
        p = p->next;
        q = q->next;
    }
    return q == NULL;
}
```

Reverse Linked List

Idea: Standard template.

```
ListNode* reverseList(ListNode* head)
{
    ListNode *p = head, *q = NULL, *pre = NULL;
    while (p)
    {
        q = p->next;
        p->next = pre;
        pre = p;
        p = q;
    }
    return pre;
}
```

Reverse Linked List II

Idea: 关键的节点：`m - 1`, `m`, `n`, `n + 1`, 因为会变成：`m - 1 ==> n, m ==> n + 1, m ~ n reversed`.

1. Find `m - 1` th node.
2. From `m` th to `n` th node, reverse. After the loop, `pre` points to `n` th, `p` points to `n + 1` th node.
3. `m - 1` th node points to `pre`, `m` th node points to `p`.

```cpp
ListNode* reverseBetween(ListNode* head, int m, int n)
{
    if (head == NULL)
        return NULL;
    ListNode dummy(0);
    dummy.next = head;
    ListNode *p = head, *pre = &dummy;
    for (int i = 0; i < m - 1; i++)
        pre = pre->next;
    ListNode *p0 = pre, *p1 = pre->next;
    p = p1;
    for (int i = 0; i <= n - m; i++)
    {
        ListNode *q = p->next;
        p->next = pre;
        pre = p;
        p = q;
    }
    p0->next = pre;
    p1->next = p;
    return dummy.next;
}
```

Remove Linked List Elements

Idea: Use dummy node.

```cpp
ListNode* removeElements(ListNode* head, int val)
{
    if (head == NULL)
        return head;
    ListNode dummy(0);
    dummy.next = head;
    ListNode *p = head, *pre = &dummy;
    while (p)
    {
        if (p->val == val)
        {
            pre->next = p->next;
            delete p;
            p = pre->next;
        }
        else
        {
            pre = p;
            p = p->next;
        }
    }
    return dummy.next;
}
```

Intersection of Two Linked Lists

Idea: Fast and Slow Pointer.

```c
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB)
{
    if (headA == NULL || headB == NULL)
        return NULL;
    ListNode *p1 = headA, *p2 = headB;
    int count = 0, flag = 0;
    while (p1 || p2)
    {
        if (p1 == NULL || p2 == NULL)
        {
            flag = p1 != NULL;
            count++;
        }
        p1 = p1 ? p1->next : p1;
        p2 = p2 ? p2->next : p2;
    }
    if (flag)
        swap(headA, headB);
    p1 = headA;
    p2 = headB;
    while (count)
    {
        p2 = p2->next;
        count--;
    }
    while (p1 && p2 && p1 != p2)
    {
        p1 = p1->next;
        p2 = p2->next;
    }
    return p1;
}
```

Array / String

H-Index / H-Index II

Idea: Sort the citations. Then, there is a rule: if `citations[i] >= n - i`, it means h-index is at least `n - i`. From `n - 1` to `0`, find the maxval of `n - i`.

```cpp
int hIndex(vector<int>& citations)
{
    int n = citations.size();
    if (n == 0)
        return 0;
    sort(citations.begin(), citations.end());
    int i = n - 1;
    for (; i >= 0 && citations[i] >= n - i; i--); // <== can be solved by binary searc
h.
    return n - i - 1;
}
// Use binary search to do find the result.
int hIndex(vector<int>& citations)
{
    int n = citations.size();
    if (n == 0)
        return 0;
    sort(citations.begin(), citations.end());
    int s = 0, e = n - 1, mid;
    while (s + 1 < e)
    {
        mid = s + (e - s) / 2;
        if (citations[mid] >= n - mid)
            e = mid;
        else
            s = mid;
    }
    if (citations[s] >= n - s)
        return n - s;
    else if (citations[e] >= n - e)
        return n - e;
    else
        return n - e - 1;
}
```

Maximum Gap

Idea: `O(n)` solution must use **Bucket Sort**. The idea is that, if we put the elements into several buckets, and keep the two elements with maximum gap cannot be put into the same bucket, then we just check the difference between two buckets. So the question is, how can we make sure the two elements cannot put into the same bucket? The answer is `bucketSize = ceil(Maxval - Minval) / (n - 1)`.

> Proof: The total gap is `(Maxval - Minval)`, and there are `(n - 1)` gaps now. `MaxGap * (n - 1) >= (Maxval - Minval) => MaxGap >= (Maxval - Minval) / (n - 1)`. However, the maximum gap within a bucket is `bucketSize - 1`. So `bucketSize` can be set to `ceil(Maxval - Minval) / (n - 1)`.

```cpp
class Bucket
{
public:
    Bucket() : isempty(true) {}
    int minval;
    int maxval;
    bool isempty;
};

int maximumGap(vector<int> nums)
{
    int n = nums.size();
    if (n < 2)
        return 0;
    if (n == 2)
        return abs(nums[0] - nums[1]);
    auto it = minmax_element(nums.begin(), nums.end());
    int minval = *it.first, maxval = *it.second;
    int totalGap = maxval - minval;
    if (totalGap == 0)
        return 0;
    int bucketSize = (int)ceil((float)totalGap / (n - 1));
    int numBuckets = (maxval - minval) / bucketSize + 1;
    vector<Bucket> buckets(numBuckets);
    for (int i = 0; i < n; i++)
    {
        int bid = (nums[i] - minval) / bucketSize;
        if (buckets[bid].isempty)
        {
            buckets[bid].minval = buckets[bid].maxval = nums[i];
            buckets[bid].isempty = false;
        }
        else
        {
            buckets[bid].minval
                = min(buckets[bid].minval, nums[i]);
            buckets[bid].maxval
                = max(buckets[bid].maxval, nums[i]);
        }
    }
    int last_bid = 0; // bucket 0 is sure non-empty
                      // because of minval exists.
    int maxgap = INT_MIN;
    for (int i = 1; i < numBuckets; i++)
    {
        if (buckets[i].isempty)
            continue;
        else
        {
            maxgap = max(maxgap,
                buckets[i].minval - buckets[last_bid].maxval);
            last_bid = i;
        }
    }
    return maxgap;
}
```

Minimum Window Substring

Idea: Two Pointers - fast and slow. Key problem: how to know substring in src contains target in O(1) time? Use a `valid_len`. 当`srcHash`中某个字符出现次数还没达到target中时，`valid_len++`, 则一旦`valid_len`变为`target.size()`时，则满足题意了。同理，删除字符时，如果删除之后此字符出现次数小于target中时，`valid_len--`, 使得substring不再包含完整target中字符。

```
string minWindow(string s, string t)
{
    int hash[256] = {0};
    int m = s.size(), n = t.size();
    for (int i = 0; i < n; i++)
        hash[t[i]]++;
    int valid_len = 0;
    int hash_tmp[256] = {0};
    int i = 0, j = 0;
    int minLen = INT_MAX;
    string minStr;
    while (j < m)
    {
        while (j < m && valid_len < t.size())
        {
            hash_tmp[s[j]]++;
            if (hash_tmp[s[j]] <= hash[s[j]])
                valid_len++;
            j++;
        }
        while (valid_len == t.size() && i < j)
        {
            if (j - i < minLen)
            {
                minLen = j - i;
                minStr = s.substr(i, j - i);
            }
            hash_tmp[s[i]]--;
            if (hash_tmp[s[i]] < hash[s[i]])
                valid_len--;
            i++;
        }
    }
    return minStr;
}
```

Longest Palindromic Substring

Idea: Simple idea is using DP to calculate whether substring `[i ~ j]` is palindrome.
An advanced idea is Manacher's algorithm. See below. Explanation here.

```cpp
string LPS_Manacher(const string& a)
{
    // 1. abc => #a#b#c#
    int n = a.size();
    string s(2 * n + 1, '#');
    for (int i = 1; i < 2 * n + 1; i += 2)
        s[i] = a[(i - 1) / 2];
    n = s.size();
    // 2. Pi stores how many elements right or left of i (not including i).
    vector<int> P(n, 0);
    // 3. an initial palindrome is located at: center 0, right bound is center + P0,
    // where P0 is 0.
    int center = 0, right = center + P[0];
    for (int i = 1; i < n; i++)
    {
        // for each index i, calculate its mirror_i of the center
        int mirror_i = 2 * center - i;
        // if i < right, meaning that, the array's center within right,
        // Pi will be: P[mirror_i] or at least right - i.
        // otherwise, we have no information of Pi, setting it 0
        P[i] = i < right ? min(P[mirror_i], right - i) : 0;
        // expand in two directions to update Pi.
        while (i - P[i] - 1 >= 0 && i + P[i] + 1 < n
            && s[i + P[i] + 1] == s[i - P[i] - 1])
            P[i]++;
        // if centering at i and right bound is out of current right bound,
        // update the center and right bound.
        if (i + P[i] > right)
        {
            center = i;
            right = i + P[i];
        }
    }

    int maxlen = 0, center_maxlen = 0;
    for (int i = 0; i < n; i++)
    {
        if (P[i] > maxlen)
        {
            center_maxlen = i;
            maxlen = P[i];
        }
    }
    return a.substr((center_maxlen - P[center_maxlen]) / 2, maxlen);
}
```

Even if we use `O(n^2)` DP method, there may be memory limit exceeded or time limit exceeded error. So the following code could pass LeetCode OJ.

```cpp
string longestPalindrome(string s)
{
    int n = s.size();
    if (n <= 1)
        return s;
    if (n == 2)
        return s[0] == s[1] ? s : s.substr(0, 1);
    bool P[n][n]; // Note Here, vector<vector<bool>> cannot pass.
    for (int i = 0; i < n; i++)
        P[i][i] = 1;
    for (int i = 0; i < n - 1; i++)
        P[i][i + 1] = s[i] == s[i + 1];
    int i = 0, j = 2, k = 3;
    while (j < n)
    {
        P[i][j] = P[i + 1][j - 1] && s[i] == s[j];
        i++;
        j++;
        if (j == n)
        {
            i = 0;
            j = k++;
        }
    }
    string rst;
    int maxlen = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j = i; j < n; j++)
        {
            if (P[i][j] && j - i + 1 > maxlen)
            {
                maxlen = j - i + 1;
                rst = s.substr(i, maxlen);
            }
        }
    }
    return rst;
}
```

Continuous Subarray Sum

Idea: Prefix Sum. Keep minimal prefix sum up to now, and maximal sum.

```cpp
vector<int> continuousSubarraySum(vector<int>& A)
{
    int n = A.size();
    vector<int> rst;
    if (n == 0)
        return rst;
    vector<int> P(n + 1, 0);
    for (int i = 1; i <= n; i++)
        P[i] = P[i - 1] + A[i - 1];
    int minPIndex = 0, minP = P[0], maxSum = INT_MIN;
    int start = -1, end = -1;
    for (int i = 1; i <= n; i++)
    {
        int sum = P[i] - minP;
        if (sum > maxSum)
        {
            start = minPIndex;
            end = i - 1;
            maxSum = sum;
        }
        if (P[i] < minP)
        {
            minPIndex = i;
            minP = P[i];
        }
    }
    if (start == -1)
        return rst;
    rst.resize(2);
    rst[0] = start;
    rst[1] = end;
    return rst;
}
```

Continuous Subarray Sum II

Idea: find `max_subarray` and `min_subarray`. if `TotalSum - Sum_of_min_subarray > Sum_max_subarray`, then there should be a discontinuous array. Otherwise, it is continuous.
There are two solutions, find at here - two parts solution and here - duplicate array solution.

Submatrix Sum

Idea: Store the column sum. Then for two rows, get the column sum difference, which is the column sum between these two rows. At last, use `subarray sum` to solve it. `O(n^3)`.

```cpp
vector<int> getSum(vector<vector<int>>& prefixSum, int s, int e)
{
    int m = prefixSum.size();
    int n = m ? prefixSum[0].size() : m;
    vector<int> rst;
    if (m * n == 0)
        return rst;
    rst.resize(n, 0);
    for (int i = 0; i < n; i++)
        rst[i] = prefixSum[e][i] - prefixSum[s][i];
    return rst;
}

vector<vector<int>> subarraySum(vector<int>& A)
{
    int n = A.size();
    vector<int> prefixSum(n + 1, 0);
    for (int i = 1; i <= n; i++)
        prefixSum[i] = prefixSum[i - 1] + A[i - 1];
    unordered_map<int, int> hash;
    vector<vector<int>> rst;
    for (int i = 0; i <= n; i++)
    {
        auto it = hash.find(prefixSum[i]);
        if (it == hash.end())
            hash.insert(pair<int,int>(prefixSum[i], i));
        else
        {
            vector<int> temp = {it->second, i - 1};
            rst.push_back(temp);
        }
    }
    return rst;
}

vector<vector<int>> submatrixSum(vector<vector<int>>& matrix)
{
    int m = matrix.size();
    int n = m ? matrix[0].size() : m;
    vector<vector<int>> rst;
    if (m * n == 0)
        return rst;
    vector<vector<int>> prefixSum(m + 1, vector<int>(n, 0));
    for (int i = 1; i <= m; i++)
    {
        for (int j = 0; j < n; j++)
        {
            prefixSum[i][j] = prefixSum[i - 1][j] + matrix[i - 1][j];
        }
    }

    for (int s = 0; s <= m; s++)
    {
        for (int e = s + 1; e <= m; e++)
        {
            vector<int> Sum = getSum(prefixSum, s, e);
            vector<vector<int>> tempRst = subarraySum(Sum);
            for (int k = 0; k < tempRst.size(); k++)
            {
                vector<int> pt1 = {s, tempRst[k][0]};
```

```
                vector<int> pt2 = {e - 1, tempRst[k][1]};
                //if (s == e - 1 && tempRst[k][0] == tempRst[k][1])
                //    continue;
                rst.push_back(pt1);
                rst.push_back(pt2);
                return rst;
            }
        }
    }
    return rst;
}
```

Bounding Box Sum

Idea: Note that: integral image's computation, and how to get a valid bounding box.

```cpp
#include <vector>
#include <iostream>

using namespace std;

vector<vector<int>> integImg(const vector<vector<int>>& A)
{
    int m = A.size();
    int n = m ? A[0].size() : m;
    if (m * n == 0)
        return A;
    vector<vector<int>> rst(m, vector<int>(n, 0));
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (i == 0 && j == 0)
                rst[i][j] = A[i][j];
            else if (i == 0 && j > 0)
                rst[i][j] = rst[i][j - 1] + A[i][j];
            else if (i > 0 && j == 0)
                rst[i][j] = rst[i - 1][j] + A[i][j];
            else
                rst[i][j] = rst[i - 1][j] + rst[i][j - 1]
                    - rst[i - 1][j - 1] + A[i][j];
        }
    }
    return rst;
}

bool getValidBoundingBox(int& i0, int& j0, int& i1, int& j1, int m, int n)
{
    if (i0 >= m || j0 >= n)
        return false;
    if (i1 < 0 || j1 < 0)
        return false;
    i0 = max(0, i0);
    j0 = max(0, j0);
    i1 = min(m - 1, i1);
    j1 = min(n - 1, j1);
    return true;
}

int boundingBoxSum(vector<vector<int>>& integA, int i, int j, int height, int width)
{
    int m = integA.size();
    int n = m ? integA[0].size() : m;
    if (m * n == 0)
        return 0;
    if (height <= 0 || width <= 0)
        return 0;
    int i0 = i, j0 = j, i1 = i + height - 1, j1 = j + width - 1;
    bool flag = getValidBoundingBox(i0, j0, i1, j1, m, n);
    if (flag == false)
        return 0;
    int sum11 = integA[i1][j1];
    int sum10 = j0 == 0 ? 0 : integA[i1][j0 - 1];
    int sum01 = i0 == 0 ? 0 : integA[i0 - 1][j1];
    int sum00 = (i0 == 0 || j0 == 0) ? 0 : integA[i0 - 1][j0 - 1];
    return sum11 - sum10 - sum01 + sum00;
```

```
}

int main(int argc, char const *argv[])
{
    vector<vector<int>> A = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
    for (vector<int>& x : A)
    {
        for (int y : x)
            cout << y << " ";
        cout << endl;
    }
    cout << endl;
    vector<vector<int>> integA = integImg(A);
    for (vector<int>& x : integA)
    {
        for (int y : x)
            cout << y << " ";
        cout << endl;
    }
    int sum = boundingBoxSum(integA, -1, -1, 2, 2);
    cout << "sum: " << sum << endl;
    return 0;
}
```

A better solution:

1. `computeIntegral` and `getBoundingBoxSum` is 2D solution. (update: `O(n^2)`, get: `O(1)`);
2. `computeIntegral2` and `getBoundingBoxSum2` is 1D solution (update: `O(n)`, get: `O(n)`);

```cpp
vector<vector<int>> computeIntegral(vector<vector<int>>& A)
{
    int m = A.size();
    int n = m ? A[0].size() : m;
    if (m * n == 0)
    {
        vector<vector<int>> G;
        return G;
    }
    vector<vector<int>> G(m + 1, vector<int>(n + 1, 0));
    for (int i = 1; i <= m; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            G[i][j] = G[i - 1][j] + G[i][j - 1] - G[i - 1][j - 1] + A[i - 1][j - 1];
        }
    }
    return G;
}

int getBoundingBoxSum(vector<vector<int>>& G, int i1, int j1, int i2, int j2)
{
    int s00 = G[i1][j1];
    int s01 = G[i1][j2 + 1];
    int s10 = G[i2 + 1][j1];
    int s11 = G[i2 + 1][j2 + 1];
    int result = s11 - s01 - s10 + s00;
    return result;
}

vector<vector<int>> computeIntegral2(vector<vector<int>>& A)
{
    int m = A.size();
    int n = m ? A[0].size() : m;
    if (m * n == 0)
    {
        vector<vector<int>> G;
        return G;
    }
    vector<vector<int>> G(m, vector<int>(n + 1, 0));
    for (int i = 0; i < m; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            G[i][j] = G[i][j - 1] + A[i][j - 1];
        }
    }
    return G;
}

int getBoundingBoxSum2(vector<vector<int>>& G, int i1, int j1, int i2, int j2)
{
    int sum = 0;
    for (int i = i1; i <= i2; i++)
    {
        sum += G[i][j2 + 1] - G[i][j1];
    }
    return sum;
}

```

```cpp
int main()
{
    vector<vector<int>> A = { { 1, 2, 3, 4 }, { 1, 2, 3, 4 }, { 1, 2, 3, 4 } };
    vector<vector<int>> G = computeIntegral2(A);
    int rst = getBoundingBoxSum2(G, 0, 0, 2, 2);
    cout << rst << endl;
    for (auto x : G)
    {
        for (auto y : x)
            cout << y << " ";
        cout << endl;
    }
    system("pause");
    return 0;
}
```

Longest Substring Without Repeating Characters

Idea: Two Pointers. Using `vector` to represent hash map will make it fast.

```cpp
int lengthOfLongestSubstring(string s)
{
    int n = s.size();
    if (n == 0)
        return 0;
    int hash[256] = {0};
    int i = 0, j = 0;
    int maxlen = 0;
    while (j < n)
    {
        while (j < n && hash[s[j]] == 0)
        {
            hash[s[j]]++;
            j++;
        }
        if (j == n)
        {
            maxlen = max(maxlen, j - i);
            break;
        }
        while (i < j && hash[s[j]] > 0)
        {
            maxlen = max(maxlen, j - i);
            hash[s[i]]--;
            i++;
        }
    }
    return maxlen;
}
```

Longest Substring with At Most K Distinct Characters

Idea: `valid_len` to track whether contains enough characters.

```
int lengthOfLongestSubstringKDistinct(string s, int k)
{
    int n = s.size();
    if (n <= k)
        return n;
    if (k == 0)
        return 0;
    int i = 0, j = 0, valid_len = 0;
    int hash[256] = { 0 };
    int maxlen = 0;
    int ds, de;
    while (j < n)
    {
        while (j < n && valid_len <= k)
        {
            if (hash[s[j]] == 0 && valid_len == k)
                break;
            if (hash[s[j]] == 0)
                valid_len++;
            hash[s[j]]++;
            j++;
            maxlen = max(maxlen, j - i);
        }
        while (i < j && valid_len >= k)
        {
            if (hash[s[i]] == 1)
                valid_len--;
            hash[s[i]]--;
            i++;
        }
    }
    return maxlen;
}
```

Subarray Sum (Classic)

Idea: Prefix Sum.

```
A |_|_|_|_|_|_|
P |_|_|_|_|_|_|_|
```

`P[i]` means first `i` element sum, `A[0 ~ i-1]`. So if `P[j] == P[i]`, it means first `i` elements and first `j` elements are of the same sum, `=> A[i] ~ A[j - 1]` sum equal to `0`.

1. `Sum[i,...,j] = A[i] ~ A[j] => P[j + 1] - P[i]`
2. `P[j] - P[i] => A[i] ~ A[j - 1]`

记住：`P[i]`为`A`中前`i`个元素的和

Minimum Size Subarray Sum

Idea: Two Pointers. Three `while` loops. `sum` is defined as the range `[i, j)`.

```cpp
int minSubArrayLen(int s, vector<int>& nums)
{
    int n = nums.size();
    if (n == 0)
        return 0;
    int minlen = INT_MAX;
    int i = 0, j = 0;
    int sum = 0;
    while (j < n)
    {
        while (j < n && sum < s)
        {
            sum += nums[j];
            j++;
        } // After each step, j points to the element excluded from the range.
        while (i < j && sum >= s)
        {
            minlen = min(minlen, j - i);
            sum -= nums[i];
            i++;
        } // After each step, i points to the element included from the range (when i < j)
    }
    return minlen == INT_MAX ? 0 : minlen;
}
```

Remove Duplicates from Sorted Array

Idea: In-place copy, two pointers.

```cpp
int removeDuplicates(vector<int>& nums)
{
    int n = nums.size();
    if (n == 0)
        return 0;
    int i = 0, j = 0;
    while (j < n)
    {
        if (j < n - 1 && nums[j] == nums[j + 1])
        {
            int temp = nums[j];
            while (j < n && nums[j] == temp) j++;
            nums[i++] = temp;
        }
        else
            nums[i++] = nums[j++];
    }
    return i;
}
```

Partition Array (Classic!!! Must Check Again!!!)

Idea: This is the partition step in quick sort.

```cpp
void partition(vector<int>& A, int k)
{
    int n = A.size();
    int i = 0, j = n - 1;
    while (i <= j)
    {
        while (i <= j && A[i] < k) i++;
        while (i <= j && A[j] >= k) j--;
        if (i <= j)
        {
            swap(A[i], A[j]);
            i++;
            j--;
        }
    }
    // The final i is the 1st element >= k
}
```

QuickSort Implementation

In QuickSort's `partition(A, s, e)`, the pivot can be `A[e]`. The code will be like this:

```
int partition(vector<int>& A, int s, int e)
{
    if (s >= e)
        return s;
    int i = s, j = e, pivot = A[e];
    while (i <= j)
    {
        while (i <= j && A[i] < pivot) i++;
        while (i <= j && A[j] >= pivot) j--;
        if (i <= j)
            swap(A[i], A[j]);
    }
    // Now A[i] is the 1st >= pivot, and we know A[e] == pivot exactly
    swap(A[i], A[e]); // to make A[i] == pivot
    return i; // A[i] == pivot, and A[i+1,...] >= pivot
}
// NOTE: None of the judgement i <= j cannot be replaced by i < j !!!
// See these examples to check:
// [790, 504, 277, 449, 662]
// [576, 927, 122, 138, 252]
// [139, 583, 324, 521, 545]
```

1. Recursive version

```
void QuickSort(vector<int>& A, int s, int e)
{
    if (s >= e)
        return;
    int index = partition(A, s, e);
    QuickSort(A, s, index - 1);
    QuickSort(A, index + 1, e);
}
```

2. Iterative version

```
void myQuickSort2(vector<int>& v, int s, int e)
{
    stack<pair<int, int>> S;
    S.push(pair<int, int>(s, e));
    while (!S.empty())
    {
        pair<int, int> params = S.top();
        S.pop();
        int ss = params.first, ee = params.second;
        int index = partition(v, ss, ee);
        if (index - 1 > ss)
            S.push(pair<int, int>(ss, index - 1));
        if (index + 1 < ee)
            S.push(pair<int, int>(index + 1, ee));
    }
}
```

Sort Colors

Idea: Quick Sort. Code is the same as `QuickSort`.

Kth Largest Element in an Array

Idea: 1. Transform it into find (n + 1 - k)th smallest. 2. Partition Array.

```
int partition(vector<int>& A, int s, int e)
{
    if (s >= e)
        return s;
    int pivot = A[e];
    int i = s, j = e;
    while (i <= j)
    {
        while (i <= j && A[i] < pivot) i++;
        while (i <= j && A[j] >= pivot) j--;
        if (i <= j)
            swap(A[i], A[j]);
    }
    swap(A[i], A[e]);
    return i;
}
int findKthSmallest(vector<int>& A, int s, int e, int k)
{
    int index = partition(A, s, e);
    int kthSmallest = index - s + 1;
    if (kthSmallest == k)
        return A[index];
    else if (kthSmallest < k)
        return findKthSmallest(A, index + 1, e, k - kthSmallest);
    else
        return findKthSmallest(A, s, index - 1, k);
}
int findKthLargest(vector<int>& nums, int k)
{
    int n = nums.size();
    if (n == 0)
        return INT_MIN;
    return findKthSmallest(nums, 0, n - 1, n + 1 - k);
}
```

Sort Letters by Case

Idea: Partition array.

```
void sortLetters(string &letters)
{
    int n = letters.size();
    if (n == 0)
        return;
    int i = 0, j = n - 1;
    while (i <= j)
    {
        while (i <= j && letters[i] <= 'z' && letters[i] >= 'a')
            i++;
        while (i <= j && letters[j] >= 'A' && letters[j] <= 'Z')
            j--;
        if (i <= j)
            swap(letters[i], letters[j]);
    }
}
```

Nuts & Bolts Problem

Idea: Quick-Sort. Use nuts to partition bolts, and then use bolts to partition nuts.

```cpp
int Partition(vector<string>& nuts, vector<string>& bolts,
    int start, int end, Comparator compare)
{
    string pivot = nuts[start];
    int i = start, j = end, k = -1;
    while (i <= j)
    {
        while (i <= j && compare.cmp(pivot, bolts[i]) >= 0)
        {
            if (compare.cmp(pivot, bolts[i]) == 0)
                k = i;
            i++;
        }
        while (i <= j && compare.cmp(pivot, bolts[j]) < 0)
            j--;
        if (i <= j)
            swap(bolts[i], bolts[j]);
    }
    swap(bolts[k], bolts[j]);
    int index = j;
    pivot = bolts[index];
    i = start, j = end, k = -1;
    while (i <= j)
    {
        while (i <= j && compare.cmp(nuts[i], pivot) <= 0)
        {
            if (compare.cmp(nuts[i], pivot) == 0)
                k = i;
            i++;
        }
        while (i <= j && compare.cmp(nuts[j], pivot) > 0)
            j--;
        if (i <= j)
            swap(nuts[i], nuts[j]);
    }
    swap(nuts[k], nuts[j]);
    return index;
}

void QuickSort(vector<string>& nuts, vector<string>& bolts,
    int start, int end, Comparator compare)
{
    if (start >= end)
        return;
    int index = Partition(nuts, bolts, start, end, compare);
    QuickSort(nuts, bolts, start, index - 1, compare);
    QuickSort(nuts, bolts, index + 1, end, compare);
}

void sortNutsAndBolts(vector<string> &nuts, vector<string> &bolts, Comparator compare)
{
    int n1 = nuts.size(), n2 = bolts.size();
    if (n1 != n2 || n1 == 0)
        return;
    int n = n1;
    QuickSort(nuts, bolts, 0, n - 1, compare);
}
```

Trapping Rain Water

Idea:

1. **Two Pointers -** 对撞型指针. 左右两根指针，其小者控制了盛水的上界。哪个小，就移动哪个；水量为此指针右边或者左边的高度与此指针的高度差。
2. 递减索引栈. 当前柱子可以贡献的水量为：`max(0, min(rightHeight, leftHeight) - curHeight) * (right - left - 1)`.

Container With Most Water

Idea: Two Pointers, similar to Trapping Rain Water.

```cpp
int maxArea(vector<int> &heights)
{
    int n = heights.size();
    if (n <= 1)
        return 0;
    int i = 0, j = n - 1;
    int leftMaxHeight = heights[i], rightMaxHeight = heights[j];
    int maxarea = 0, area;
    while (i < j)
    {
        while (i < j && leftMaxHeight <= rightMaxHeight)
        {
            area = (j - i) * leftMaxHeight;
            maxarea = max(maxarea, area);
            i++;
            leftMaxHeight = max(leftMaxHeight, heights[i]);
        }
        while (i < j && rightMaxHeight < leftMaxHeight)
        {
            area = (j - i) * rightMaxHeight;
            maxarea = max(maxarea, area);
            j--;
            rightMaxHeight = max(rightMaxHeight, heights[j]);
        }
    }
    return maxarea;
}

// A Simpler code:
int maxArea(vector<int>& height)
{
    int n = height.size();
    if (n <= 1)
        return 0;
    int i = 0, j = n - 1;
    int area = -1, max_area = INT_MIN;
    while (i < j)
    {
        while (i < j && height[i] <= height[j])
        {
            area = (j - i) * height[i];
            max_area = max(max_area, area);
            i++;
        }
        while (i < j && height[i] > height[j])
        {
            area = (j - i) * height[j];
            max_area = max(max_area, area);
            j--;
        }
    }
    return max_area;
}
```

Median of Two Sorted Arrays

Idea: Find median `==>` Find `kth` Large.

1. 找第 `k` 大 `==>` 找第 `k/2` 大，怎么把问题缩减一半？检查 `A[k/2]` 和 `B[k/2]` 谁小；当两个数组合并时

当小的那个进入时，大的那个肯定还没有进入合并数组，故说明合并数组的`size`少于`k`，所以可以放心的扔掉A的前`k/2`个数，这样就把问题规模缩减到了`k/2`.

2. Corner Cases: 如果`k/2`大于其中一数组长度，那么可以放心把另外一个数组的`k/2`部分全部去掉。

Maximum Subarray / Minimum Subarray

Idea: **PrefixSum**.

- Maximum Subarray:
  - Make three variables: `prefixSum`, `minPrefixSum`, `maxSubarraySum`;
  - `maxSubarraySum = prefixSum - minPrefixSum`
- Minimum Subarray, similar idea, only modify `minPrefixSum` and `maxSubarraySum` to `maxPrefixSum` and `minSubarraySum`.

```
int maxSubArray(vector<int>& nums)
{
    int n = nums.size();
    if (n == 0)
        return 0;
    vector<int> prefixSum(n + 1, 0);
    for (int i = 1; i <= n; i++)
        prefixSum[i] = prefixSum[i - 1] + nums[i - 1];
    int maxSum = INT_MIN, minPrefixSum = prefixSum[0];
    for (int i = 1; i <= n; i++)
    {
        int temp = prefixSum[i] - minPrefixSum;
        maxSum = max(maxSum, temp);
        minPrefixSum = min(minPrefixSum, prefixSum[i]);
    }
    return maxSum;
}
```

Count and Say

Idea: 注意统计连续相同字符的代码，需要`i < n - 1`且最终`i`指向当前连续相同字符的末尾。

```cpp
string next(string& s)
{
    int n = s.size();
    string rst;
    int count = 1;
    int i = 0;
    while (i < n)
    {
        while (i < n - 1 && s[i] == s[i + 1])
        {
            i++;
            count++;
        }
        char c = s[i];
        rst += to_string(count);
        rst += c;
        count = 1;
        i++;
    }
    return rst;
}
```

Interleaving Positive and Negative Numbers

Idea:

1. Partition array, left negative, right positive.
2. Check whether negative more than positive.
3. Interleave the array.

```cpp
int partition(vector<int>& A)
{
    int n = A.size();
    if (n == 0)
        return 0;
    int i = 0, j = n - 1;
    while (i <= j)
    {
        while (i <= j && A[i] < 0)
            i++;
        while (i <= j && A[j] > 0)
            j--;
        if (i <= j)
        {
            swap(A[i], A[j]);
        }
    }
    return j;
}
void interleave(vector<int>& A, int flag)
{
    int n = A.size();
    int i = flag, j = (n + flag) / 2;
    while (i != j)
    {
        swap(A[i], A[j]);
        j++;
        i += 2;
    }
}
void rerange(vector<int> &A)
{
    int n = A.size();
    if (n == 0)
        return;
    int num_neg = partition(A) + 1;
    int num_pos = n - num_neg;
    if (num_neg >= num_pos)
        interleave(A, 1);
    else
        interleave(A, 0);
}
```

Product of Array Except Self

Idea: Prefix-Product and Suffix-Product.

```cpp
vector<int> productExceptSelf(vector<int>& nums)
{
    int n = nums.size();
    if (n == 0)
        return vector<int>(0);
    vector<int> P(n + 1, 1), Q(n + 1, 1); // Pi: first i numbers' products, Qi: last i..
    for (int i = 1; i <= n; i++)
        P[i] = P[i - 1] * nums[i - 1];
    for (int i = 1; i <= n; i++)
        Q[i] = Q[i - 1] * nums[n - i];
    vector<int> rst(n, 0);
    for (int i = 0; i < n; i++)
        rst[i] = P[i] * Q[n - i - 1];
    return rst;
}
```

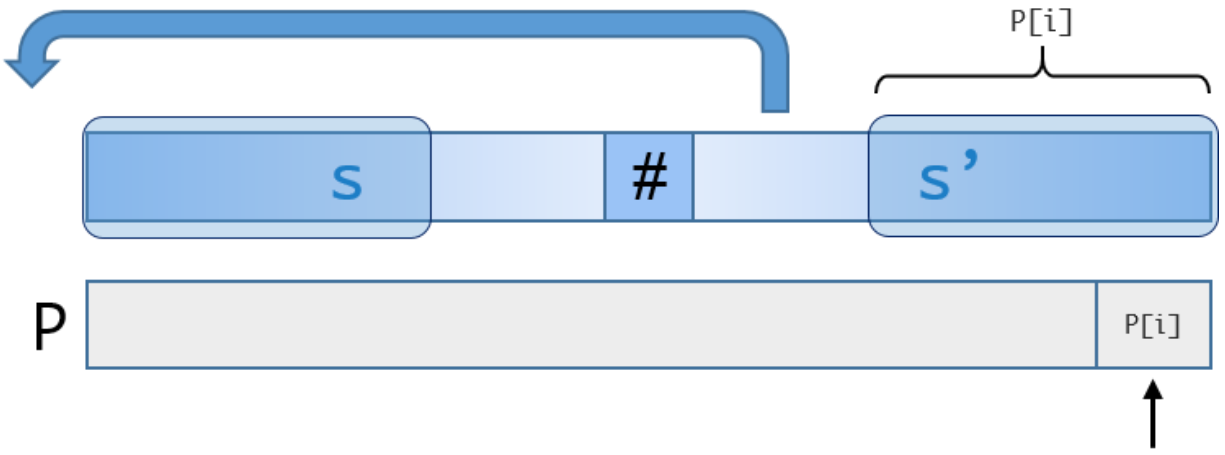
Largest Number

Idea: sort.

```cpp
struct TComp
{
    bool operator()(int a, int b) const
    {
        if (a == 0)
            return false;
        if (b == 0)
            return true;
        string sa = to_string(a), sb = to_string(b);
        string sasb = sa + sb, sbsa = sb + sa;
        long long nsasb = atoll(sasb.c_str());
        long long nsbsa = atoll(sbsa.c_str());
        return nsasb > nsbsa;
    }
};
string largestNumber(vector<int> &num)
{
    int n = num.size();
    if (n == 0)
        return "";
    TComp compobj;
    sort(num.begin(), num.end(), compobj);
    if (num[0] == 0)
        return "0";
    string s;
    for (int i = 0; i < num.size(); i++)
    {
        s += to_string(num[i]);
    }
    return s;
}
```

Next Permutation

Idea: 见图



s and s' are mirrored.

```cpp
vector<int> nextPermutation(vector<int> &nums)
{
    int n = nums.size();
    if (n == 0)
        return nums;
    int i = n - 2;
    while (i >= 0 && nums[i] >= nums[i + 1])
        i--;
    if (i == -1)
    {
        reverse(nums.begin(), nums.end());
        return nums;
    }
    int index = i; // nums[i] < nums[i + 1]
    for (i = index + 1; i < n && nums[i] > nums[index]; i++);
    int index2 = i - 1;
    swap(nums[index], nums[index2]);
    reverse(nums.begin() + index + 1, nums.end());
    return nums;
}
```

Gas Station

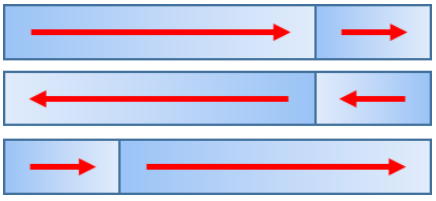
Idea:

1. $\sum_{i=0}^{N-1}(G_i - C_i)$, i.e. total gas `-` total cost, if the result `< 0`, then must not go the entire circle;
2. $\sum_{i=0}^{k}(G_i - C_i)$, we calculate the left gas up to `k + 1` station, once this value `< 0`, it means that the last $G_k - C_k$ is so negative, so that `0 ~ k` must not be the start point. We then start from `k + 1`.

```cpp
int canCompleteCircuit(vector<int> &gas, vector<int> &cost)
{
    int sum = 0, total = 0;
    int n = gas.size();
    if (n == 0)
        return true;
    int j = 0;
    for (int i = 0; i < n; i++)
    {
        int left = gas[i] - cost[i]; // the left gas after we pass station i
        sum += left; // the total left gas after we pass station i from feasible station.
        total += left;
        if (sum < 0) // once we find from feasible station the total left gas is < 0
        {
            j = i + 1; // we must start from i + 1, because it is impossible
            sum = 0;  // to start from the last feasible station.
        }
    }
    if (total < 0)
        return -1;
    else
        return j;
}
```

Rotate Array

Idea: Same as reverse words, i.e. reverse each word, then reverse the whole string.



```cpp
void rotate(vector<int>& nums, int k)
{
    int n = nums.size();
    if (n == 0)
        return;
    k = k % n;
    if (k == 0)
        return;
    reverse(nums.begin(), nums.begin() + n - k);
    reverse(nums.begin() + n - k, nums.end());
    reverse(nums.begin(), nums.end());
}
```

Reverse Words in a String

Idea: First make sure the string start and end with alphabet characters, and only single whitespace between them. Then reverse.

```
void reverseWords(string &s)
{
    int n = s.size();
    int i = 0, j = 0;
    // 1. make sure the string start with alphabet characters,
    // and single whitespace between them if there is.
    while (j < n)
    {
        if (s[j] == ' ' && (j == 0 || s[j - 1] == ' '))
            j++;
        else
        {
            s[i++] = s[j++];
        }
    }
    // Note here, there may be a whitespace at the end. So check it.
    if (s[i - 1] == ' ')
        i--;
    s.resize(i);
    n = i;
    // Reverse each word and then reverse the whole string.
    i = 0;
    j = 0;
    while (j < n)
    {
        while (j < n && s[j] != ' ') j++;
        reverse(s.begin() + i, s.begin() + j);
        j++;
        i = j;
    }
    reverse(s.begin(), s.end());
}
```

Largest Number

Idea: Note the case the biggest number is 0, where we will keep only one 0.

```
struct TComp
{
    bool operator()(const string& a, const string& b) const
    {
        string ab = a + b;
        string ba = b + a;
        int n = ab.size();
        for (int i = 0; i < n; i++)
            if (ab[i] != ba[i])
                return ab[i] > ba[i];
        return false;
    }
};
string largestNumber(vector<int>& nums)
{
    vector<string> vs;
    int n = nums.size();
    if (n == 0)
        return "";
    for (int i = 0; i < n; i++)
        vs.push_back(to_string(nums[i]));
    TComp compobj;
    sort(vs.begin(), vs.end(), compobj);
    string rst;
    if (vs[0] == "0")
        return "0";
    for (int i = 0; i < vs.size(); i++)
        rst += vs[i];
    return rst;
}
```

Rotate Image

Idea: Rotate Image by clockwise 90 degree can be accomplished by transpose and flip with x axis.

```cpp
void transpose(vector<vector<int>>& matrix)
{
    int n = matrix.size();
    for (int i = 0; i < n; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            swap(matrix[i][j], matrix[j][i]);
        }
    }
}
void flip_x(vector<vector<int>>& matrix)
{
    int n = matrix.size();
    for (int i = 0; i < n; i++)
    {
        int s = 0, e = n - 1;
        while (s < e)
        {
            swap(matrix[i][s], matrix[i][e]);
            s++;
            e--;
        }
    }
}
void rotate(vector<vector<int>>& matrix)
{
    int n = matrix.size();
    if (n == 0)
        return;
    transpose(matrix);
    flip_x(matrix);
}
```

Spiral Matrix

Idea: Note that when sprially loop a matrix of size `m x n`, the order is like:
Steps: `n(right), m - 1(down), n - 1(left), m - 2(up), ...` until any step achieves 0.

```cpp
vector<int> spiralOrder(vector<vector<int>>& matrix)
{
    int m = matrix.size();
    int n = m ? matrix[0].size() : m;
    int total = m * n;
    vector<int> rst(total, 0);
    if (total == 0)
        return rst;
    int steps[2] = { n, m - 1 };
    int shift[4][2] = { {0, 1}, {1, 0}, {0, -1}, {-1, 0} };
    int i = 0, j = -1, k = 0;
    int index = 0;
    while (steps[k % 2])
    {
        for (int t = 0; t < steps[k % 2]; t++)
        {
            i += shift[k % 4][0];
            j += shift[k % 4][1];
            rst[index++] = matrix[i][j];
        }
        steps[k % 2]--;
        k++;
    }
    return rst;
}
```

Group Anagrams

Idea: Make each string's sorted string as the key of a hashmap.

```cpp
vector<vector<string>> groupAnagrams(vector<string>& strs)
{
    vector<string> sortedstrs = strs;
    unordered_map<string, vector<int>> hashmap;
    for (int i = 0; i < strs.size(); i++)
    {
        sort(sortedstrs[i].begin(), sortedstrs[i].end());
        hashmap[sortedstrs[i]].push_back(i);
    }
    vector<vector<string>> rst(hashmap.size());
    int k = 0;
    for (auto it = hashmap.begin(); it != hashmap.end(); it++)
    {
        vector<string> tmp;
        for (int i = 0; i < it->second.size(); i++)
        {
            tmp.push_back(strs[it->second[i]]);
        }
        sort(tmp.begin(), tmp.end());
        rst[k++] = tmp;
    }
    return rst;
}
```

Compare Version Numbers

Idea: Note that `stringstream` return its state only after the operation `>>`. Also note the cases such as `1.0` and `1`, which is the same.

```
int compareVersion(string version1, string version2)
{
    replace(version1.begin(), version1.end(), '.', ' ');
    replace(version2.begin(), version2.end(), '.', ' ');
    stringstream ss1(version1), ss2(version2);
    int v1, v2;
    while (ss1 || ss2)
    {
        v1 = 0;
        ss1 >> v1;
        v2 = 0;
        ss2 >> v2;
        if (v1 != v2)
            break;
    }
    return v1 > v2 ? 1 : v1 < v2 ? -1 : 0;
}
```

ZigZag Conversion

Idea: The zigzag is like this:

```
/*n=numRows
Δ=2n-2    0                       2n-2                       4n-4
Δ=        1                 2n-3  2n-1              4n-5   4n-3
Δ=        2           2n-4        2n            4n-6       .
Δ=        .         .             .           .            .
Δ=        .     n+1               .       3n-1             .
Δ=        n-2 n+1                 3n-4  3n-2               5n-6
Δ=2n-2    n-1                     3n-3                     5n-5
*/
```

Two jumps method. At each row, it follows two jumps. For example, in row `1`, it jumps from `1 => 2n - 3 => 2n - 1`. First jump is `(n - i - 1) * 2`, second jump is `i * 2`. But note that, in row `0` or `n - 1`, the first jump are `(n - 1) * 2` and `0`, the second jump are `0` and `(n - 1) * 2`. If the jump is `0`, it means it actually don't jump. So in this case we should ignore.

```cpp
string convert(string s, int numRows)
{
    int n = s.size();
    if (numRows <= 1)
        return s;
    int step1, step2;
    string rst;
    for (int i = 0; i < numRows; i++)
    {
        step1 = 2 * (numRows - i - 1);
        step2 = 2 * i;
        if (i >= n)
            break;
        rst += s[i];
        int j = i;
        while (1)
        {
            j += step1;
            if (j >= n)
                break;
            if (step1)
                rst += s[j];
            j += step2;
            if (j >= n)
                break;
            if (step2)
                rst += s[j];
        }
    }
    return rst;
}
```

Implement strStr()

Idea: Just implement a `O(m*n)` solution is okay.

```cpp
int strStr(string haystack, string needle)
{
    int m = haystack.size(), n = needle.size();
    for (int i = 0; i <= m - n; i++)
    {
        if (needle == haystack.substr(i, n))
            return i;
    }
    return -1;
}
```

Word Pattern

Idea: Hash map. But note that:

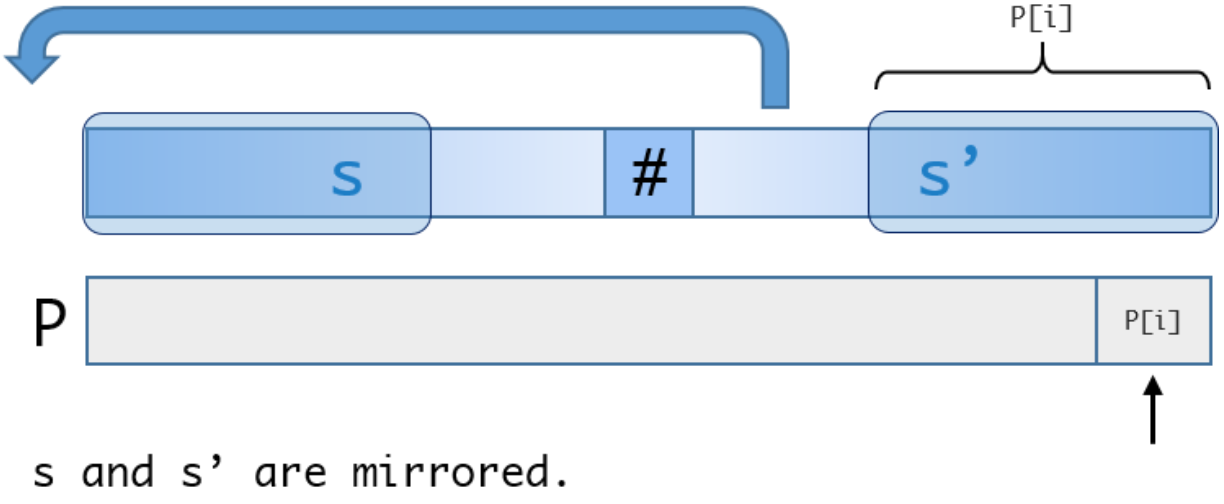
1. Use two hash map, forward and backward must be both okay;
2. Check there may be the case pattern and string non-equal length.

```cpp
bool wordPattern(string pattern, string str)
{
    unordered_map<char, string> hashMap1;
    unordered_map<string, char> hashMap2;
    int start = 0, end = 0;
    for (int i = 0; i < pattern.size(); i++)
    {
        char c = pattern[i];
        if (start >= str.size())
            return false;
        end = str.find(' ', start);
        string temp;
        if (end != string::npos)
            temp = str.substr(start, end - start);
        else
            temp = str.substr(start);
        start = end == string::npos ? str.size() : end + 1;
        auto it1 = hashMap1.find(c);
        auto it2 = hashMap2.find(temp);
        if (it1 == hashMap1.end() && it2 == hashMap2.end())
        {
            hashMap1[c] = temp;
            hashMap2[temp] = c;
        }
        else if (it1 == hashMap1.end() && it2 != hashMap2.end()
            || it1 != hashMap1.end() && it2 == hashMap2.end())
            return false;
        else
        {
            if (it1->second != temp || it2->second != c)
                return false;
        }
    }
    return start == str.size();
}
```

Shortest Palindrome (Hard!!! Check Again!!!)

Idea:

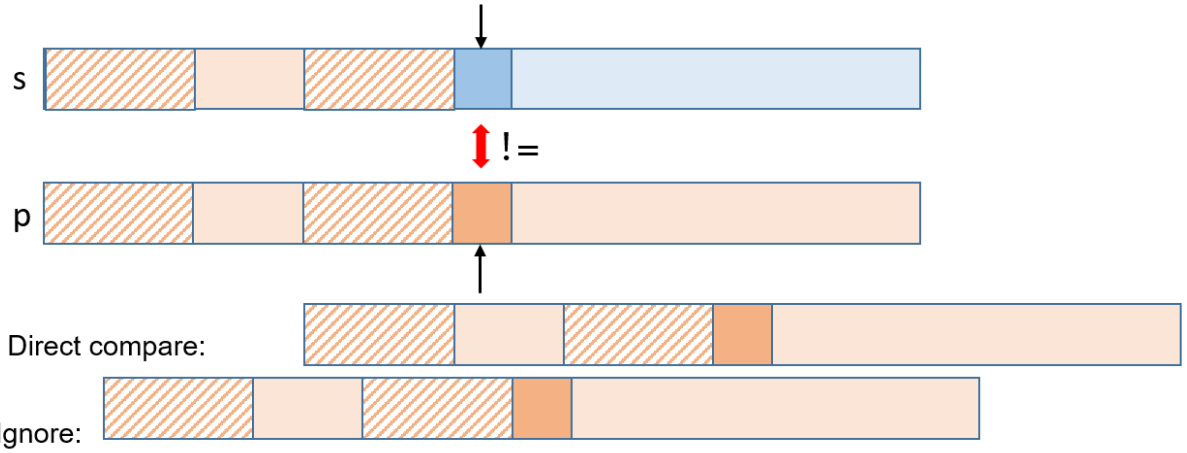
1. KMP algorithm, to calculate prefix and suffix sub-string maximum length.
2. Once we have the prefix and suffix sub-string maximum length, we apply it to `s '#' mirror_s`



```cpp
string shortestPalindrome(string s)
{
    int m = s.size();
    string t = s;
    reverse(t.begin(), t.end());
    t = s + "#" + t;
    vector<int> p(t.size(), 0);
    for (int i = 1; i < t.size(); i++)
    {
        int k = p[i - 1];
        while (k > 0 && t[k] != t[i])
            k = p[k - 1];
        p[i] = k + (t[k] == t[i]);
    }
    return t.substr(m + 1, m - p[2 * m]) + s;
}
```

Note: KMP algorithm:

1. When we match `p` in `s`, the trivial idea is that, while `s[i] == p[j]`, then `i++, j++`; once `s[i] != p[j]`, we will set `i` to `i - j + 1`, and `j` to `0`.
2. However, it is not necessary, see the figure below.



Because, if we need to compare this state, it means we believe, the overlapping part should be equal, which is ensured impossible. (otherwise, the prefix-suffix maximum length will be larger than current value.)

```cpp
int KMP(string s, string p)
{
    int m = s.size(), n = p.size();
    if (m < n)
        return -1;
    vector<int> P(n, 0); // Pi is the prefix-suffix max len array.
    for (int i = 1; i < n; i++)
    {
        int k = P[i - 1];
        while (k > 0 && p[k] != p[i])
            k = P[k - 1];
        P[i] = k + (p[k] == p[i]);
    }
    int i = 0, j = 0;
    while (i < m && j < n)
    {
        if (j == -1)
        {
            i++;
            j = 0;
        }
        if (s[i] == p[j])
        {
            i++;
            j++;
        }
        else
        {
            j = j > 0 ? P[j - 1] : -1;
        }
    }
    return j == n ? i - n : -1;
}
```

Candy

Idea: Two traverse.

1. From left to right, if find `a[i] > a[i - 1] && val[i] <= val[i - 1]`, `val[i] = val[i - 1] + 1`;
2. From right to left, if find `a[i] > a[i + 1] && val[i] <= val[i + 1]`, `val[i] = val[i + 1] + 1`.

```cpp
int candy(vector<int>& ratings)
{
    int n = ratings.size();
    vector<int> rst(n, 1);
    for (int i = 1; i < n; i++)
    {
        if (ratings[i] > ratings[i - 1] && rst[i] <= rst[i - 1])
            rst[i] = rst[i - 1] + 1;
    }
    for (int i = n - 2; i >= 0; i--)
    {
        if (ratings[i] > ratings[i + 1] && rst[i] <= rst[i + 1])
            rst[i] = rst[i + 1] + 1;
    }
    return accumulate(rst.begin(), rst.end(), 0, plus<int>());
}
```

Merge Intervals

Idea: Interval classic method:

1. Sort the original interval array `arr` according to the start point.
2. Traverse the original interval array `arr` and push back it into the resultant array `rst`.
    1. if `rst.empty() || rst.back().end < arr[i].start`, then `rst.push_back(arr[i])`;
    2. otherwise, make the `rst.back().end` to be the maximum value of itself and `arr[i].end`.

```
vector<Interval> merge(vector<Interval>& intervals)
{
    sort(intervals.begin(), intervals.end(),
        [](const Interval& a, const Interval& b)
        { return a.start < b.start; });
    vector<Interval> rst;
    int n = intervals.size();
    int i = 0;
    while (i < n)
    {
        if (rst.empty() || rst.back().end < intervals[i].start)
            rst.push_back(intervals[i]);
        else
            rst.back().end = max(rst.back().end, intervals[i].end);
        i++;
    }
    return rst;
}
```

Insert Interval

Idea: Interval classic method, as stated in Merge Intervals.

1. Pass the intervals whose end point is smaller than `newInterval`, directly pushing back them into the `rst` array.
2. Follow the **Interval classic method**, push `newInterval` into `rst`.
3. Follow the **Interval classic method**, push the rest into `rst`.

```cpp
vector<Interval> insert(vector<Interval>& intervals, Interval newInterval)
{
    int n = intervals.size();
    int i = 0;
    vector<Interval> rst;
    while (i < n && intervals[i].start <= newInterval.start)
    {
        rst.push_back(intervals[i]);
        i++;
    }
    if (rst.empty() || rst.back().end < newInterval.start)
        rst.push_back(newInterval);
    else
        rst.back().end = max(rst.back().end, newInterval.end);
    while (i < n)
    {
        if (rst.back().end < intervals[i].start)
            rst.push_back(intervals[i]);
        else
            rst.back().end = max(rst.back().end, intervals[i].end);
        i++;
    }
    return rst;
}
```

Follow up: What if the intervals are stored as Linked List?

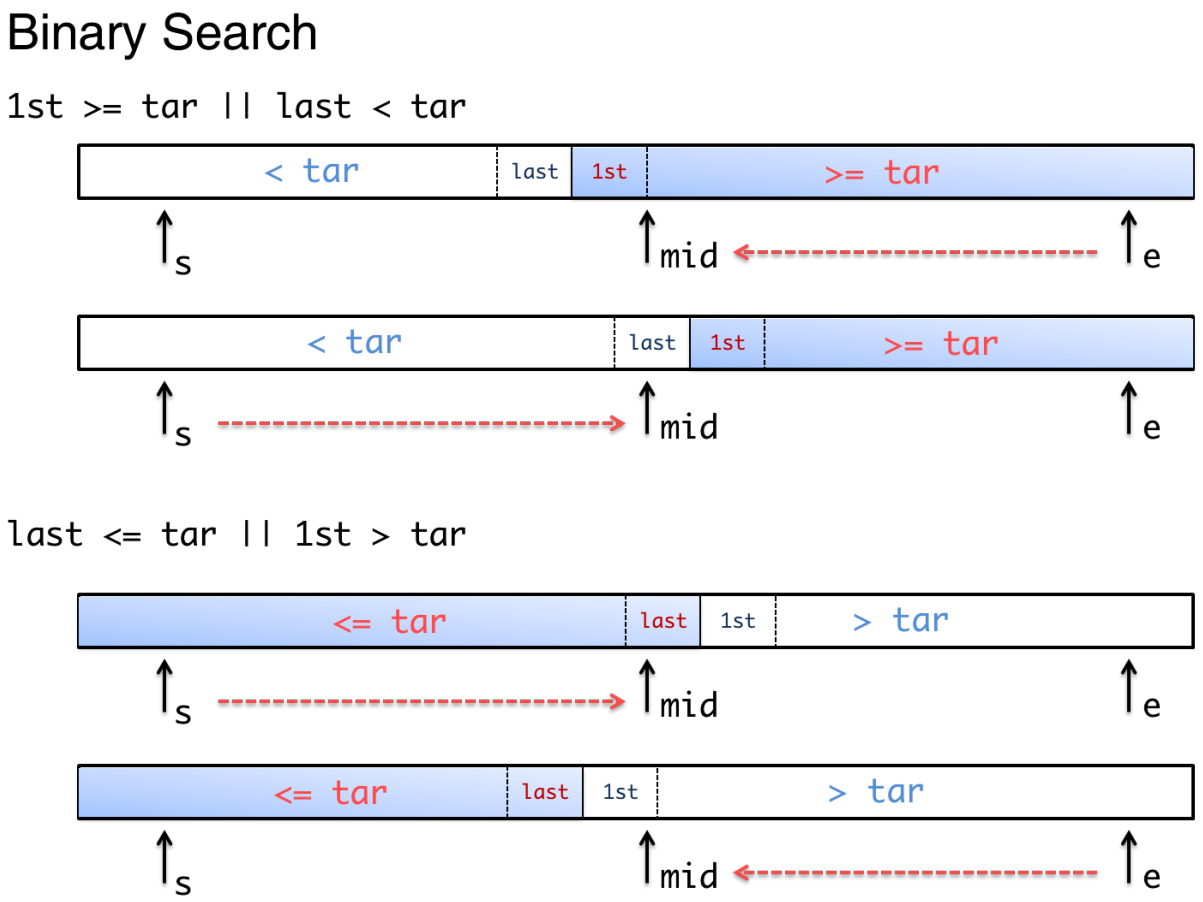
```cpp
#include <iostream>
#include <vector>
#include <limits.h>

using namespace std;

class Interval
{
public:
    Interval() : start(-1), end(-1) {}
    Interval(int x1, int x2) : start(x1), end(x2) {}
    int start, end;
};

class ListNode
{
public:
    ListNode() : next(NULL) {}
    ListNode(Interval x1) : interval(x1), next(NULL) {}
    Interval interval;
    ListNode* next;
};

ListNode* build(const vector<Interval>& arr)
{
    ListNode *head = NULL, *tail = NULL;
    for (int i = 0; i < arr.size(); i++)
    {
        if (head == NULL)
            head = tail = new ListNode(arr[i]);
        else
        {
            tail->next = new ListNode(arr[i]);
            tail = tail->next;
        }
    }
    return head;
}

void print(ListNode* head)
{
    ListNode *p = head;
    while (p)
    {
        cout << "(" << p->interval.start
            << ", " << p->interval.end << ")" << endl;
        p = p->next;
    }
}

void remove(ListNode* head)
{
    ListNode *p = head;
    while (p)
    {
        ListNode *tmp = p->next;
        delete p;
        p = tmp;
    }
}
```

```
61.
62. ListNode* insert(ListNode* head, const Interval& x)
63. {
64.     ListNode* px = new ListNode(x);
65.     if (head == NULL)
66.         return px;
67.     ListNode dummy(Interval(INT_MIN, INT_MIN));
68.     dummy.next = head;
69.     ListNode *p = head, *pre = &dummy;
70.     // Find the first node whose start > x.start
71.     while (p && p->interval.start <= x.start)
72.     {
73.         pre = p;
74.         p = p->next;
75.     }
76.     // Insert x:
77.     // (1) really insert or
78.     // (2) expand the last node.end to the maximum of itself and x.end
79.     if (pre->interval.end < x.start)
80.     {
81.         pre->next = px;
82.         px->next = p;
83.         pre = px;
84.     }
85.     else
86.         pre->interval.end = max(pre->interval.end, x.end);
87.     // Loop until we meet the first node p whose start > pre's node.end
88.     // here, we denote p's previous node as last.
89.     ListNode *last = pre;
90.     while (p && pre->interval.end >= p->interval.start)
91.     {
92.         pre->interval.end = max(pre->interval.end, p->interval.end);
93.         last = p;
94.         p = p->next;
95.     }
96.     // we know from pre->next to last, these nodes should be deleted, so
97.     last->next = NULL;
98.     remove(pre->next);
99.     // connect pre to p
100.    pre->next = p;
101.    return dummy.next;
102. }
103.
104. int main(int argc, char const *argv[])
105. {
106.     vector<Interval> arr
107.     {
108.         Interval(1, 2), Interval(3, 5), Interval(6, 7),
109.         Interval(8, 10), Interval(12, 16), Interval(18, 20)
110.     };
111.     ListNode* head = build(arr);
112.     print(head);
113.     cout << endl;
114.     head = insert(head, Interval(-2, 8));
115.     print(head);
116.     return 0;
117. }
```

Binary Search

Note: Standard Templates

> 口诀：
> 一者大于等于TAR，中者大于等于TAR时尾前移；先看s后看e，都是大于等于。
> 后者小于等于TAR，中者小于等于TAR时头后移；先看e后看s，都是小于等于。
> 如果没有见到等于，找其对偶者加一减一处理。
>
> 记住：判断时的坐标（s还是e）和上面 `while` 循环中的坐标是镜像对称的。



1. `1st >= tar` : `<--- e` : 先看s，后看e
2. `last <= tar` : `s --->` : 先看e，后看s
3. `1st > tar` : `last <= tar, + 1` : `s --->` : 先看e，后看s
4. `last < tar` : `1st >= tar, - 1` : `<--- e` : 先看s，后看e

```cpp
// A: [1,2,2,2,3,4]
// 1st >= tar
int binarySearch1(vector<int>& A, int tar)
{
    int n = A.size();
    if (n == 0)
        return -1;
    int s = 0, e = n - 1, mid;
    while (s + 1 < e)
    {
        mid = s + (e - s) / 2;
        if (A[mid] >= tar)
            e = mid;
        else
            s = mid;
    }
    if (A[s] >= tar)
        return s;
    else if (A[e] >= tar)
        return e;
    else
        return n;
}

// last <= tar
int binarySearch2(vector<int>& A, int tar)
{
    int n = A.size();
    if (n == 0)
        return -1;
    int s = 0, e = n - 1, mid;
    while (s + 1 < e)
    {
        mid = s + (e - s) / 2;
        if (A[mid] <= tar)
            s = mid;
        else
            e = mid;
    }
    if (A[e] <= tar)
        return e;
    else if (A[s] <= tar)
        return s;
    else
        return -1;
}

// 1st > tar <==> last <= tar, + 1
int binarySearch3(vector<int>& A, int tar)
{
    int n = A.size();
    if (n == 0)
        return -1;
    int s = 0, e = n - 1, mid;
    while (s + 1 < e)
    {
        mid = s + (e - s) / 2;
        if (A[mid] <= tar)
            s = mid;
        else
```

```
            e = mid;
    }
    if (A[e] <= tar)
        return e + 1;
    else if (A[s] <= tar)
        return s + 1;
    else
        return 0;
}

// last < tar <==> 1st >= tar, - 1
int binarySearch4(vector<int>& A, int tar)
{
    int n = A.size();
    if (n == 0)
        return -1;
    int s = 0, e = n - 1, mid;
    while (s + 1 < e)
    {
        mid = s + (e - s) / 2;
        if (A[mid] >= tar)
            e = mid;
        else
            s = mid;
    }
    if (A[s] >= tar)
        return s - 1;
    else if (A[e] >= tar)
        return e - 1;
    else
        return n - 1;
}
```

Subarray Sum II

Idea: Prefix Sum. Loop for each `P[i]` meaning first `i` elements' sum, find `P[i] - P[j] <= end` and `P[i] - P[j] >= start`. So it means find a `P[j]` satisfying `>= P[i] - end` and `<= P[i] - start`. So the problem becomes:

1. find 1st element `>= P[i] - end`, saying result is `x_id`.
2. find last element `<= P[i] - start`, saying result is `y_id`.
   Normally `y_id >= x_id`, and `(x_id, i - 1), ..., (y_id, i - 1)` are legal solutions. However, the corner cases should be carefully handled:
   In fact, the results have several cases:
3. `x_id > i - 1`, meaning that both `x_id` and `y_id` is larger than `i - 1`, so no solution.
4. `y_id == -1`, meaning that the range `P[i] - end ~ P[i] - start` smaller than `P[0]`, so the last one `<= P[i] - start` should be `-1` (no elements exists). But in this case, the first one `>= P[i] - end` is returned by `0`, meaning this range doesn't exist. So no solution too.
5. `y_id > i - 1`. in this case, in the potential results `(x_id, i - 1), ..., (y_id, i - 1)` only part of them are legal, i.e. `(x_id, i - 1), ..., (i - 1, i - 1)`, so the solution is `i - 1 - x_id + 1 = i - x_id`.
6. All the results `(x_id, i - 1), ..., (y_id, i - 1)` are legal, so solution is `y_id - x_id + 1`.

```cpp
int find_first(vector<int>& P, int end, int tar)
{// find 1st element >= tar
    int s = 0, e = end, mid;
    while (s + 1 < e)
    {
        mid = s + (e - s) / 2;
        if (P[mid] >= tar)
            e = mid;
        else
            s = mid;
    }
    if (P[s] >= tar)
        return s;
    else if (P[e] >= tar)
        return e;
    else
        return end + 1;
}
int find_last(vector<int>& P, int end, int tar)
{// find last element <= tar
    int s = 0, e = end, mid;
    while (s + 1 < e)
    {
        mid = s + (e - s) / 2;
        if (P[mid] <= tar)
            s = mid;
        else
            e = mid;
    }
    if (P[e] <= tar)
        return e;
    else if (P[s] <= tar)
        return s;
    else
        return -1;
}

int subarraySumII(vector<int>& A, int start, int end)
{
    int n = A.size();
    if (n == 0)
        return 0;
    vector<int> P(n + 1, 0);
    for (int i = 1; i <= n; i++)
        P[i] = P[i - 1] + A[i - 1];
    int count = 0;
    for (int i = 1; i <= n; i++)
    {
        int x = P[i] - end;
        int y = P[i] - start;
        int x_id = find_first(P, i, x);
        int y_id = find_last(P, i, y);
        if (x_id > i - 1 || y_id == -1)
            continue;
        if (y_id > i - 1)
            count += i - 1 - x_id + 1;
        else
            count += y_id - x_id + 1;
    }
    return count;
```

```
}
```

Search for a Range

Idea: find `1st >= tar` and `last <= tar`.

```
int mylowerbound(vector<int>& A, int tar)
{
    int n = A.size();
    if (n == 0)
        return -1;
    int s = 0, e = n - 1, mid;
    while (s + 1 < e)
    {
        mid = s + (e - s) / 2;
        if (A[mid] >= tar)
            e = mid;
        else
            s = mid;
    }
    return A[s] == tar ? s : A[e] == tar ? e : -1;
}
int myupperbound(vector<int>& A, int tar)
{
    int n = A.size();
    if (n == 0)
        return -1;
    int s = 0, e = n - 1, mid;
    while (s + 1 < e)
    {
        mid = s + (e - s) / 2;
        if (A[mid] <= tar)
            s = mid;
        else
            e = mid;
    }
    return A[e] == tar ? e + 1 : A[s] == tar ? s + 1 : 0;
}
vector<int> searchRange(vector<int>& nums, int target)
{
    vector<int> rst(2, -1);
    rst[0] = mylowerbound(nums, target);
    if (rst[0] == -1)
        return rst;
    rst[1] = myupperbound(nums, target) - 1;
    return rst;
}
```

If we use STL code, then

```cpp
vector<int> searchRange(vector<int>& nums, int target)
{
    vector<int> rst(2, -1);
    auto it1 = lower_bound(nums.begin(), nums.end(), target);
    if (it1 == nums.end() || *it1 != target)
        return rst;
    rst[0] = it1 - nums.begin();
    rst[1] = upper_bound(nums.begin(), nums.end(), target) - nums.begin() - 1;
    return rst;
}
```

Search in Rotated Sorted Array (Classic!!!)

Idea:

1. Find the minimal value's index, called `pivot`.
2. Use this `pivot` to binary search the array.

```cpp
int find_pivot(vector<int>& A)
{
    int n = A.size();
    if (n == 0 || n == 1 || n == 2)
        return n == 0 ? -1 : n == 1 ? 0 : A[0] <= A[1] ? 0 : 1;
    int s = 0, e = n - 1, mid;
    while (s + 1 < e)
    {
        mid = s + (e - s) / 2;
        if (A[mid] > A[e])
            s = mid;
        else if (A[mid] < A[s])
            e = mid;
        else
            break;
    }
    if (s + 1 == e)
        return e;
    else
        return 0;
}
int search(vector<int>& nums, int target)
{
    int n = nums.size();
    if (n == 0)
        return false;
    int pivot = find_pivot(nums);
    int s = 0, e = n - 1, mid;
    int ss, ee, midmid;
    while (s + 1 < e)
    {
        mid = s + (e - s) / 2;
        midmid = (mid + pivot) % n;
        ss = (s + pivot) % n;
        ee = (e + pivot) % n;
        if (nums[midmid] >= target)
            e = mid;
        else
            s = mid;
    }
    ss = (s + pivot) % n;
    ee = (e + pivot) % n;
    return nums[ss] == target ? ss : nums[ee] == target ? ee : -1;
}
```

Find Minimum in Rotated Sorted Array

Idea: binary search.

1. For corner cases: `n == 0 || n == 1 || n == 2`: return `INT_MIN` or `nums[0]` or `min(nums[0], nums[1])` respectively.
2. Otherwise: in binary search, there are only three cases:
    1. `A[mid] > A[e] => s = mid`
    2. `A[mid] < A[s] => e = mid`
    3. otherwise: `break`

After loop ends, check `s + 1 == e` ? if yes, it means it comes from 2.1 and 2.2 cases, and we know that `A[e]` must be the minimal value; if no, it means it comes from 2.3 case, and we know that the array is not rotated(shift equals to 0), then return `A[0]`.

```
int findMin(vector<int>& A)
{
    int n = A.size();
    if (n == 0 || n == 1 || n == 2)
        return n == 0 ? INT_MIN : n == 1 ? A[0] : min(A[0], A[1]);
    int s = 0, e = n - 1, mid;
    while (s + 1 < e)
    {
        mid = s + ((e - s) >> 1);
        if (A[mid] > A[e])
            s = mid;
        else if (A[mid] < A[s])
            e = mid;
        else
            break;
    }
    if (s + 1 == e)
        return A[e];
    else
        return A[0];
}
```

Find Minimum in Rotated Sorted Array II

Idea: Similar to Find Minimum in Rotated Sorted Array, but after the while loop stops, there are more cases to check.
After loop ends, check `s + 1 == e` ? if yes, it means it comes from 2.1 and 2.2 cases, and we know that `A[e]` must be the minimal value; if no, it means it possibly meet the duplicated cases.
So: we do two directions `O(n)` scanning, find the minimum. if not found, it means all values are equal (return `A[0]`)

```cpp
int findMin(vector<int>& nums)
{
    int n = nums.size();
    if (n == 0 || n == 1 || n == 2)
        return n == 0 ? INT_MIN : n == 1 ? A[0] : min(A[0], A[1]);
    int s = 0, e = n - 1, mid;
    while (s + 1 < e)
    {
        mid = s + (e - s) / 2;
        if (nums[mid] > nums[e])
            s = mid;
        else if (nums[mid] < nums[s])
            e = mid;
        else
            break;
    }
    if (s + 1 == e)
        return nums[e];
    else
    {
        int i;
        for (i = mid; i < n - 1 && nums[i] <= nums[i + 1]; i++);
        if (i != n - 1)
            return nums[i + 1];
        for (i = mid; i >= 0 && nums[i] <= nums[i + 1]; i--);
        if (i != -1)
            return nums[i + 1];
        return nums[0];
    }
}
```

Find the Duplicate Number

Idea: Try `i = [1,...,n]`, if `EqualLargerThan(i, nums) + i > n+1`, it means result is `>= i`. Otherwise, result `< i`.

Find Peak Element

Idea: Binary search.

```cpp
int findPeakElement(vector<int>& nums)
{
    int n = nums.size();
    if (n == 0 || n == 1)
        return n == 0 ? -1 : 0;
    int s = 0, e = n - 1, mid;
    while (s + 1 < e)
    {
        mid = s + (e - s) / 2;
        if (nums[mid] >= nums[mid + 1])
            e = mid;
        else
            s = mid;
    }
    if (nums[s] >= nums[s + 1])
        return s;
    else
        return e;
}
```

Find Peak Element II

Idea: Starting from a point, select one of its 4-neighbors larger than it, go to this point until get the peak. This original idea is `O(n^2)`. Now, for one row, find its maximal element, and see its up and down two elements. Go to the larger one and drop the other half. This is the basic idea of binary search. Row-wise and column-wise to loop the process above, then must find the optimal solution.

```cpp
int find_row_max(vector<vector<int>>& A, int r, int cs, int ce)
{
    int maxIndex, maxval = INT_MIN;
    for (int i = cs; i <= ce; i++)
    {
        if (A[r][i] > maxval)
        {
            maxIndex = i;
            maxval = A[r][i];
        }
    }
    return maxIndex;
}
int find_col_max(vector<vector<int>>& A, int c, int rs, int re)
{
    int maxIndex, maxval = INT_MIN;
    for (int i = rs; i <= re; i++)
    {
        if (A[i][c] > maxval)
        {
            maxIndex = i;
            maxval = A[i][c];
        }
    }
    return maxIndex;
}
vector<int> findPeakII(vector<vector<int> > A)
{
    int m = A.size();
    int n = m ? A[0].size() : m;
    vector<int> rst;
    if (m * n == 0)
        return rst;
    int rs = 0, re = m - 1, rm;
    int cs = 0, ce = n - 1 ,cm;
    bool flag = true;
    while (rs + 1 < re || cs + 1 < ce)
    {
        if (flag == true)
        {
            if (rs + 1 < re)
            {
                rm = rs + (re - rs) / 2;
                int rmaxIndex = find_row_max(A, rm, cs, ce);
                int crt = A[rm][rmaxIndex], upper = A[rm - 1][rmaxIndex],
                    down = A[rm + 1][rmaxIndex];
                if (crt > upper && crt > down)
                {
                    rst.push_back(rm);
                    rst.push_back(rmaxIndex);
                    return rst;
                }
                else if (crt > upper)
                {
                    rs = rm;
                }
                else
                {
                    re = rm;
                }
```

```
            }
            flag = false;
        }
        else
        {
            if (cs + 1 < ce)
            {
                cm = cs + (ce - cs) / 2;
                int cmaxIndex = find_col_max(A, cm, rs, re);
                int crt = A[cmaxIndex][cm], left = A[cmaxIndex][cm - 1],
                    right = A[cmaxIndex][cm + 1];
                if (crt > left && crt > right)
                {
                    rst.push_back(cmaxIndex);
                    rst.push_back(cm);
                    return rst;
                }
                else if (crt > left)
                {
                    cs = cm;
                }
                else
                {
                    ce = cm;
                }
            }
            flag = true;
        }
    }
}
```

H-Index II (Classic!!!)

Sqrt(x)

Idea: find the last one `z` such that `z*z <= x`.

Search a 2D Matrix

Idea:

1. Find the first one `> target` for the first column. (`upper_bound`)
2. Find target exactly at the row. (`find`)

```cpp
// STL solution:
bool searchMatrix(vector<vector<int>>& matrix, int target)
{
    int m = matrix.size();
    int n = m ? matrix[0].size() : m;
    if (m * n == 0)
        return false;
    vector<int> x(m);
    for (int i = 0; i < m; i++)
        x[i] = matrix[i][0];
    auto it = upper_bound(x.begin(), x.end(), target);
    int index = it - x.begin() - 1;
    if (index == -1)
        return false;
    auto it2 = find(matrix[index].begin(), matrix[index].end(), target);
    return it2 != matrix[index].end();
}
```

Binary Tree

1. Full binary tree: each node has 0 or 2 children.
2. Perfect binary tree: each internal node has 2 children. All leaf node same depth.
3. Complete binary tree: except possibly the last, each level is completed filled.

Binary Tree Paths

Idea: DFS. **Leaf Node Stop Condition**.

```cpp
void dfs(TreeNode* root, string crtrst, vector<string>& finalrst)
{
    string temp = to_string(root->val);
    if (root->left == NULL && root->right == NULL)
    {
        finalrst.push_back(crtrst + temp);
        return;
    }
    if (root->left)
        dfs(root->left, crtrst + temp + "->", finalrst);
    if (root->right)
        dfs(root->right, crtrst + temp + "->", finalrst);
}
vector<string> binaryTreePaths(TreeNode* root)
{
    vector<string> finalrst;
    string crtrst;
    if (root == NULL)
        return finalrst;
    dfs(root, crtrst, finalrst);
    return finalrst;
}
```

Lowest Common Ancestor

Idea: Find the path, and compare.

```cpp
bool getPath(TreeNode* root, TreeNode* x, vector<TreeNode*>& path)
{
    if (root == NULL)
        return false;
    path.push_back(root);
    if (root == x)
        return true;
    bool b1 = getPath(root->left, x, path);
    if (b1)
        return true;
    bool b2 = getPath(root->right, x, path);
    if (b2)
        return true;
    path.pop_back();
    return false;
}
TreeNode *lowestCommonAncestor(TreeNode *root, TreeNode *A, TreeNode *B)
{
    if (A == NULL && B == NULL)
        return NULL;
    if (A == NULL || B == NULL)
        return A == NULL ? B : A;
    vector<TreeNode*> PathA, PathB;
    getPath(root, A, PathA);
    getPath(root, B, PathB);
    int i = 0;
    int l1 = PathA.size(), l2 = PathB.size();
    for (; i < min(l1, l2) && PathA[i] == PathB[i]; i++);
    return PathA[i - 1];
}
```

A more efficient solution is DFS.

```cpp
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q)
{
    if (root == NULL)
        return NULL;
    if (root == p || root == q)
        return root;
    TreeNode *left = lowestCommonAncestor(root->left, p, q);
    TreeNode *right = lowestCommonAncestor(root->right, p, q);
    return left && right ? root : left ? left : right ? right : NULL;
}
```

Inorder Successor in Binary Search Tree

Idea: 2 cases:

- Case 1: if `p` has right child, find the leftmost child of `p->right`.
- Case 2: otherwise, traverse from `root`, in the path to `p`, the last node turning left is the answer.

```
TreeNode* inorderSuccessor(TreeNode* root, TreeNode* p)
{
    if (root == NULL)
        return NULL;
    TreeNode* crt;
    if (p->right)
    {
        crt = p->right;
        while (crt->left)
            crt = crt->left;
        return crt;
    }
    else
    {
        crt = root;
        TreeNode *pre = NULL;
        while (crt != p)
        {
            if (p->val < crt->val)
            {
                pre = crt;
                crt = crt->left;
            }
            else
            {
                crt = crt->right;
            }
        }
        return pre;
    }
}
```

Binary Search Tree Iterator

Idea: using iterative version of in-order traversal.

```
class BSTIterator
{
public:
    //@param root: The root of binary tree.
    BSTIterator(TreeNode *root)
    {
        p = root;
    }
    //@return: True if there has next node, or false
    bool hasNext()
    {
        return p || !S.empty();
    }
    //@return: return next node
    TreeNode* next()
    {
        if (!hasNext())
            return NULL;
        while (p)
        {
            S.push(p);
            p = p->left;
        }
        TreeNode* rst = S.top();
        S.pop();
        p = rst->right;
        return rst;
    }
    stack<TreeNode*> S;
    TreeNode* p;
};
```

Binary Tree Preorder Traversal

Idea: **Standard Template**

```
vector<int> preorderTraversal(TreeNode* root)
{
    vector<int> rst;
    if (root == NULL)
        return rst;
    stack<TreeNode*> S;
    S.push(root);
    while (!S.empty())
    {
        TreeNode* p = S.top();
        S.pop();
        rst.push_back(p->val);
        if (p->right)
            S.push(p->right);
        if (p->left)
            S.push(p->left);
    }
    return rst;
}
```

Binary Tree Inorder Traversal

Idea: **Standard Template!!!**

1. If `crt` not `NULL`, always push the left child until `crt` becomes `NULL`.
2. Visit and pop the top of the stack.
3. Set `crt` as the stack's top's right child.

`while` condition: `crt != NULL || !S.empty()`.

```
vector<int> inorderTraversal(TreeNode* root)
{
    vector<int> rst;
    if (root == NULL)
        return rst;
    stack<TreeNode*> S;
    TreeNode* crt = root;
    while (crt != NULL || !S.empty())
    {
        // 1. Push Step:
        while (crt)
        {
            S.push(crt);
            crt = crt->left;
        }
        // 2. Pop Step:
        TreeNode* tmp = S.top();
        S.pop();
        crt = tmp->right;
        // 3. Print Step:
        rst.push_back(tmp->val);
    }
    return rst;
}
```

Kth Smallest Element in a BST

Idea: Iterative Inorder Traversal.

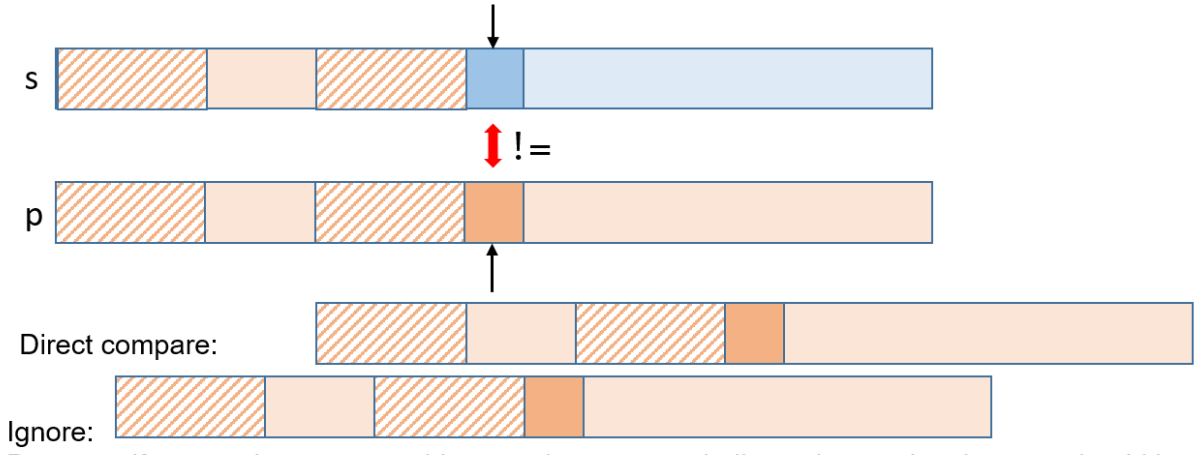
```cpp
int kthSmallest(TreeNode* root, int k)
{
    if (k <= 0 || root == NULL)
        return INT_MIN;
    TreeNode *p = root;
    stack<TreeNode*> S;
    int count = 0;
    while (p || !S.empty())
    {
        while (p)
        {
            S.push(p);
            p = p->left;
        }
        TreeNode* x = S.top();
        S.pop();
        count++;
        if (count == k)
            return x->val;
        p = x->right;
    }
    return INT_MIN;
}
```

Search Range in Binary Search Tree

Idea: Also use iterative inorder traversal.
Modification: when pop the stack, check its value: if `> k2`, then `break`; if `< k1`, then drop. Also, when pushing the left child, if current node's parent `< k1`, then `break`.

Binary Tree Postorder Traversal

Idea 1: Two stacks, one stack using quasi-preorder solution, push the result into the other stack. Finally print 2nd stack.
Idea 2: **Standard Template!!!** See Explanation Here.



Because, if we need to compare this state, it means we believe, the overlapping part should be equal, which is ensured impossible. (otherwise, the prefix-suffix maximum length will be larger than current value.)

```cpp
vector<int> postorderTraversal(TreeNode *root)
{
    vector<int> rst;
    if (root == NULL)
        return rst;
    stack<TreeNode*> S;
    TreeNode *p = root;
    while (p || !S.empty())
    {
        // 1. Push Step:
        while (p)
        {
            if (p->right)
                S.push(p->right);
            S.push(p);
            p = p->left;
        }
        // 2. Pop Step:
        TreeNode* tmp = S.top();
        S.pop();
        // Two cases:
        // (1) tmp's right has not been processed.
        // Note here we must check S.empty(),
        // because for the last node, i.e. the root,
        // when it has been poped out, S will be empty.
        if (!S.empty() && tmp->right == S.top())
        {
            S.pop();
            S.push(tmp);
            p = tmp->right;
        }
        else // (2) tmp's right does not exist, or has not been processed.
        {
            rst.push_back(tmp->val); // do sth. here.
            p = NULL;
        }
    }
    return rst;
}
```

Binary Tree Zigzag Level Order Traversal

Idea: Two stacks to represent a queue. One stack stores the `crtlevel` nodes, the other one stores the `nextlevel` nodes. If `crtlevel` stack is empty, swap the two stacks. Of course, use a `bool normalOrder` to control which child should be push into the `nextlevel` stack first.

```
vector<vector<int>> zigzagLevelOrder(TreeNode* root)
{
    vector<vector<int>> rst;
    if (root == NULL)
        return rst;
    vector<stack<TreeNode*>> S(2);
    int level = 0;
    S[0].push(root);
    vector<int> temprst;
    while (!S[level % 2].empty())
    {
        TreeNode *tmp = S[level % 2].top();
        temprst.push_back(tmp->val);
        S[level % 2].pop();
        TreeNode *left(tmp->left), *right(tmp->right);
        if (level % 2 == 1)
            swap(left, right);
        if (left)
            S[(level + 1) % 2].push(left);
        if (right)
            S[(level + 1) % 2].push(right);
        if (S[level % 2].empty())
        {
            level++;
            rst.push_back(temprst);
            temprst.clear();
        }
    }
    return rst;
}
```

Convert Sorted Array to Binary Search Tree With Minimal Height

Idea: Divide and Conquer.

```
TreeNode *helper(vector<int>& A, int s, int e)
{
    if (s > e)
        return NULL;
    int mid = s + (e - s) / 2;
    TreeNode *root = new TreeNode(A[mid]);
    root->left = helper(A, s, mid - 1);
    root->right = helper(A, mid + 1, e);
    return root;
}
TreeNode *sortedArrayToBST(vector<int> &A)
{
    int n = A.size();
    return helper(A, 0, n - 1);
}
```

Segment Tree Build / Segment Tree Build II

Idea: The classic template.

```cpp
// build I
SegmentTreeNode * build(int start, int end)
{
    if (start > end)
        return NULL;
    SegmentTreeNode* root = new SegmentTreeNode(start, end);
    if (start != end)
    {
        int mid = start + (end - start) / 2;
        root->left = build(start, mid);
        root->right = build(mid + 1, end);
    }
    return root;
}
// build II
SegmentTreeNode* helper(vector<int>& A, int s, int e)
{
    if (s > e)
        return NULL;
    SegmentTreeNode* root = new SegmentTreeNode(s, e);
    if (s != e)
    {
        int mid = s + (e - s) / 2;
        root->left = helper(A, s, mid);
        root->right = helper(A, mid + 1, e);
        root->max = max(root->left->max, root->right->max);
    }
    else
        root->max = A[s];
    return root;
}
SegmentTreeNode * build(vector<int>& A)
{
    int n = A.size();
    if (n == 0)
        return NULL;
    return helper(A, 0, n - 1);
}
```

Segment Tree Query / Segment Tree Query II

Idea:

1. `foo(root, start, end) = max(foo(root->left, start, end), foo(root->right, start, end))`
2. `foo(root, start, end) = foo(root->left, start, end) + foo(root->right, start, end)`

```
// query I
int query(SegmentTreeNode *root, int start, int end)
{
    if (root == NULL)
        return INT_MIN;
    int s = root->start, e = root->end;
    if (start > e || end < s)
        return INT_MIN;
    if (s >= start && e <= end)
        return root->max;
    return max(query(root->left, start, end), query(root->right, start, end));
}
// query II
int query(SegmentTreeNode *root, int start, int end)
{
    if (root == NULL || start > end)
        return 0;
    int s = root->start, e = root->end;
    if (start > e || end < s)
        return 0;
    if (start <= s && end >= e)
        return root->count;
    return query(root->left, start, end) + query(root->right, start, end);
}
```

Segment Tree Modify

Idea: check `index` is in left or right, or root itself. Modify the corresponding branch. At last update the maximum value.

```
void modify(SegmentTreeNode *root, int index, int value)
{
    if (root == NULL)
        return;
    int s = root->start, e = root->end;
    if (index > e || index < s)
        return;
    if (s == e)
    {
        root->max = value;
        return;
    }
    int mid = s + (e - s) / 2;
    if (index <= mid)
        modify(root->left, index, value);
    else
        modify(root->right, index, value);
    root->max = max(root->left->max, root->right->max);
}
```

Count of Smaller Number before itself

Idea: Dynamically updating the segment tree. At first no element added. With looping over each element in `A`, count the number of elements smaller than `A[i]` and make the number of `A[i]` add 1.

```
class SegNode
{
public:
    SegNode(int s, int e) : start(s), end(e),
        count(0), left(NULL), right(NULL) {}
    SegNode() : start(-1), end(-1),
        count(0), left(NULL), right(NULL) {}
    int start, end;
    int count;
    SegNode *left, *right;
};
SegNode* construct(int start, int end)
{
    if (start > end)
        return NULL;
    SegNode* root = new SegNode(start, end);
    if (start == end)
        return root;
    else
    {
        int mid = start + (end - start) / 2;
        root->left = construct(start, mid);
        root->right = construct(mid + 1, end);
        return root;
    }
}
int query(SegNode* root, int minval, int maxval)
{
    if (root == NULL || minval > maxval)
        return 0;
    int s = root->start, e = root->end;
    if (minval > e || maxval < s)
        return 0;
    if (minval <= s && maxval >= e)
        return root->count;
    return query(root->left, minval, maxval)
        + query(root->right, minval, maxval);
}
void modify(SegNode* root, int value)
{
    if (root == NULL)
        return;
    int s = root->start, e = root->end;
    if (value > e || value < s)
        return;
    if (s == e)
        root->count++;
    else
    {
        modify(root->left, value);
        modify(root->right, value);
        root->count = root->left->count + root->right->count;
    }
}
vector<int> countOfSmallerNumberII(vector<int> &A)
{
    int n = A.size();
    vector<int> rst(n, 0);
    if (n <= 1)
        return rst;
```

```
61.     auto it = minmax_element(A.begin(), A.end());
62.     int minval = *it.first;
63.     int maxval = *it.second;
64.     SegNode* root = construct(minval, maxval);
65.     for (int i = 0; i < n; i++)
66.     {
67.         rst[i] = query(root, minval, A[i] - 1);
68.         modify(root, A[i]);
69.     }
70.     return rst;
71. }
```

Interval Minimum Number

Idea: Classic Segment Tree Problem, use `construct` and `query`.

```cpp
class SegNode
{
public:
    SegNode(int s, int e) : start(s), end(e), left(NULL), right(NULL) {}
    SegNode() : start(-1), end(-1), left(NULL), right(NULL) {}
    int start, end;
    int minval;
    SegNode *left, *right;
};
SegNode* construct(vector<int>& A, int s, int e)
{
    if (s > e)
        return NULL;
    SegNode* root = new SegNode(s, e);
    if (s == e)
        root->minval = A[s];
    else
    {
        int mid = s + (e - s) / 2;
        root->left = construct(A, s, mid);
        root->right = construct(A, mid + 1, e);
        root->minval = min(root->left->minval, root->right->minval);
    }
    return root;
}
int query(SegNode* root, int start, int end)
{
    if (root == NULL || start > end)
        return INT_MAX;
    int s = root->start, e = root->end;
    if (start > e || end < s)
        return INT_MAX;
    if (start <= s && end >= e)
        return root->minval;
    return min(query(root->left, start, end), query(root->right, start, end));
}
vector<int> intervalMinNumber(vector<int> &A, vector<Interval> &queries)
{
    int n = A.size();
    int nq = queries.size();
    vector<int> rst(nq, INT_MAX);
    if (n == 0)
        return rst;
    if (queries.size() == 0)
        return rst;
    SegNode* root = construct(A, 0, n - 1);
    for (int i = 0; i < queries.size(); i++)
        rst[i] = query(root, queries[i].start, queries[i].end);
    return rst;
}
```

Interval Sum / Interval Sum II

Idea: Classic segment tree problem.

```
class SegNode
{
public:
    SegNode(int s, int e) : start(s), end(e), left(NULL), right(NULL), sum(0) {}
    SegNode() : start(-1), end(-1), left(NULL), right(NULL), sum(0) {}
    int start, end;
    long long sum;
    SegNode *left, *right;
};
SegNode* build(vector<int>& A, int s, int e)
{
    if (s > e)
        return NULL;
    SegNode* root = new SegNode(s, e);
    if (s == e)
        root->sum = A[s];
    else
    {
        int mid = s + (e - s) / 2;
        root->left = build(A, s, mid);
        root->right = build(A, mid + 1, e);
        root->sum = root->left->sum + root->right->sum;
    }
    return root;
}
long long query(SegNode* root, int s, int e)
{
    if (root == NULL || s > e)
        return 0;
    int start = root->start, end = root->end;
    if (s > end || e < start)
        return 0;
    if (s <= start && e >= end)
        return root->sum;
    return query(root->left, s, e) + query(root->right, s, e);
}
vector<long long> intervalSum(vector<int> &A, vector<Interval> &queries)
{
    int n = A.size();
    int m = queries.size();
    vector<long long> rst(m, 0);
    if (n == 0)
        return rst;
    SegNode* root = build(A, 0, n - 1);
    for (int i = 0; i < m; i++)
    {
        rst[i] = query(root, queries[i].start, queries[i].end);
    }
    return rst;
}
```

Range Sum Query - Mutable

Idea: Segment Tree.

```
class NumArray
{
public:
    class SegTreeNode
    {
    public:
        SegTreeNode() : start(-1), end(-1), sum(0), left(NULL), right(NULL) {}
        SegTreeNode(int x1, int x2) : start(x1), end(x2), sum(0),
            left(NULL), right(NULL) {}
        int start, end;
        int sum;
        SegTreeNode *left, *right;
    };
    SegTreeNode* buildTree(vector<int>& A, int s, int e)
    {
        if (s > e)
            return NULL;
        SegTreeNode *root = new SegTreeNode(s, e);
        if (s == e)
        {
            root->sum = A[s];
            return root;
        }
        int mid = s + (e - s) / 2;
        root->left = buildTree(A, s, mid);
        root->right = buildTree(A, mid + 1, e);
        root->sum = root->left->sum + root->right->sum;
        return root;
    }
    NumArray(vector<int> &nums)
    {
        int n = nums.size();
        root = buildTree(nums, 0, n - 1);
    }
    void modify(SegTreeNode* root, int index, int val)
    {
        if (root == NULL)
            return;
        int s = root->start, e = root->end;
        if (index < s || index > e)
            return;
        if (s == e)
        {
            root->sum = val;
            return;
        }
        modify(root->left, index, val);
        modify(root->right, index, val);
        root->sum = root->left->sum + root->right->sum;
    }
    void update(int i, int val)
    {
        modify(root, i, val);
    }
    int query(SegTreeNode* root, int s, int e)
    {
        if (root == NULL || s > e)
            return 0;
        int start = root->start, end = root->end;
        if (e < start || s > end)
```

```
            return 0;
        if (e >= end && s <= start)
            return root->sum;
        return query(root->left, s, e) + query(root->right, s, e);
    }
    int sumRange(int i, int j)
    {
        return query(root, i, j);
    }
    SegTreeNode *root;
};
// Your NumArray object will be instantiated and called as such:
// NumArray numArray(nums);
// numArray.sumRange(0, 1);
// numArray.update(1, 10);
// numArray.sumRange(1, 2);
```

Count Complete Tree Nodes

Idea: Check whether current tree is perfect, if perfect, return `2^h - 1`; else return `1 + foo(root->left) + foo(root->right)`.

```
int countNodes(TreeNode* root)
{
    if (root == NULL)
        return 0;
    TreeNode *left = root, *right = root;
    int h = 0;
    while (left && right)
    {
        left = left->left;
        right = right->right;
        h++;
    }
    if (left == NULL && right == NULL)
        return (1 << h) - 1;
    else
        return 1 + countNodes(root->left) + countNodes(root->right);
}
```

Dynamic Programming

Stone Game

Idea: Interval DP. Define `dp[i][j]` is the minimal score obtained from `A[i]` to `A[j]`. Then there is `dp[i][j] = min{dp[i][k] + dp[k + 1][j]} + sum(i,...,j)`, where `sum(i,...,j)` can be easily calculated using `prefixSum`.

```cpp
int stoneGame(vector<int>& A)
{
    int n = A.size();
    if (n <= 1)
        return 0;
    if (n == 2)
        return A[0] + A[1];
    vector<int> prefixSum(n + 1, 0);
    for (int i = 1; i <= n; i++)
        prefixSum[i] = prefixSum[i - 1] + A[i - 1];
    // Sij = prefixSum[j + 1] - prefixSum[i];
    vector<vector<int>> dp(n, vector<int>(n, 0));
    for (int i = 0; i < n - 1; i++)
        dp[i][i + 1] = A[i] + A[i + 1];
    int i = 0, j = 2, k = 3;
    while (j < n)
    {
        int minval = INT_MAX;
        for (int m = i; m < j; m++)
            minval = min(minval, dp[i][m] + dp[m + 1][j]);
        minval += prefixSum[j + 1] - prefixSum[i];
        dp[i][j] = minval;
        i++;
        j++;
        if (j == n)
        {
            i = 0;
            j = k;
            k++;
        }
    }
    return dp[0][n - 1];
}
```

Jump Game / Jump Game II

Idea: `O(n^2)`. Note that in the following Jump Game II, `j + A[j] >= i` should have been `dp[j] != INT_MAX && j + A[j] >= i`, but this is not necessary because here we just need to find the first `j` satisfying the condition, which will never exist after a `j` with `dp[j] == INT_MAX`.

```cpp
// jump game I
bool canJump(vector<int>& nums)
{
    int n = nums.size();
    if (n == 0)
        return true;
    vector<int> dp(n, 0);
    dp[0] = 1;
    for (int i = 1; i < n; i++)
    {
        for (int j = i - 1; j >= 0; j--)
        {
            if (dp[j] == 0)
                return false;
            else if (i - j <= nums[j])
            {
                dp[i] = 1;
                break;
            }
        }
        if (dp[i] == 0)
            return false;
    }
    return dp[n - 1];
}
// jump game II
int jump(vector<int> A)
{
    int n = A.size();
    if (n == 0)
        return 0;
    vector<int> dp(n, 0);
    for (int i = 1; i < n; i++)
    {
        dp[i] = INT_MAX;
        for (int j = 0; j < i; j++)
        {
            if (j + A[j] >= i) // <==
            {
                dp[i] = dp[j] + 1;
                break;
            }
        }
    }
    return dp[n - 1];
}
```

Climbing Stairs

Idea: Coordinate DP. `f(n) = f(n - 1) + f(n - 2)`, so we need to initialize `f(0)` and `f(1)` only.

```cpp
int climbStairs(int n)
{
    // f(n) = f(n - 1) + f(n - 2)
    vector<int> dp(3, 0);
    dp[0] = 1;
    dp[1] = 1;
    for (int i = 2; i <= n; i++)
        dp[i % 3] = dp[(i - 1) % 3] + dp[(i - 2) % 3];
    return dp[n % 3];
}
```

House Robber, House Robber II (Check Again!!!)

Idea: Coordinate DP. **Local and Global solution**. 局部全局解法，其中 `全局解` 为 `局部解` 和 `前一个全局解` 取 `极值` 的结果。 难处在于如何推局部解的递推式。

1. House Robber.
   Define `L[i]` as the max money could obtain up to `A[i]`, requiring `A[i]` must be robbed. `G[i]` as the global solution. So
   `L[i] = G[i - 2] + A[i]`, and `G[i] = max{L[i], G[i - 1]}`.
   Here `G[i] = max{L[i], G[i - 1]}` is a template for such kind of problem.

```cpp
int rob(vector<int>& nums)
{
    int n = nums.size();
    if (n == 0)
        return 0;
    if (n == 1)
        return nums[0];
    if (n == 2)
        return max(nums[0], nums[1]);
    vector<int> L(n, 0), G(n, 0);
    L[0] = nums[0];
    L[1] = nums[1];
    G[0] = L[0]; // G[i] = max(L[i], G[i - 1])
    G[1] = max(L[1], G[0]);
    for (int i = 2; i < n; i++)
    {
        L[i] = G[i - 2] + nums[i];
        G[i] = max(L[i], G[i - 1]);
    }
    return G[n - 1];
}
```

2. If all in a circle, then the optimal solution must lie in two cases:
   (1) `A[0]` must be selected, `A[n - 1]` must be not selected;
   (2) `A[n - 1]` must be selected, `A[0]` must be not selected.
   Case 1 is actually applying House Robber algorithm to `A[0, ..., n - 2]`, and Case 2 is actually to `A[1, ..., n - 1]`.

```cpp
int helper(vector<int>& A, int s, int e)
{
    int n = e - s + 1;
    vector<int> L(n, 0), G(n, 0);
    L[0] = G[0] = A[s];
    L[1] = A[s + 1];
    G[1] = max(L[1], G[0]);
    for (int i = 2; i < n; i++)
    {
        L[i] = G[i - 2] + A[s + i];
        G[i] = max(L[i], G[i - 1]);
    }
    return G[n - 1];
}
int rob(vector<int>& nums)
{
    int n = nums.size();
    if (n == 0)
        return 0;
    if (n == 1)
        return nums[0];
    if (n == 2)
        return max(nums[0], nums[1]);
    int x = helper(nums, 0, n - 2);
    int y = helper(nums, 1, n - 1);
    return max(x, y);
}
```

Minimum Path Sum, Unique Paths, Unique Paths II

Idea: Coordinate DP. Very Similar.
For Minimum Path Sum: Initialize `dp[0][*], dp[*][0]`, and function is `dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j]`.

```cpp
// Minimum Path Sum
int minPathSum(vector<vector<int> > &grid)
{
    int m = grid.size();
    int n = m ? grid[0].size() : m;
    if (m * n == 0)
        return 0;
    vector<vector<int>> dp(m, vector<int>(n, 0));
    dp[0][0] = grid[0][0];
    for (int i = 1; i < m; i++)
        dp[i][0] = dp[i - 1][0] + grid[i][0];
    for (int i = 1; i < n; i++)
        dp[0][i] = dp[0][i - 1] + grid[0][i];
    for (int i = 1; i < m; i++)
    {
        for (int j = 1; j < n; j++)
        {
            dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]) + grid[i][j];
        }
    }
    return dp[m - 1][n - 1];
}
// Unique Path I
int uniquePaths(int m, int n)
{
    if (m * n == 0)
        return 0;
    vector<vector<int>> dp(m, vector<int>(n, 1));
    for (int i = 1; i < m; i++)
    {
        for (int j = 1; j < n; j++)
        {
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
        }
    }
    return dp[m - 1][n - 1];
}
// Unique Path II: obstacle exists
int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid)
{
    int m = obstacleGrid.size();
    int n = m ? obstacleGrid[0].size() : m;
    if (m * n == 0)
        return 0;
    vector<vector<int>> dp(m, vector<int>(n, 0));
    dp[0][0] = obstacleGrid[0][0] == 0;
    if (dp[0][0] == 0)
        return 0;
    for (int i = 1; i < m; i++)
        dp[i][0] = obstacleGrid[i][0] == 0 ? dp[i - 1][0] : 0;
    for (int j = 1; j < n; j++)
        dp[0][j] = obstacleGrid[0][j] == 0 ? dp[0][j - 1] : 0;
    for (int i = 1; i < m; i++)
    {
        for (int j = 1; j < n; j++)
        {
            dp[i][j] = obstacleGrid[i][j] == 0 ? dp[i - 1][j] + dp[i][j - 1] : 0;
        }
    }
    return dp[m - 1][n - 1];
```

```
}
```

Unique Binary Search Trees, Unique Binary Search Trees II

Idea:

1. Problem I: Interval DP. 1D `=>` 2D. `dp[i,j] = sum(dp[i,r-1]*dp[r+1,j]), r = i,...,j`.
2. Problem II: Interval DP, memorization. Store the result for range `[s, e]` in a matrix: `vector<TreeNode*> A[n][n]`

```
// Problem I:
int numTrees(int n)
{
    vector<int> dp(n + 1, 0);
    dp[0] = 1;
    dp[1] = 1;
    for (int i = 2; i <= n; i++)
    {
        int sum = 0;
        for (int j = 0; j <= i - 1; j++)
            sum += dp[j] * dp[i - 1 - j];
        dp[i] = sum;
    }
    return dp[n];
}
// Problem II:
vector<TreeNode*> helper(int s, int e, vector<vector<vector<TreeNode*>>>& matrix)
{
    if (s > e)
        return vector<TreeNode*>(1, NULL);
    vector<TreeNode*>& temp = matrix[s - 1][e - 1];
    if (!temp.empty())
        return temp;
    for (int i = s; i <= e; i++)
    {
        vector<TreeNode*> lefts = helper(s, i - 1, matrix);
        vector<TreeNode*> rights = helper(i + 1, e, matrix);
        for (int i1 = 0; i1 < lefts.size(); i1++)
        {
            for (int i2 = 0; i2 < rights.size(); i2++)
            {
                TreeNode* root = new TreeNode(i);
                root->left = lefts[i1];
                root->right = rights[i2];
                temp.push_back(root);
            }
        }
    }
    return temp;
}
vector<TreeNode*> generateTrees(int n)
{
    if (n == 0)
        return vector<TreeNode*>();
    vector<vector<vector<TreeNode*>>> matrix(n, vector<vector<TreeNode*>>(n));
    return helper(1, n, matrix);
}
```

Decode Ways

Idea: check the current and the previous characters, consider the current character is `'0'`. There will be 4 cases to consider.
Define `dp[i]` as the result for first `i` characters, `x` as the number of combination of `s[i - 2], s[i - 1]`. so

1. `s[i - 1] == 0`
   we know that `s[i - 1]` cannot be decoded independently, so it must rely on the combination of `s[i - 2], s[i - 1]`.
   - 1.1: if `10 <= x <= 26`, meaning the combination is valid, so `dp[i] = dp[i - 2]`;
   - 1.2: otherwise, meaning the combination is non-valid, so `dp[i] = 0`;

2. `s[i - 1] != 0`
   we know that `s[i - 1]` is sured to be decoded independently, so we will see whether the combination of `s[i - 2], s[i - 1]` can also produce another solution.
   - 2.1: if `10 <= x <= 26`, meaning the combination is valid, so `dp[i] = dp[i - 2] + dp[i - 1]`;
   - 2.2: otherwise, meaning the combination is non-valid, so `dp[i] = dp[i - 1]`.

```
int numDecodings(string& s)
{
    int n = s.size();
    if (n == 0)
        return 0;
    vector<int> dp(n + 1, 0);
    dp[0] = 1; //
    dp[1] = s[0] != '0';
    for (int i = 2; i <= n; i++)
    {
        int x = (s[i - 2] - '0') * 10 + (s[i - 1] - '0');
        if (s[i - 1] == '0')
            dp[i] = (x <= 26 && x >= 10) ? dp[i - 2] : 0;
        else
            dp[i] = (x <= 26 && x >= 10) ? dp[i - 1] + dp[i - 2] : dp[i - 1];
    }
    return dp[n];
}
```

Perfect Squares

Idea: `dp[n] = min{dp[n-k*k]+1}, k = 1,2,...`

```cpp
int numSquares(int n)
{
    if (n == 0)
        return 1;
    vector<int> dp(n + 1, 0);
    for (int i = 1; i * i <= n; i++)
        dp[i * i] = 1;
    for (int i = 1; i <= n; i++)
    {
        if (dp[i] == 0)
        {
            int minval = INT_MAX;
            for (int j = 1; j * j < i; j++)
                minval = min(minval, dp[i - j * j]);
            dp[i] = minval + 1;
        }
    }
    return dp[n];
}
```

Longest Increasing Subsequence (`O(NlogN)` solution, Check again! Classic)

Idea:

1. Classic, DP solution, `O(n^2)`: `dp[i] = max{dp[j]}, j < i && nums[j] < nums[i]`, meaning the longest increasing subsequence ended at `i`. Finally, find the max value of all `dp[i]`.

```cpp
int longestIncreasingSubsequence(vector<int> nums)
{
    int n = nums.size();
    if (n == 0)
        return 0;
    vector<int> dp(n, 1);
    for (int i = 1; i < n; i++)
    {
        for (int j = 0; j < i; j++)
        {
            if (nums[j] <= nums[i])
                dp[i] = max(dp[i], dp[j] + 1);
        }
    }
    return *max_element(dp.begin(), dp.end());
}
```

2. Advanced, Binary Search Solution, `O(nlogn)`: Define `vector<int> tail`, and `tail[i]` records the tail value of LIS of length `i + 1`. So `tail` must be a non-decreasing sequence. So if met a new element `nums[i]`, it must follow two cases:
    1. If `nums[i]` is larger than the back of `tail`, then just push back `nums[i]` to tail;
    2. Find the `lower_bound` of `tail`, if `nums[i]` is smaller than this upper bound, replace it.

```cpp
int lengthOfLIS(vector<int>& nums)
{
    int n = nums.size();
    if (n == 0)
        return 0;
    vector<int> tail;
    tail.reserve(n);
    for (int i = 0; i < n; i++)
    {
        int x = nums[i];
        auto it = lower_bound(tail.begin(), tail.end(), x);
        if (it == tail.end())
            tail.push_back(x);
        else if (x < *it)
            *it = x;
    }
    return tail.size();
}
```

Longest Common Substring (Classic)

Idea:

1. Define `dp[i][j]` as the length of the longest common substring of `s[0,...,i-1]` and `t[0,...,j-1]`, ended at `s[i-1]` and `t[j-1]`.
2. So `if s[i-1] != t[j-1]`, then `dp[i][j] == 0`; otherwise, `dp[i][j] = dp[i-1][j-1] + 1`.
3. Finally, compute the maximum length of all `dp[i][j]`. Note: In this step, `i` and `j` must be started from `0` because we should consider `s` or `t` empty case. And the return value is the maximum length, not `dp[m][n]`!.

```cpp
int longestCommonSubstring(string &A, string &B)
{
    int m = A.size(), n = B.size();
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    int maxlen = 0;
    for (int i = 1; i <= m; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            dp[i][j] = A[i - 1] == B[j - 1] ? dp[i - 1][j - 1] + 1 : 0;
            maxlen = max(maxlen, dp[i][j]);
        }
    }
    return maxlen;
}
```

Longest Common Subsequence

Idea: **Local and Global solution**.

1. Define `L[i][j]` as the max length of the Longest Common Subsequence of `A`'s first `i` characters and `B`'s first `j` characters, where the subsequence ending with `A[i - 1]` and `B[j - 1]`;
2. Define `G[i][j]` as the same as `L[i][j]` except not requiring the subsequence ends with `A[i - 1]` and `B[j - 1]`.

So, according to **Local and Global Solution**, we can easily find that

- `L[i][j] = G[i - 1][j - 1] + 1` (if `A[i - 1] == B[j - 1]`) or `0` (otherwise);
- `G[i][j] = max{L[i][j], G[i - 1][j], G[i][j - 1]}` `<==== Note Here`

```
inline int max3(int a, int b, int c)
{
    return max(a, max(b, c));
}
int longestCommonSubsequence(string A, string B)
{
    int m = A.size(), n = B.size();
    typedef vector<vector<int>> Mat;
    Mat L(m + 1, vector<int>(n + 1, 0));
    Mat G(m + 1, vector<int>(n + 1, 0));
    for (int i = 1; i <= m; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            L[i][j] = A[i - 1] == B[j - 1] ? G[i - 1][j - 1] + 1 : 0;
            G[i][j] = max3(L[i][j], G[i - 1][j], G[i][j - 1]);
        }
    }
    return G[m][n];
}
```

Note that, a simpler solution is that, we don't define `L` and just use `G`. In this case, the function is

- `G[i][j] = max{G[i - 1][j], G[i][j - 1], G[i - 1][j - 1] + 1}`
  (if `A[i - 1] == B[j - 1]`), or
- `G[i][j] = max{G[i - 1][j], G[i][j - 1], G[i - 1][j - 1]}` (otherwise)

Maximum Subarray

Idea: Very similar to Longest Increasing Subsequence. The final answer is in the minimum/maximum in the `dp[i]`. Also, it is very similar to Best Time to Buy and Sell Stocks I: make the price change as the value of the array, then find the maximum of the subarray. Only note that the final result must be larger than 0.

Maximum Product Subarray

Idea: define two `dp`: `dpMax[i], dpMin[i]`: 以`nums[i]`为结尾的子序列的最大值和最小值。So:

1. `dpMax[i] = max{ nums[i], nums[i]*dpMax[i-1], nums[i]*dpMin[i-1] }`,
2. `dpMin[i] = {nums[i], nums[i]*dpMax[i-1], nums[i]*dpMin[i-1]}`.
3. Find the maximum in `dpMax[i]`.

```cpp
inline pair<int, int> minmax3(int a, int b, int c)
{
    return pair<int, int>(min(a, min(b, c)), max(a, max(b, c)));
}
int maxProduct(vector<int>& nums)
{
    int n = nums.size();
    if (n == 0)
        return 0;
    vector<int> dpMax(n, 0), dpMin(n, 0);
    dpMax[0] = dpMin[0] = nums[0];
    for (int i = 1; i < n; i++)
    {
        pair<int, int> temp = minmax3
        (
            dpMax[i - 1] * nums[i],
            dpMin[i - 1] * nums[i],
            nums[i]
        );
        dpMax[i] = temp.second;
        dpMin[i] = temp.first;
    }
    return *max_element(dpMax.begin(), dpMax.end());
}
```

Triangle (Classic)

Idea: `dp[i][j] = min(dp[i-1][j], dp[i-1][j-1]) + triangle[i][j]`

```cpp
int minimumTotal(vector<vector<int>>& triangle)
{
    int n = triangle.size();
    if (n == 0)
        return 0;
    for (int i = 1; i < n; i++)
    {
        triangle[i][0] += triangle[i - 1][0];
        triangle[i][i] += triangle[i - 1][i - 1];
    }
    for (int i = 2; i < n; i++)
    {
        for (int j = 1; j < i; j++)
        {
            triangle[i][j] += min(triangle[i - 1][j - 1], triangle[i - 1][j]);
        }
    }
    return *min_element(triangle[n - 1].begin(), triangle[n - 1].end());
}
```

Ugly Number II (Check Again!!!)

Idea: Use `ptr2, ptr3, ptr5` points to the ugly numbers, the `i` th ugly number is `min{Ugly[ptr2]*2, Ugly[ptr3]*3, Ugly[ptr5]*5}`, and which one is the minimal will result in corresponding pointer `++`.
Note: Any ugly number must be represented as a former ugly number multiplied by `2, 3` or `5`. The `++` operator acts on the minimal value only, so the three pointers always cover the next ugly number.

```cpp
int nthUglyNumber(int n)
{
    if (n == 1)
        return 1;
    vector<int> dp(n + 1, 0);
    dp[1] = 1;
    int ptr2 = 1, ptr3 = 1, ptr5 = 1;
    int v1, v2, v3;
    for (int i = 2; i <= n; i++)
    {
        v1 = dp[ptr2] * 2;
        v2 = dp[ptr3] * 3;
        v3 = dp[ptr5] * 5;
        int minval = min(v1, min(v2, v3));
        dp[i] = minval;
        if (minval == v1)
            ptr2++;
        if (minval == v2)
            ptr3++;
        if (minval == v3)
            ptr5++;
    }
    return dp[n];
}
```

Best Time to Buy and Sell Stocks I/II/III/IV (Hard, Check Again!!!)

Idea: Please refer to a special note about these problems.

Maximal Square

Idea: define `dp[i][j]` as the max width ended with point `(i, j)`. Then `dp[i][j] = 0`, if `matrix[i][j] == 0`, and `dp[i][j] = min{dp[i-1][j-1], dp[i-1][j], dp[i][j-1]} + 1`. Find the maximum in `dp[i][j]`.

```cpp
inline int min3(int a, int b, int c)
{
    return min(a, min(b, c));
}
int maximalSquare(vector<vector<char>>& matrix)
{
    int m = matrix.size();
    int n = m ? matrix[0].size() : m;
    if (m * n == 0)
        return 0;
    vector<vector<int>> dp(m, vector<int>(n, 0));
    int maxlen = 0;
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (i == 0 || j == 0)
                dp[i][j] = matrix[i][j] == '1';
            else
                dp[i][j] = matrix[i][j] == '1' ?
                    min3(dp[i - 1][j],
                        dp[i][j - 1],
                        dp[i - 1][j - 1]) + 1 : 0;
            maxlen = max(maxlen, dp[i][j]);
        }
    }
    return maxlen * maxlen;
}
```

Longest Valid Parentheses (Hard, Check Again!!!)

Idea: There should be a **Parentheses-related Problem Set**.
Define `dp[i]` is the longest parentheses' length ended at `i`. Then we have functions:

1. `if s[i] == '(': dp[i] = 0;` a string ended with `(` cannot be valid.
2. `if s[i] == ')':`
    - `if s[i-1] == '(', dp[i] = dp[i-2] + 2;`
    - `if s[i-1] == ')',`
        - `if s[i - dp[i-1] - 1] == '(', dp[i] = dp[i - 1] + 2 + dp[i - dp[i-1] - 2]`
        - `dp[i] = 0`

Wildcard Matching (Hard, Rolling Array, Check Again!!!)

Idea: Define the state `P[i][j]` to be whether `s[0..i)` matches `p[0..j)`. The state equations are as follows:

- `P[i][j] = P[i - 1][j - 1] && (s[i - 1] == p[j - 1] || p[j - 1] == '?'), if p[j - 1] != '*';`
- `P[i][j] = P[i][j - 1] || P[i - 1][j - 1] || P[i - 1][j], if p[j - 1] == '*'.`

Note:

1. This problem must be solved with a rolling array, otherwise MLE error.
2. In Wildcard Matching, `'*'` can represent any sequence, not like Regular Expression Matching.

```cpp
bool isMatch(string s, string p)
{
    int m = s.size(), n = p.size();
    vector<vector<int>> dp(2, vector<int>(n + 1, 0));
    dp[0][0] = 1;
    for (int j = 1; j <= n; j++)
        dp[0][j] = dp[0][j - 1] && p[j - 1] == '*';
    for (int i = 1; i <= m; i++)
    {
        for (int j = 0; j <= n; j++)
        {
            if (j == 0)
            {
                dp[i % 2][j] = 0;
                continue;
            }
            if (p[j - 1] != '*')
                dp[i % 2][j] = dp[(i - 1) % 2][j - 1]
                    && (p[j - 1] == '?' || p[j - 1] == s[i - 1]);
            else
                dp[i % 2][j] = dp[i % 2][j - 1]
                    || dp[(i - 1) % 2][j - 1] || dp[(i - 1) % 2][j];
        }
    }
    return dp[m % 2][n];
}
```

Regular Expression Matching

Idea: Very similar to Wildcard Matching, but here `'*'` means zero or more of the preceding element. So **once there is a `'*'`, it must NOT be the first character**.

1. If `p[j - 1]` is not `'*'`, then it's the same as Wildcard Matching: `dp[i][j] = dp[i - 1][j - 1] && (p[j - 1] == '.' || s[i - 1] == p[j - 1])`;
2. Otherwise, any condition will cause `dp[i][j] == 1`:
    1. `dp[i][j - 1]`：前 `j - 1` 个正则表达式即可匹配前 `i` 个字符，最后一个又是 `'*'`，那这个 `'*'` 代表正则表达式最后一个字符出现**1**次，即可匹配；
    2. `dp[i][j - 2]`：前 `j - 2` 个正则表达式即可匹配前 `i` 个字符，最后一个又是 `'*'`，那这个 `'*'` 代表正则表达式最后一个字符出现**0**次，即可匹配；
    3. `dp[i - 1][j - 1]` 匹配，同时 `p[j - 2] == '.'`(意味着最后一个字符可以任意匹配)或者 `p[j - 2] == s[i - 1]`(意味着最后一个字符确实是重复的)，即可匹配；
    4. `dp[i - 1][j]` 匹配，同时 `p[j - 2] == '.'`(意味着最后一个字符可以任意匹配)或者 `p[j - 2] == s[i - 1]`(意味着最后一个字符确实是重复的)，即可匹配；如：`aaaa` 和 `a*`。

```cpp
bool isMatch(string s, string p)
{
    int m = s.size(), n = p.size();
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    dp[0][0] = 1;
    for (int j = 1; j <= n; j++)
        dp[0][j] = j >= 2 && p[j - 1] == '*' && dp[0][j - 2];
    for (int i = 1; i <= m; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            if (p[j - 1] != '*')
                dp[i][j] = dp[i - 1][j - 1] && (p[j - 1] == '.' || s[i - 1] == p[j - 1]);
            else
            {
                dp[i][j] = dp[i][j - 1] || dp[i][j - 2]
                || (dp[i - 1][j - 1] && (p[j - 2] == '.' || s[i - 1] == p[j - 2]))
                || (dp[i - 1][j] && (p[j - 2] == '.' || s[i - 1] == p[j - 2]));
            }
        }
    }
    return dp[m][n];
}
```

Maximal Rectangle (Hard, Check Again!!!)

Idea: Use Largest Rectangle in Histogram to solve. Currently don't understand DP solution.

Palindrome Partitioning II (Hard, Classic, Check Again!!!)

Idea:

1. Generate DP for checking whether a substring of `s` is palindrome. Define `dp[i][j]` as whether `s[i,...,j]` is palindrome, then `dp[i][i] = 1` and `dp[i][i+1] = 1, if s[i] == s[i+1]`. `dp[i][j] = dp[i+1][j-1] && s[i]==s[j]`.
2. Define another `dp[i]` as the minCut for the first `i` characters of `s`. Initialize `dp[i] = i - 1` because the maximal cut for the first `i` characters is `i-1`. Then `dp[i] = min{ dp[j] + 1, for j = 0,1,...,i-1: isPalindrome(j,i-1) }`. Return `dp[n]`.

```
void genIsPalindrome(vector<vector<int>>& P, string& s)
{
    int n = P.size();
    for (int i = 0; i < n; i++)
        P[i][i] = 1;
    for (int i = 0; i < n - 1; i++)
        P[i][i + 1] = s[i] == s[i + 1];
    if (n <= 2)
        return;
    int i = 0, j = 2, k = 3;
    while (j < n)
    {
        P[i][j] = P[i + 1][j - 1] && s[i] == s[j];
        i++;
        j++;
        if (j == n)
        {
            i = 0;
            j = k;
            k++;
        }
    }
}
int minCut(string s)
{
    int n = s.size();
    if (n == 0)
        return 0;
    vector<vector<int>> isPalindrome(n, vector<int>(n, 0));
    genIsPalindrome(isPalindrome, s);
    vector<int> dp(n + 1, 0);
    dp[0] = -1;
    for (int i = 1; i <= n; i++)
    {
        int minval = INT_MAX;
        for (int j = 0; j < i; j++)
        {
            if (isPalindrome[j][i - 1])
            {
                minval = min(minval, dp[j] + 1);
            }
        }
        dp[i] = minval;
    }
    return dp[n];
}
```

Scramble String (Hard, Check Again!!!)

Idea: Currently only a memorized dfs version, no DP yet!

Interleaving String (Classic)

Idea: **Bi-sequence DP**. Define `dp[i][j]` as whether `s1` first `i` characters and `s2` first `j` characters can be interleaving string for `s3` first `i+j` characters. Then the function is: `dp[i][j] = (dp[i-1][j] && s1[i-1]==s3[i+j-1]) || (dp[i][j-1] && s2[j-1]==s3[i+j-1])`.

```cpp
bool isInterleave(string s1, string s2, string s3)
{
    int m = s1.size(), n = s2.size(), k = s3.size();
    if (k != m + n)
        return false;
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    for (int i = 0; i <= m; i++)
    {
        for (int j = 0; j <= n; j++)
        {
            dp[i][j] = i == 0 && j == 0
            || i > 0 && dp[i - 1][j] && s1[i - 1] == s3[i + j - 1]
            || j > 0 && dp[i][j - 1] && s2[j - 1] == s3[i + j - 1];
        }
    }
    return dp[m][n];
}
```

For such kind of `O(n^2)` complexity DP problem, it can also be optimised by **Rolling Array** (2D Rolling Array).

```cpp
bool isInterleave(string s1, string s2, string s3)
{
    int n1 = s1.size(), n2 = s2.size(), n3 = s3.size();
    if (n1 + n2 != n3)
        return false;
    if (n1 == 0)
        return s2 == s3;
    if (n2 == 0)
        return s1 == s3;
    vector<vector<int>> dp(2, vector<int>(n2 + 1, 0));
    dp[0][0] = 1;
    for (int i = 1; i <= n2; i++)
        dp[0][i] = dp[0][i - 1] && s2[i - 1] == s3[i - 1];
    for (int i = 1; i <= n1; i++)
    {
        for (int j = 0; j <= n2; j++)
        {
            if (j == 0)
                dp[i % 2][j] = dp[(i - 1) % 2][j] && s1[i - 1] == s3[i - 1];
            else
            {
                dp[i % 2][j] =
                    (dp[(i - 1) % 2][j] && s1[i - 1] == s3[i + j - 1])
                    || (dp[i % 2][j - 1] && s2[j - 1] == s3[i + j - 1]);
            }
        }
    }
    return dp[n1 % 2][n2];
}
```

Edit Distance (Classic)

Idea: **Bi-sequence DP**. Define `dp[i][j]` as the edit distance from `s1[0,...,i-1]` to `s2[0,...,j-1]`. Then `dp[i][j] = min{dp[i-1][j] + 1, dp[i][j-1] + 1, dp[i-1][j-1] + s1[i-1]!=s2[j-1]}`.

```cpp
inline int min3(int a, int b, int c)
{
    return min(a, min(b, c));
}
int minDistance(string word1, string word2)
{
    int m = word1.size();
    int n = word2.size();
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    for (int i = 1; i <= n; i++)
        dp[0][i] = i;
    for (int i = 1; i <= m; i++)
        dp[i][0] = i;
    for (int i = 1; i <= m; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            dp[i][j] = min3(dp[i - 1][j] + 1,
                dp[i][j - 1] + 1,
                dp[i - 1][j - 1] + (word1[i - 1] != word2[j - 1]));
        }
    }
    return dp[m][n];
}
```

Word Break

Idea: Define `f[i]` is for first `i` characters, whether it can be broken into strings within the dictionary. So `f[i] = any one f[j] && s.substr(j, i - j), 0 <= j < i`. But we cannot loop `j` from `0` because the word length `i - j` must be `<= maxlen` of the words in dictionary.

```cpp
bool wordBreak(string s, unordered_set<string> &dict)
{
    int maxlen = 0;
    for (auto it = dict.begin(); it != dict.end(); it++)
        maxlen = max(maxlen, (int)it->size());
    int n = s.size();
    vector<int> dp(n + 1, 0);
    dp[0] = 1;
    for (int i = 1; i <= n; i++)
    {
        for (int j = i - 1; j >= max(0, i - maxlen); j--)
        {
            if (dp[j] && dict.count(s.substr(j, i - j)))
            {
                dp[i] = true;
                break;
            }
        }
    }
    return dp[n];
}
```

Dungeon Game

Idea: **Coordinate DP**.

1. Define `dp[i][j]` as the minimal health point required for `dungeon[i][j]`.
2. Then we have the function: `dp[i][j] = max(min(dp[i+1][j], dp[i][j+1]) - dungeon[i][j], 1)`.
3. Initialization: `dp[0][0] = max(1 - dungeon[i][j], 1)` and `dp[m-1][j] = max(dp[m-1][j+1] - dungeon[i][j], 1)` and `dp[i][n-1] = max(dp[i+1][n-1] - dungeon[i][j], 1)`.

Distinct Subsequences (Classic)

Idea: **Bi-sequence DP**.

1. Define `dp[i][j]` as the number of deleting ways from `s[0,...,i-1]` to `t[0,...,j-1]`.
2. Then `dp[i][j] = dp[i-1][j], if s[i-1]!=t[j-1]` (meaning: if `s[i-1]!=t[j-1]`, `s[i-1]` must be also deleted when transform `s[0,...,i-2]` to `t[0,...,j-1]`), or `dp[i][j] = dp[i-1][j-1] + dp[i-1][j], if s[i-1]==t[j-1]` (meaning: if `s[i-1]==t[j-1]`, beside the ways from `s[0,...,i-2]` to `t[0,...,j-1]`, we can also transform `s[0,...,i-2]` to `t[0,...,j-2]`).

```
int numDistinct(string &S, string &T)
{
    int m = S.size(), n = T.size();
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    for (int i = 0; i <= m; i++)
        dp[i][0] = 1;
    for (int i = 1; i <= m; i++)
    {
        for (int j = 1; j <= min(i, n); j++)
        {
            dp[i][j] = dp[i - 1][j]
                + (S[i - 1] == T[j - 1]) * dp[i - 1][j - 1];
        }
    }
    return dp[m][n];
}
```

Coin Change (Hard!!!)

Idea: dfs will TLE. It should be a backpacking DP.

Coins in a Line

Idea: **Strategy DP**. Define `dp[i]` is for `i` current player must win. So `dp[i] = !(dp[i - 1] && dp[i - 2])`.

Coins in a Line II

Idea: **Strategy DP**. Define `dp[i]` is for only `i` coins left, the maximum coins first player can get. Then `dp[i] = max{A[n - i] + min(dp[i - 2], dp[i - 3]), A[n - i] + A[n - i + 1] + min(dp[i - 3], dp[i - 4])}`.

Coins in a Line III

Idea: Strategy DP and Interval DP. Define `dp[i][j]` as for the range `A[i ~ j]`, the maximal coins first player can obtain. Then like Coins in a Line II, the function should be `dp[i][j] = max(Ai + min(dp[i + 2][j], dp[i + 1][j - 1]), Aj + min(dp[i + 1][j - 1], dp[i][j - 2]))`. Finally check `dp[0][n - 1] > sum / 2`.

Backpack

Idea: Backpack DP.

1. Define `dp[i][v]` as first `i` objects put into volume `v`, the maximal score.
2. `dp[i][v]` has two cases: (1) if `A[i - 1] > v`, then `dp[i - 1][v]` (反正`A[i-1]`也不可能放进去了); (2) if `A[i - 1] <= v`, then `max(dp[i - 1][j]（不放A[i-1]）, dp[i - 1][v - A[i-1]]（放A[i-1]） + A[i-1])`.

Minimum Adjustment Cost

Idea: Backpack DP. Define `dp[i][k]` as the first `i` elements minimal adjusted cost if `A[i - 1]` set to `k`. So `dp[0][k] = 0` and `dp[i][k] = min{dp[i - 1][m]} + |k - A[i - 1]|` where `m = k - target, ..., k + target`.

```
int MinAdjustmentCost(vector<int> A, int target)
{
    int n = A.size();
    if (n == 0)
        return 0;
    int minval = *min_element(A.begin(), A.end());
    int maxval = *max_element(A.begin(), A.end());
    for_each(A.begin(), A.end(), [minval](int& x){ x-= minval; });
    maxval -= minval;
    vector<vector<int>> dp(n + 1, vector<int>(maxval + 1, INT_MAX));
    fill(dp[0].begin(), dp[0].end(), 0);
    for (int i = 1; i <= n; i++)
    {
        for (int k = 0; k <= maxval; k++)
        {
            int low = max(0, k - target);
            int high = min(maxval, k + target);
            auto it = min_element(dp[i - 1].begin() + low, dp[i - 1].begin() + high + 1);
            dp[i][k] = *it == INT_MAX ? INT_MAX : *it + abs(k - A[i - 1]);
        }
    }
    return *min_element(dp[n].begin(), dp[n].end());
}
```

k Sum

Idea: Backpack DP. Define `dp[i][j][v]` as the number of for first `i` elements, selecting `j` of them whose sum is `v`. So `dp[i][0][0] = 1`, other face / edge elements (the matrix is a cube, `dp[0][0][0]` is the vertex, it also has 3 faces and 3 edges). Then the function should be: `dp[i][j][v] = dp[i - 1][j][v]` if `A[i - 1] > v`; and `dp[i][j][v] = dp[i - 1][j][v]（不放A[i-1]） + dp[i - 1][j - 1][v - A[i - 1]]（放入A[i-1]）`.

```cpp
int kSum(vector<int> A, int k, int target)
{
    int n = A.size();
    if (target < 0 || k < 0)
        return 0;
    vector<vector<vector<int>>> dp(n + 1,
        vector<vector<int>>(k + 1, vector<int>(target + 1, 0)));
    // i(n), j(k), v(target)
    for (int i = 0; i <= n; i++)
        dp[i][0][0] = 1;
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= k; j++)
        {
            for (int v = 1; v <= target; v++)
            {
                dp[i][j][v] = A[i - 1] > v ? dp[i - 1][j][v] :
                    dp[i][j][v] = dp[i - 1][j][v] + dp[i - 1][j - 1][v - A[i - 1]];
            }
        }
    }
    return dp[n][k][target];
}
```

Coin Change

Idea: Backpack DP. `dp[i][j] = min(1 + dp[i][j - A[i-1]], dp[i-1][j]) or dp[i-1][j]`

```cpp
int coinChange(vector<int>& coins, int amount)
{
    int n = coins.size();
    int dp[2][amount + 1];
    dp[0][0] = 0;
    for (int i = 1; i <= amount; i++)
        dp[0][i] = INT_MAX;
    for (int i = 1; i <= n; i++)
    {
        for (int j = 0; j <= amount; j++)
        {
            if (coins[i - 1] > j)
                dp[i % 2][j] = dp[(i - 1) % 2][j];
            else
            {
                int temp = dp[i % 2][j - coins[i - 1]];
                dp[i % 2][j] = min(dp[(i - 1) % 2][j],
                    temp == INT_MAX ? INT_MAX : temp + 1);
            }
        }
    }
    return dp[n % 2][amount] == INT_MAX ? -1 : dp[n % 2][amount];
}
```

Data Structure (Hash / Heap / Stack / Queue)

Longest Consecutive Sequence

Idea: **HashSet**
Make two sets, one contains all nums, and the other contains processed nums. Loop the num array, for any num in the array, if it is not in the processed set, then put it into the set and also check its larger ones and less ones exists: if exist, then put it into the sets.

Largest Rectangle in Histogram

Idea: **Increasing-valued Index Stack (递增栈)**
对于任何一个点的最大面积是这样求的：1) 求出其左边第一个比它小的索引`leftLowerIndex`，求出其右边第一个比它小的索引`rightLowerIndex`. 2) `(rightLowerIndex - leftLowerIndex - 1) * height`. 递增栈就可以满足此要求。当推入一个数时，把栈顶比它大的数逐个pop出，这些被pop出的数就被视为当前点，而即将被推入的数就是它们的`rightLowerIndex`.

```
int largestRectangleArea(vector<int> &height)
{
    stack<int> S;
    int n = height.size();
    int maxarea = 0;
    for (int i = 0; i <= n; i++)
    {
        int h = i < n ? height[i] : -1;
        while (!S.empty() && height[S.top()] >= h)
        {
            int curIndex = S.top();
            S.pop();
            int leftLowerIndex = S.empty() ? -1 : S.top();
            int area = (i - leftLowerIndex - 1)*height[curIndex];
            maxarea = max(maxarea, area);
        }
        S.push(i);
    }
    return maxarea;
}
```

Stack Sorting

Idea: **Decreasing Stack** - `O(n^2)`

Max Tree

Idea: **Decreasing Stack**

1. Keep the `TreeNode*` in the stack decreasing order.
2. When there is a new `TreeNode*` coming to the stack, iteratively pop-out the top of stack if they are smaller than the new node. During the populating, create a tree with each node being the right child of the node pop-out next, at last the whole tree as the left child of the new node. Push the new node into the stack.
3. At last, push a dummy `TreeNode` with value `INT_MAX`, and finally its left child is the max tree we want.

```cpp
TreeNode* maxTree(vector<int> A)
{
    stack<TreeNode*> S;
    int n = A.size();
    TreeNode dummy(INT_MAX);
    for (int i = 0; i <= n; i++)
    {
        int num = i < n ? A[i] : INT_MAX;
        TreeNode* node = i < n ? new TreeNode(num) : &dummy;
        TreeNode* topnode = NULL;
        while (!S.empty() && num > S.top()->val)
        {
            topnode = S.top();
            S.pop();
            if (!S.empty())
                S.top()->right = topnode;
        }
        node->left = topnode;
        S.push(node);
    }
    return dummy.left;
}
```

Heapify

Idea: **Heap**. The last node who has children/child holds the index `m` satisfying `m = HeapSize / 2 - 1`.

- 由后往前，每个节点都做大者下沉操作。

```cpp
for (int i = m; i >= 0; i--)
    shiftdown(A, i);
```

- 由前向后，每个节点都做小者上浮操作

```cpp
for (int i = 0; i < n; i++)
    shiftup(A, i);
```

ShiftUp: A[i] => A[(i-1) / 2],
向前上浮，使得已有的Heapified A[0 ~ i-1]加上A[i]后依然Heapified.
如果是向后下沉，无法保证A[0 ~ i-1]加上A[i]后依然是Heapified.
(A[i]可能是后面中更小的元素，比A[0 ~ i-1]都小)

ShiftDown: A[i] => A[2 * i + 1] or A[2 * i + 2],
向后下沉，使得已有的Heapified A[i+1 ~ n-1]加上A[i]后依然Heapified.
如果是向前上浮，无法保证A[i+1 ~ n-1]加上A[i]后依然是Heapified.
(A[i]可能是前面中更大的元素，比A[i+1 ~ n-1]都小)

Number of Airplanes in the Sky

Idea: Scan-line Problem. Transfer the intervals into `node{int time; int flag}`. Then sort the nodes. Note that, the node representing landing airplane should be prior to the taking-off ones, i.e. compare function must be carefully handled when indices are the same.

```
class Node
{
public:
    Node() : time(-1), is_start(true) {}
    Node(int t, bool s) : time(t), is_start(s) {}
    int time;
    bool is_start;
};
struct TComp
{
    bool operator()(const Node& a, const Node& b) const
    {
        if (a.time == b.time) // time index same, end point prior to start point
            return (int)a.is_start < (int)b.is_start;
        else
            return a.time < b.time;
    }
};
int countOfAirplanes(vector<Interval> &airplanes)
{
    int n = airplanes.size();
    if (n == 0)
        return 0;
    vector<Node> nodes(2 * n);
    for (int i = 0; i < n; i++)
    {
        int start_time = airplanes[i].start;
        int end_time = airplanes[i].end;
        nodes[i] = Node(start_time, true);
        nodes[i + n] = Node(end_time, false);
    }
    TComp CompObj;
    sort(nodes.begin(), nodes.end(), CompObj);
    int count = 0, maxcount = 0;
    for (int i = 0; i < nodes.size(); i++)
    {
        if (nodes[i].is_start)
        {
            count++;
            maxcount = max(maxcount, count);
        }
        else
        {
            count--;
        }
    }
    return maxcount;
}
```

Insert Interval

Idea: Scan line.

```cpp
class Node
{
public:
    Node(int v, int c) : val(v), flag(c) {}
    Node() : val(-1), flag(0) {}
    int val;
    int flag;
};
struct TComp
{
    bool operator()(const Node& a, const Node& b) const
    {
        if (a.val == b.val)
            return a.flag < b.flag;
        else
            return a.val < b.val;
    }
};
vector<Interval> insert(vector<Interval> &intervals, Interval newInterval)
{
    vector<Interval> rst;
    int n = intervals.size();
    if (n == 0)
        return vector<Interval>(1, newInterval);
    vector<Node> nodes(2 * n);
    for (int i = 0; i < n; i++)
    {
        int start = intervals[i].start;
        int end = intervals[i].end;
        nodes[i] = Node(start, 0);
        nodes[i + n] = Node(end, 1);
    }
    nodes.push_back(Node(newInterval.start, 0));
    nodes.push_back(Node(newInterval.end, 1));
    TComp CompObj;
    sort(nodes.begin(), nodes.end(), CompObj);
    n = nodes.size();
    int last = 0, scount = 1;
    for (int i = 1; i < n; i++)
    {
        if (nodes[i].flag == 0)
            scount++;
        else
            scount--;
        if (scount == 0)
        {
            rst.push_back(Interval(nodes[last].val, nodes[i].val));
            last = i + 1;
        }
    }
    return rst;
}
```

A more clean code:

```cpp
vector<Interval> insert(vector<Interval> &intervals, Interval newInterval)
{
    intervals.push_back(newInterval);
    sort(intervals.begin(), intervals.end(),
        [](const Interval& a, const Interval& b)
        { return a.start < b.start; });
    vector<Interval> rst;
    for (int i = 0; i < intervals.size(); i++)
    {
        if (rst.empty() || intervals[i].start > rst.back().end)
            rst.push_back(intervals[i]);
        else
            rst.back().end = max(rst.back().end, intervals[i].end);
    }
    return rst;
}
```

Number of Islands

Idea: **Union-Find** or BFS.

```cpp
// Union-Find code
int find(int index, vector<int>& father) // basic find
{
    while (index != father[index])
        index = father[index];
    return index;
}
int compress_find(int index, vector<int>& father) // find with compressing path
{
    int rst = index;
    while (rst != father[rst])
        rst = father[rst];
    while (index != father[index])
    {
        int fa = father[index];
        father[index] = rst;
        index = fa;
    }
    return rst;
}
// Merge:
void merge()
{
    vector<int> father;
    // initialize father
    // ... ...
    int v0, v1; // for two point nodes
    int fa0 = compress_find(v0, father);
    int fa1 = compress_find(v1, father);
    father[fa0] = fa1;
}
```

Add and Search Word - Data structure design

Idea: **Trie**. `TrieNode { bool existed; TrieNode* children[26]; }`. `root` points to no character. The final pointer should be at the final character, never being `NULL`. Once `NULL`, means not existed.

```cpp
class WordDictionary
{
public:
    class TrieNode
    {
    public:
        TrieNode() : existed(false) { memset(children, 0, sizeof(TrieNode*) * 26); }
        bool existed;
        TrieNode* children[26];
    };

    WordDictionary()
    {
        root = new TrieNode();
    }

    void addWord(string word)
    {
        TrieNode* p = root;
        for (int i = 0; i < word.size(); i++)
        {
            char c = word[i];
            if (p->children[c - 'a'] == NULL)
                p->children[c - 'a'] = new TrieNode();
            p = p->children[c - 'a'];
        }
        // after the loop, p must be at the final character of the word.
        p->existed = true;
    }

    bool search_helper(string& word, int s, TrieNode* root)
    {
        // this function starting point is root is the original root, s is 0.
        // so root must be at the parent of the character which will be processed.
        // after processing, the pointer should be at the final character.
        if (s == word.size())
            return root->existed;
        TrieNode *p = root;
        for (int i = s; i < word.size() && p; i++)
        {
            char c = word[i];
            if (c != '.')
                p = p->children[c - 'a'];
            else
            {
                for (char tc = 'a'; tc <= 'z'; tc++)
                {
                    word[i] = tc;
                    if (search_helper(word, i, p))
                        return true;
                }
                word[i] = '.';
                return false;
            }
        }
        return p && p->existed;
    }
    bool search(string word)
    {
        return search_helper(word, 0, root);
```

```
    }

    TrieNode *root;
};
```

Trapping Rain Water II

Idea: Using a heap to store the boundary. The element with minimal height should be visited to trap to its neighbors.

Sliding Window Median, Data Stream Median

Idea:

1. For Data Stream Median, we only need to use system's `priority_queue` as heap.

```
priority_queue<int, vector<int>, greater<int>> pq; // minHeap
priority_queue<int, vector<int>, less<int>> pq; // maxHeap
```

```
class MedianFinder
{
public:
    // Adds a number into the data structure.
    void addNum(int num)
    {
        int n1 = maxHeap.size(), n2 = minHeap.size();
        if (n1 == n2)
        {
            maxHeap.push(num);
            int top1 = maxHeap.top();
            int top2 = minHeap.empty() ? INT_MAX : minHeap.top();
            if (top1 > top2)
            {
                maxHeap.pop();
                minHeap.pop();
                maxHeap.push(top2);
                minHeap.push(top1);
            }
        }
        else
        {
            maxHeap.push(num);
            int top1 = maxHeap.top();
            maxHeap.pop();
            minHeap.push(top1);
        }
    }
    // Returns the median of current data stream
    double findMedian()
    {
        int n1 = maxHeap.size(), n2 = minHeap.size();
        if (n1 == n2 && n1 == 0)
            return INT_MIN;
        if (n1 == n2)
            return 0.5 * ((double)maxHeap.top() + (double)minHeap.top());
        else
            return maxHeap.top();
    }
    priority_queue<int> maxHeap;
    priority_queue<int, vector<int>, greater<int>> minHeap;
};
```

2. In Sliding Window Median, we need to add element and remove element. So a `HashHeap` needs to be implemented to support remove element. A hash heap supporting (1) duplicates (2) removing elements is as follows:

```cpp
class HashHeap
{
public:
    class Node
    {
    public:
        Node() : index(0), num(0) {}
        Node(int index_, int num_) : index(index_), num(num_) {}
        int index;
        int num;
    };

    HashHeap(bool isMinHeap_) : isMinHeap(isMinHeap_), sz(0) { data.resize(1, 0); }

    bool empty() const { return data.size() == 1; }
    int size() const { return sz; }
    void push(int val)
    {
        auto it = hashmap.find(val);
        if (it != hashmap.end())
            it->second.num++;
        else
        {
            data.push_back(val);
            int index = data.size() - 1;
            hashmap[val] = Node(index, 1);
            shiftUp(index);
        }
        sz++;
    }

    void pop()
    {
        if (data.size() == 1)
            return;
        int val = data[1];
        auto it = hashmap.find(val);
        if (it->second.num > 1)
            it->second.num--;
        else
        {
            swap_data(1, data.size() - 1);
            hashmap.erase(it);
            data.pop_back();
            shiftDown(1);
        }
        sz--;
    }

    int top() const
    {
        if (data.size() == 1)
            return INT_MIN;
        return data[1];
    }

    void remove(int val)
    {
        auto it = hashmap.find(val);
        if (it == hashmap.end())
```

```
            return;
        if (it->second.num > 1)
            it->second.num--;
        else
        {
            int index = it->second.index;
            swap_data(index, data.size() - 1);
            hashmap.erase(it);
            data.pop_back();
            shiftUp(index);
            shiftDown(index);
        }
        sz--;
    }

private:
    void swap_data(int id1, int id2)
    {
        int val1 = data[id1], val2 = data[id2];
        Node n1 = hashmap[val1], n2 = hashmap[val2];
        swap(data[id1], data[id2]);
        hashmap[val1] = Node(id2, n1.num);
        hashmap[val2] = Node(id1, n2.num);
    }
    void shiftUp(int index)
    {
        while (1)
        {
            int parent = index / 2;
            if (parent < 1 || !compare(data[index], data[parent]))
                break;
            swap_data(index, parent);
            index = parent;
        }
    }
    void shiftDown(int index)
    {
        while (1)
        {
            int lchild = index * 2, rchild = index * 2 + 1;
            int whichone;
            if (lchild >= data.size())
                break;
            else if (rchild >= data.size())
            {
                whichone = lchild;
            }
            else
            {
                int lval = data[lchild], rval = data[rchild];
                whichone = compare(lval, rval) ? lchild : rchild;
            }
            if (compare(data[index], data[whichone]))
                break;
            swap_data(index, whichone);
            index = whichone;
        }
    }
    bool compare(int a, int b)
    {
        return isMinHeap ? a < b : !(a < b);
```

```
122.     }
123. private:
124.     unordered_map<int, Node> hashmap;
125.     vector<int> data;
126.     bool isMinHeap;
127.     int sz;
128. };
```

And, the main code is as follows:

```cpp
void push_value(HashHeap& maxHeap, HashHeap& minHeap, int val)
{
    int sz1 = maxHeap.size(), sz2 = minHeap.size();
    int M = sz1 == 0 ? INT_MIN : maxHeap.top();
    int m = sz2 == 0 ? INT_MAX : minHeap.top();
    if (sz1 == sz2)
    {
        if (val < m)
        {
            maxHeap.push(val);
        }
        else
        {
            maxHeap.push(m);
            minHeap.pop();
            minHeap.push(val);
        }
    }
    else
    {
        if (val < m)
        {
            maxHeap.push(val);
            M = maxHeap.top();
            minHeap.push(M);
            maxHeap.pop();
        }
        else
        {
            minHeap.push(val);
        }
    }
}

void pop_value(HashHeap& maxHeap, HashHeap& minHeap, int val)
{
    int sz1 = maxHeap.size(), sz2 = minHeap.size();
    int M = sz1 == 0 ? INT_MIN : maxHeap.top();
    int m = sz2 == 0 ? INT_MAX : minHeap.top();
    if (sz1 == sz2)
    {
        if (val < m)
        {
            maxHeap.remove(val);
            maxHeap.push(m);
            minHeap.pop();
        }
        else
        {
            minHeap.remove(val);
        }
    }
    else
    {
        if (val < m)
        {
            maxHeap.remove(val);
        }
        else
        {
```

```
            minHeap.remove(val);
            minHeap.push(M);
            maxHeap.pop();
        }
    }
}

vector<int> medianSlidingWindow(vector<int> &nums, int k) {
    // write your code here
    int n = nums.size();
    vector<int> rst;
    if (n < k || k <= 0)
        return rst;
    HashHeap maxHeap(false), minHeap(true);
    for (int i = 0; i < n; i++)
    {
        if (i < k)
        {
            push_value(maxHeap, minHeap, nums[i]);
        }
        else
        {
            rst.push_back(maxHeap.top());
            push_value(maxHeap, minHeap, nums[i]);
            pop_value(maxHeap, minHeap, nums[i - k]);
        }
    }
    rst.push_back(maxHeap.top());
    return rst;
}
```

Sliding Window Maximum

Idea:

1. brute-force `O()`
2. heap. 由于需要删除指定元素，故需要实现Hash Heap. `O(nlogk)`
3. **deque** = stack + queue. 每个元素只有一次进入和排出deque，故为 `O(n)` .
    1. 递减栈 `=>` 先进去的且比当前数值小的肯定不会是最大值，队列头为最大值。
    2. 队列 `=>` 队列头的索引显示如果超出sliding window了，则要pop掉。

```cpp
vector<int> maxSlidingWindow(vector<int> &nums, int k)
{
    int n = nums.size();
    vector<int> rst;
    if (n < k || k <= 0)
        return rst;
    deque<int> dq;
    rst.resize(n - k + 1, 0);
    for (int i = 0; i < n; i++)
    {
        while (!dq.empty() && dq.front() <= i - k)
            dq.pop_front();
        while (!dq.empty() && nums[dq.back()] < nums[i])
            dq.pop_back();
        dq.push_back(i);
        if (i >= k - 1)
            rst[i - k + 1] = nums[dq.front()];
    }
    return rst;
}
```

Building Outline

Idea: Scan-line + Hash Heap. **Scan-line problem must carefully process compare function! how to handle same index?**

```
class Node
{
public:
    Node() : index(-1), height(INT_MIN), is_start(true) {}
    Node(int id, int h, bool flag) : index(id), height(h), is_start(flag) {}
    int index;
    int height;
    bool is_start;
};
struct TComp // <= must be carefully handled!
{
    bool operator()(const Node& a, const Node& b) const
    {
        if (a.index == b.index)
        { // first add higher building, first remove lower building,
          // first handle start building, make sure no gap everywhere.
            if (a.is_start && b.is_start)
                return a.height > b.height;
            else if (!a.is_start && !b.is_start)
                return a.height < b.height;
            else
                return (int)a.is_start > (int)b.is_start;
        }
        else
            return a.index < b.index;
    }
};
vector<vector<int> > buildingOutline(vector<vector<int> > &buildings)
{
    int n = buildings.size();
    vector<vector<int>> rst;
    if (n == 0)
        return rst;
    // scan line: store line segments using end points, then sort
    vector<Node> nodes(n * 2);
    for (int i = 0; i < n; i++)
    {
        int start = buildings[i][0];
        int end = buildings[i][1];
        int height = buildings[i][2];
        nodes[i].index = start;
        nodes[i].height = height;
        nodes[i].is_start = true;
        nodes[i + n].index = end;
        nodes[i + n].height = height;
        nodes[i + n].is_start = false;
    }
    TComp CompObj;
    sort(nodes.begin(), nodes.end(), CompObj);
    HashHeap hp(false); // <= max hash heap
    int start_index = -1, curHeight = 0;
    for (int i = 0; i < nodes.size(); i++)
    {
        Node node = nodes[i];
        if (node.is_start)
        {
            if (hp.empty())
            {
                hp.push(node.height);
                start_index = node.index;
```

```
            }
            else
            {
                curHeight = hp.top();
                hp.push(node.height);
                if (hp.top() > curHeight)
                {
                    vector<int> temp = {start_index, node.index, curHeight};
                    // note above, it is curHeight, not node.height or hp.top()!!!
                    rst.push_back(temp);
                    start_index = node.index;
                }
            }
        }
        else
        {
            curHeight = hp.top();
            hp.remove(node.height);
            if (hp.empty() || hp.top() < curHeight)
            {
                vector<int> temp = {start_index, node.index, curHeight};
                rst.push_back(temp);
                start_index = node.index;
            }
        }
    }
    return rst;
}
```

Kth Smallest Number in Sorted Matrix

Idea: use a heap to store candidates.

```cpp
class Node
{
public:
    Node(int u, int v, int val) : u(u), v(v), val(val) {}
    Node() : u(-1), v(-1), val(-1) {}
    int u;
    int v;
    int val;
};
struct TComp
{
    bool operator()(const Node& a, const Node& b) const
    {
        return a.val > b.val;
    }
};
int kthSmallest(vector<vector<int> > &matrix, int k)
{
    int m = matrix.size();
    int n = m ? matrix[0].size() : m;
    if (m * n < k || k <= 0)
        return INT_MAX;
    priority_queue<Node, vector<Node>, TComp> pq;
    vector<vector<int>> visited(m, vector<int>(n, 0));
    pq.push(Node(0, 0, matrix[0][0]));
    Node rst;
    int offset[2][2] = { {1, 0}, {0, 1} };
    for (int i = 0; i < k; i++)
    {
        rst = pq.top();
        for (int k = 0; k < 2; k++)
        {
            int y = rst.u + offset[k][0];
            int x = rst.v + offset[k][1];
            if (y < m && x < n && visited[y][x] == 0)
            {
                pq.push(Node(y, x, matrix[y][x]));
                visited[y][x] = 1;
            }
        }
        pq.pop();
    }
    return rst.val;
}
```

Valid Parentheses

Idea: Stack.

```cpp
bool isValidParentheses(string& s)
{
    stack<char> S;
    int n = s.size();
    if (n % 2 == 1)
        return false;
    for (int i = 0; i < n; i++)
    {
        if (s[i] == '(' || s[i] == '[' || s[i] == '{')
            S.push(s[i]);
        else
        {
            if (S.empty())
                return false;
            switch (s[i])
            {
            case ')':
                if (S.top() == '(')
                    S.pop();
                else
                    return false;
                break;
            case ']':
                if (S.top() == '[')
                    S.pop();
                else
                    return false;
                break;
            case '}':
                if (S.top() == '{')
                    S.pop();
                else
                    return false;
                break;
            }
        }
    }
    return S.empty();
}
```

Animal Shelter

Idea: Remove non-front element from a queue? Put the front elements into the queue's tail and count them.

Kth Smallest Sum In Two Sorted Arrays

Idea: when pop the top of the `priority_queue`, which is represented as `(x, y, sum)`, we add `(x + 1, y, sum1)` and `(x, y + 1, sum2)` to the `priority_queue`.

```
class Point
{
public:
    Point() {}
    Point(int ia, int ib, int sum)
        : ia(ia), ib(ib), sum(sum) {}
    int ia, ib, sum;
};
struct TComp
{
    bool operator()(const Point& a, const Point& b) const
    {
        return a.sum > b.sum;
    }
};
int kthSmallestSum(vector<int>& A, vector<int>& B, int k)
{
    int m = A.size(), n = B.size();
    if (m == 0 && n == 0)
        return 0;
    if (m == 0 && n != 0)
        return k < n ? B[k - 1] : 0;
    if (m != 0 && n == 0)
        return k < m ? A[k - 1] : 0;
    vector<vector<bool>> visited(m, vector<bool>(n, false));
    priority_queue<Point, vector<Point>, TComp> pq;
    pq.push(Point(0, 0, A[0] + B[0]));
    visited[0][0] = true;
    Point pt;
    for (int i = 0; i < k; i++)
    {
        pt = pq.top();
        pq.pop();
        if (pt.ia + 1 < m && visited[pt.ia + 1][pt.ib] == false)
        {
            int sum = A[pt.ia + 1] + B[pt.ib];
            pq.push(Point(pt.ia + 1, pt.ib, sum));
            visited[pt.ia + 1][pt.ib] = true;
        }
        if (pt.ib + 1 < n && visited[pt.ia][pt.ib + 1] == false)
        {
            int sum = A[pt.ia] + B[pt.ib + 1];
            pq.push(Point(pt.ia, pt.ib + 1, sum));
            visited[pt.ia][pt.ib + 1] = true;
        }
    }
    return pt.sum;
}
```

Delete Digits

Idea:

1. Put the pair `A[i], A[i+1] (A[i] > A[i + 1])` into a heap. Priority: smaller `i` first; larger `A[i]` first.
2. Loop and pop the pairs from the heap, at most pop `k` times. Make `A[i] = '#'`, and then put a new valid pair `A[prev], A[i + 1] (A[prev] > A[i + 1])` into the heap.
3. If the heap is empty but `k > 0`, from the end to start, make the `A[i]` to `#` if `A[i]` is not `#`.

4. From the beginning copy the non-`#` elements out.

```cpp
class Node
{
public:
    Node() {}
    Node(int id1, int id2, int val1, int val2)
        : id1(id1), id2(id2), val1(val1), val2(val2) {}
    int id1, id2, val1, val2;
};
struct TComp
{
    bool operator()(const Node& a, const Node& b) const
    {
        if (a.id1 == b.id1)
            return a.val1 < b.val1;
        else
            return a.id1 > b.id1;
    }
};
int find_previous(string& A, int id)
{
    int i = id - 1;
    while (i >= 0 && A[i] == '#')
        i--;
    return i;
}
string DeleteDigits(string A, int k)
{
    int n = A.size();
    if (n == 0)
        return "";
    priority_queue<Node, vector<Node>, TComp> pq;
    for (int i = 0; i < n - 1; i++)
    {
        if (A[i] > A[i + 1])
            pq.push(Node(i, i + 1, A[i], A[i + 1]));
    }
    while (!pq.empty() && k > 0)
    {
        Node x = pq.top();
        pq.pop();
        k--;
        A[x.id1] = '#';
        int prev = find_previous(A, x.id1);
        if (prev != -1 && A[prev] > x.val2)
            pq.push(Node(prev, x.id2, A[prev], x.val2));
    }
    for (int i = n - 1; i >= 0 && k > 0; i--)
    {
        if (A[i] != '#')
        {
            A[i] = '#';
            k--;
        }
    }
    string rst;
    for (int i = 0; i < n; i++)
    {
        if (A[i] != '#' && (rst.size() > 0 || A[i] != '0'))
            rst += A[i];
    }
```

```
    return rst;
}
```

Contains Duplicate II

Idea: Use `unordered_set`.

```
bool containsNearbyDuplicate(vector<int>& nums, int k)
{
    int n = nums.size();
    unordered_set<int> S;
    for (int i = 0; i < n; i++)
    {
        if (i > k)
            S.erase(nums[i - k - 1]);
        if (S.count(nums[i]))
            return true;
        S.insert(nums[i]);
    }
    return false;
}
```

Contains Duplicate III

Idea: Use `multiset` and `queue`.

```
bool containsNearbyAlmostDuplicate(vector<int>& nums, int k, int t)
{
    if (nums.size() < 2 || k == 0)
        return false;
    queue<int> Q;
    multiset<long> S;
    for (int i = 0; i < nums.size(); i++)
    {
        if (S.size() > k)
        {
            int num = Q.front();
            Q.pop();
            S.erase(S.find(num));
        }
        auto it = S.lower_bound((long)nums[i] - (long)t);
        if (it == S.end() || *it > (long)nums[i] + (long)t)
        {
            Q.push(nums[i]);
            S.insert(nums[i]);
        }
        else
            return true;
    }
    return false;
}
```

Simplify Path

Idea: Use stack.

```
string simplifyPath(string path)
{
    int n = path.size();
    if (n == 0)
        return "";
    replace(path.begin(), path.end(), '/', ' ');
    stringstream ss(path);
    stack<string> S;
    string str;
    while (ss >> str)
    {
        if (str != "." && str != "..")
            S.push(str);
        else if (str == ".." && !S.empty())
            S.pop();
    }
    string rst;
    if (S.empty())
        return "/";
    while (!S.empty())
    {
        string temp = "/" + S.top();
        rst = temp + rst;
        S.pop();
    }
    return rst;
}
```

Evaluate Reverse Polish Notation

Idea: 逆波兰表示法

```cpp
int evalRPN(vector<string>& tokens)
{
    int n = tokens.size();
    if (n == 0)
        return 0;
    stack<int> S;
    int a, b;
    for (int i = 0; i < n; i++)
    {
        string tmp = tokens[i];
        if (tmp == "+")
        {
            a = S.top(); S.pop();
            b = S.top(); S.pop();
            S.push(b + a);
        }
        else if (tmp == "-")
        {
            a = S.top(); S.pop();
            b = S.top(); S.pop();
            S.push(b - a);
        }
        else if (tmp == "*")
        {
            a = S.top(); S.pop();
            b = S.top(); S.pop();
            S.push(b * a);
        }
        else if (tmp == "/")
        {
            a = S.top(); S.pop();
            b = S.top(); S.pop();
            S.push(b / a);
        }
        else
        {
            S.push(stoi(tmp));
        }
    }
    return S.top();
}
```

Basic Calculator / Basic Calculator II

Idea: Reverse Polish Notation.

1. Each operator has its priority.
2. If encounter number, loop until get the whole number, then push it into `S1`.
3. If encounter `'('`, directly push it into `S2`.
4. If encounter `')'`, loop until meet `'('`, and all the operators within are calculated.
5. If encounter an operator, pop up all the operators in `S2` whose priority `>=` current operator, and calculate the result. Then push the current operator to `S2`.
6. At last, pop up all the operators in `S2` and calculate.

```cpp
int eval(int x1, int x2, char op)
{
    int rst = 0;
    switch (op)
    {
        case '+': rst = x1 + x2; break;
        case '-': rst = x1 - x2; break;
        case '*': rst = x1 * x2; break;
        case '/': rst = x1 / x2; break;
    }
    return rst;
}
void helper(stack<int>& S1, stack<int>& S2)
{
    char op = S2.top(); S2.pop();
    int x2= S1.top(); S1.pop();
    int x1 = S1.top(); S1.pop();
    S1.push(eval(x1, x2, op));
}
int calculate(string s) {
    int n = s.size();
    if (n == 0)
        return 0;
    int Pr[256] = {0};
    Pr['+'] = Pr['-'] = 1;
    Pr['*'] = Pr['/'] = 2;
    unordered_set<char> opSet {'+', '-', '*', '/', '(', ')'};
    stack<int> S1, S2;
    int i = 0, val = 0;
    while (i < n)
    {
        char c = s[i];
        if (c <= '9' && c >= '0')
        {
            while (i < n && s[i] >= '0' && s[i] <= '9')
            {
                val = val * 10 + s[i] - '0';
                i++;
            }
            S1.push(val);
            val = 0;
        }
        else if (opSet.count(c))
        {
            if (c == '(')
                S2.push(c);
            else if (c == ')')
            {
                while (!S2.empty())
                {
                    if (S2.top() == '(')
                    {
                        S2.pop();
                        break;
                    }
                    helper(S1, S2);
                }
            }
            else
            {
```

```
                while (!S2.empty() && Pr[S2.top()] >= Pr[c])
                    helper(S1, S2);
                S2.push(c);
            }
            i++;
        }
        else
            i++;
    }
    while (!S2.empty())
        helper(S1, S2);
    return S1.top();
}
```

Math & Bit-Manipulation

Gray Code (Classic)

Idea: 格雷码：每位只差一个bit位的码。
对于 `n` 位的Gray Code，其等于用 `n - 1` 位的Gray Code通过镜像，并对镜像部分最高位填 `1`，并最后和原 `n - 1` 位格雷码合并的结果。注： `n == 0` 时，Gray Code为 `0`.

```
vector<int> grayCode(int n)
{
    if (n < 0)
        return vector<int>(0);
    if (n == 0)
        return vector<int>(1, 0);
    vector<int> rst = { 0, 1 };
    for (int i = 1; i < n; i++)
    {
        vector<int> mirror = rst;
        reverse(mirror.begin(), mirror.end());
        for (int k = 0; k < mirror.size(); k++)
        {
            mirror[k] |= (1 << i);
        }
        rst.insert(rst.end(), mirror.begin(), mirror.end());
    }
    return rst;
}
```

Plus One

Idea: Note that at current digit, the function is `temp = digits[i] + carry` and initial `carry` is 1.

```cpp
vector<int> plusOne(vector<int>& digits)
{
    int n = digits.size();
    if (n == 0)
        return digits;
    int carry = 1;
    for (int i = n - 1; i >= 0; i--)
    {
        int d = digits[i];
        int temp = d + carry;
        carry = temp / 10;
        digits[i] = temp % 10;
    }
    if (carry == 1)
    {
        digits[0] = 1;
        digits.push_back(0);
    }
    return digits;
}
```

A + B Problem

Idea: `a + b = (a ^ b) + ((a & b) << 1)`, `a ^ b`被称为不进位加法，`(a & b) << 1`是进位。所以迭代的把`a`转成`a ^ b`、`b`转成`(a & b) << 1`,即可计算出结果。循环截止的条件是`b == 0`.为何`b`转为`(a & b) << 1`会最终变为0呢？因为每次左移，最右边都会多出一个0，而上次最低位的0和别的数`&`不会产生1，所以0会越来越多。

```cpp
int aplusb(int a, int b)
{
    while (b != 0)
    {
        int a1 = a ^ b;
        int b1 = (a & b) << 1;
        a = a1;
        b = b1;
    }
    return a;
}
```

Add Binary

```cpp
string addBinary(string& a, string& b)
{
    int n1 = a.size(), n2 = b.size();
    if (n1 == 0)
        return b;
    if (n2 == 0)
        return a;
    string rst;
    int carry = 0;
    for (int i = 0; i < max(n1, n2); i++)
    {
        int d1 = n1 - i - 1 >= 0 ? a[n1 - i - 1] - '0' : 0,
            d2 = n2 - i - 1 >= 0 ? b[n2 - i - 1] - '0' : 0;
        int temp = d1 + d2 + carry;
        rst.push_back(temp % 2 + '0');
        carry = temp / 2;
    }
    if (carry == 1)
        rst.push_back('1');
    reverse(rst.begin(), rst.end());
    return rst;
}
```

Count 1 in Binary

Idea:

1. `num & num - 1` will cause the number of ones reduce by 1, until the result being 0.

```cpp
int countOnes(int num)
{
    int count = 0;
    while (num)
    {
        num = num & (num - 1);
        count++;
    }
    return count;
}
```

2. Hamming Window

```cpp
int countOnes(int num)
{
    int x[5] = { 0x55555555, 0x33333333, 0x0F0F0F0F, 0x00FF00FF, 0x0000FFFF };
    for (int i = 0; i < 5; i++)
    {
        num = (num & x[i]) + ((num >> (1 << i)) & x[i]);
    }
    return num;
}
```

Pow(x, n)

```cpp
double myPow(double x, int n)
{
    int m = abs(n);
    double rst = 1.0;
    while (m > 0)
    {
        if (m & 1)
            rst *= x;
        x = x * x;
        m >>= 1;
    }
    if (n < 0)
        rst = 1.0 / rst;
    return rst;
}
```

Update Bits

Idea: Note that if all bits need to be set, just return `m`.

```cpp
int updateBits(int n, int m, int i, int j)
{
    int ii = max(i, j), jj = min(i, j);
    if (ii == 31 && jj == 0)
        return m;
    int mask = (1 << (ii - jj + 1)) - 1;
    mask <<= jj;
    return ((~mask) & n) | (m << jj);
}
```

Tailing Zeros

Idea: calculate how many `5`s in `n!`. `rst = floor(n / 5) + floor(n / 25) + ...`

```cpp
long long trailingZeros(long long n)
{
    long long count = 0;
    while (n)
    {
        count += n / 5;
        n /= 5;
    }
    return count;
}
```

Number of Digit One

Idea: 分别判断个、十、百...位为1的数量，比如`n == 3141592`时，我们想判断百位为1的数字个数，我们设`m = 100`，然后`a = n / m, b = n % m`，则`a`为31415，`b`为92. 进一步，我们把`a`表示为`|x d|`，其中`d`为其个位数，`x`为高位。`d`分三种情况：

1. `d > 1`，则其完整的包含了`|x 1|00 ~ |x 1|99`，高位取值开始为0，故有`x + 1`个选择，每个选择包含100个数字；`=> (x + 1) * 100`
2. `d == 1`，则其完整的包含了`|x-1 1|00 ~ |x-1 1|99`，故有`x`个选择，每个选择包含100个数字；同时还有`|x 1|00 ~ |x 1|b`这些结果；`=> x * 100 + b + 1`

3. `d == 0`, 则其完整的包含了 `|x-1 1|00 ~ |x-1 1|99`, 故有 `x` 个选择, 每个选择包含100个数字； `=> x * 100`

```
31415|00    31415|92               if n is 3141592
31414|00    31414|99
31413|00    31413|99
31412|00    31412|99
31411|00    31411|99 ---- 100个      if n is 31411xx, 此行便取不满
31410|00    31410|99 ---- 0个        if n is 31410xx, 也只能从31401|00处开始考虑了
31401|00    31401|99 ---- 100个
31391|00    31391|99 ---- 100个
... ... ... ... ...
00011|00    00011|99 ---- 100个
00001|00    00001|99 ---- 100个
```

```
int countDigitOne(int n)
{
    if (n <= 0)
        return 0;
    long long rst = 0;
    for (long long m = 1; m <= n; m *= 10)
    {
        long long a = n / m;
        long long b = n % m;
        if (a % 10 >= 2)
            rst += (a / 10 + 1) * m;
        else if (a % 10 == 1)
            rst += a / 10 * m + b + 1;
        else
            rst += a / 10 * m;
    }
    return rst;
}
```

Palindrome Number

Idea: Note: 求比 `x` 小的最大的 `10^n`, 要用除法，不可用乘法，否则可能溢出。

```
int helper(int x)
{
    int y = 1;
    while (x / 10)
    {
        x /= 10;
        y *= 10;
    }
    return y;
}
bool isPalindrome(int x)
{
    if (x < 0)
        return false;
    if (x == 0)
        return true;
    int y = helper(x);
    while (x > 0)
    {
        int a = x / y;
        int b = x % 10;
        if (a != b)
            return false;
        x -= a * y;
        x /= 10;
        y /= 100;
    }
    return true;
}
```

Single Number

Idea: `a ^ a = 0, 0 ^ a = a`.

```
int singleNumber(vector<int>& nums)
{
    int n = nums.size();
    if (n == 0)
        return -1;
    int x = nums[0];
    for (int i = 1; i < n; i++)
        x ^= nums[i];
    return x;
}
```

Single Number II

Idea: Calculate the sum of all numbers at each bit. At each bit, the sum should be `3p + m`, where `p` is an integer, `m` is the bit number of the unique number at the bit. `(3p + m) % 3 => m`.

```cpp
int singleNumber(vector<int>& nums)
{
    vector<int> x(32, 0);
    for (int i = 0; i < nums.size(); i++)
    {
        for (int k = 0; k < 32; k++)
        {
            x[k] += ((nums[i] >> k) & 1);
        }
    }
    int rst = 0;
    for (int k = 0; k < 32; k++)
    {
        int y = x[k] % 3;
        rst |= (y << k);
    }
    return rst;
}
```

Single Number III

Idea: first calculate the `xor` of all numbers, its result `x` should be equal the `xor` of the two numbers. Then, we find the last 1 bit of `x` by `x ^= -x`. For example, if `x` is `101100`, the result will be `000100`. At the bit with `1`, the two numbers must be different. So `nums[i] & lastBit`'s result must be different for the two numbers. See code below.

```cpp
vector<int> singleNumber(vector<int>& nums)
{
    int n = nums.size();
    if (n == 0)
        return vector<int>(0);
    int x = 0;
    for (int i = 0; i < n; i++)
        x ^= nums[i];
    x &= -x;
    vector<int> rst = {0, 0};
    for (int i = 0; i < n; i++)
    {
        if (nums[i] & x)
            rst[0] ^= nums[i];
        else
            rst[1] ^= nums[i];
    }
    return rst;
}
```

Missing Number

Idea: suppose the given array `A`'s size is `n`, then it should have been `n + 1` with the missing number. So we `push_back(-1)` to `A`, and `-1` represents the missing number. Then for each `x` in `A`, if its value is not equal to its index, then swap it to its value position, so that after the loop, every number `x` will be located in `A[x]`, except the missing one, where `-1` locates at.

```cpp
int missingNumber(vector<int>& nums)
{
    int n = nums.size();
    nums.push_back(-1);
    for (int i = 0; i < nums.size(); i++)
    {
        while (nums[i] != i && nums[i] != -1)
            swap(nums[i], nums[nums[i]]);
    }
    int i = 0;
    while (i < nums.size() && nums[i] != -1) i++;
    return i;
}
```

Binary Representation

Idea: Note that we should use `double y = atof(s)` not `float y` to get the value.

```
string to_binary_integer(string& n)
{
    if (n.empty())
        return n;
    int x = atoi(n.c_str());
    if (x == 0)
        return "0";
    int flag = x < 0 ? -1 : 1;
    x = abs(x);
    string rst;
    while (x)
    {
        int y = x % 2;
        rst += y + '0';
        x /= 2;
    }
    reverse(rst.begin(), rst.end());
    return flag < 0 ? "-" + rst : rst;
}
string to_binary_decimal(string& n)
{
    double x = atof(n.c_str());
    if (abs(x) < 1e-6)
        return "";
    string rst;
    while (x > 1e-6)
    {
        double y = x * 2.0;
        if (y >= 1.0)
        {
            rst += '1';
            x = y - 1.0;
        }
        else
        {
            rst += '0';
            x = y;
        }
        if (rst.size() == 32)
            return "ERROR";
    }
    return rst;
}

string binaryRepresentation(string n)
{
    size_t found = n.find('.');
    string n1 = n.substr(0, found);
    string bn1 = to_binary_integer(n1);
    string bn2 = "";
    if (found != string::npos)
    {
        string n2 = "0." + n.substr(found + 1);
        bn2 = to_binary_decimal(n2);
        if (bn2 == "ERROR")
            return "ERROR";
    }
    if (bn2 == "")
        return bn1;
    else
```

```
61.         return bn1 + "." + bn2;
62. }
```

Fast Power

Idea: make `a^n % b` to `a^(n / 2) % b`.

```
1.  int fastPower(int a, int b, int n)
2.  {
3.      if (n == 1)
4.          return a % b;
5.      if (n == 0)
6.          return 1 % b;
7.      long long p = fastPower(a, b, n / 2);
8.      if (n % 2 == 1)
9.          return (p * ((p * (a % b)) % b)) % b;
10.     else
11.         return (p * p) % b;
12. }
```

Fraction to Recurring Decimal

Idea: Note that the cases:

1. `d == 0`.
2. `n == 0`.
3. `n` and `d` are different signs.

After that, we need to care about whether `n / d` will get a finite result. Use hash table to check.

```cpp
string fractionToDecimal(int numerator, int denominator)
{
    long long n = numerator, d = denominator;
    if (d == 0)
        return "";
    if (n == 0)
        return "0";
    long long sign = n ^ d;
    n = abs(n);
    d = abs(d);
    long long a = n / d;
    n = n % d;
    string sa = sign < 0 ? "-" : "";
    sa += to_string(a);
    if (n == 0)
        return sa;
    n *= 10;
    unordered_map<long long, int> hashMap;
    string tmp;
    int i = 0;
    bool flag = false;
    while (hashMap.count(n) == 0)
    {
        int x = n / d;
        tmp += x + '0';
        hashMap.insert(make_pair(n, i++));
        n = (n - x * d) * 10;
        if (n == 0)
        {
            flag = true;
            break;
        }
    }
    if (flag == false)
    {
        int index = hashMap.at(n);
        tmp.insert(index, "(");
        tmp += ')';
        return sa + "." + tmp;
    }
    else
        return sa + "." + tmp;
}
```

Divide Two Integers

Idea: Note the 2 cases which may cause overflow:

1. `divisor == 0`: return `INT_MAX`
2. `dividend == INT_MIN && divisor == -1`: return `INT_MAX` (because `abs(INT_MIN) = INT_MAX + 1`).

```cpp
int divide_helper(long long D, long long d)
{
    int sign = D < 0 ^ d < 0;
    D = labs(D);
    d = labs(d);
    long long orig_d = d;
    int i = 0;
    long long sum = 0;
    while (D >= d)
    {
        i = 1;
        while ((D >> 1) >= d)
        {
            d <<= 1;
            i <<= 1;
        }
        sum += i;
        D -= d;
        d = orig_d;
    }
    return sign == 1 ? -sum : sum;
}
int divide(int dividend, int divisor)
{
    if (divisor == 0 || (dividend == INT_MIN && divisor == -1))
        return INT_MAX;
    else
        return divide_helper(dividend, divisor);
}
```

Multiply Strings

Idea: `m`位数和`n`位数相乘，结果不会超过`(m + n + 1)`位。`string a, b`相乘，`a[i]`和`b[j]`相乘的结果放入`temprst[i + j + 2]`位（因为`a[m - 1]`和`b[n - 1]`相乘结果是放入`temprst[m + n]`的）。这样可以保证不用`reverse`.

```cpp
void multi(int m, string& s, int offset, string& rst)
{
    int n = s.size();
    int carry = 0;
    string temprst(rst.size(), '0');
    for (int i = n - 1; i >= 0; i--)
    {
        int x = s[i] - '0';
        int temp = m * x + carry;
        int a = temp % 10;
        temprst[i + offset + 2] = a + '0';
        carry = temp / 10;
    }
    if (carry != 0)
        temprst[-1 + offset + 2] = carry + '0';
    carry = 0;
    for (int i = temprst.size() - 1; i >= 0; i--)
    {
        int x = rst[i] - '0';
        int y = temprst[i] - '0';
        int z = carry + x + y;
        rst[i] = (z % 10) + '0';
        carry = z / 10;
    }
}
string multiply(string num1, string num2)
{
    // 123, 4567
    int n1 = num1.size(), n2 = num2.size();
    string rst(n1 + n2 + 1, '0');
    for (int i = n1 - 1; i >= 0; i--)
    {
        char c = num1[i];
        multi(c - '0', num2, i, rst);
    }
    int x = rst.find_first_not_of('0');
    if (x == string::npos)
        return "0";
    else
        return rst.substr(x);
}
```

String to Integer (atoi)

Idea: status machine. Note that when current value `> INT_MAX / 10` or current value `== INT_MAX / 10` and the value to be added is `> INT_MAX % 10`, it means, after this step, the value will be overflow.

```cpp
int myAtoi(string str)
{
    int n = str.size();
    if (n == 0)
        return 0;
    int status = 0, sign = 1, base = 0;
    for (int i = 0; i < n; i++)
    {
        char c = str[i];
        switch (status)
        {
        case 0:
            if (c == ' ')
                status = 0;
            else if (c == '+' || c == '-')
            {
                sign = c == '+' ? 1 : -1;
                status = 1;
            }
            else if (c <= '9' && c >= '0')
            {
                base = base * 10 + (c - '0');
                status = 1;
            }
            else
                status = 2;
            break;
        case 1:
            if (c <= '9' && c >= '0')
            {
                if (base > INT_MAX / 10 ||
                    base == INT_MAX / 10 && c - '0' > INT_MAX % 10)
                {
                    return sign == 1 ? INT_MAX : INT_MIN;
                }
                else
                    base = base * 10 + c - '0';
            }
            else
                status = 2;
            break;
        case 2:
            i = n;
            break;
        }
    }
    return base * sign;
}
```

Excel Sheet Column Title

Idea: 进制转换。

1. 找规律：用0-based进制表示：`A => 0, ..., Z => 25`，故当前字符为`'A' + res`，商需要减一；
2. 找规律：用1-based进制表示：`A => 1, ..., Z => 26`，故当前字符为`'A' + res - 1`，遇到`res`为0的情况，说明是`Z`，商要减一；
   `res`为余数

```cpp
// solution 1
string convertToTitle(int n)
{
    if (n <= 0)
        return "";
    n = n - 1; // 0 based
    string s;
    while (n >= 0)
    {
        int a = n / 26, b = n % 26;
        s += 'A' + b;
        n = a - 1;
    }
    reverse(s.begin(), s.end());
    return s;
}
// solution 2
string convertToTitle(int n)
{
    if (n <= 0)
        return "";
    string s;
    while (n)
    {
        int a = n / 26, b = n % 26;
        if (b == 0)
        {
            s += 'Z';
            n = a - 1;
        }
        else
        {
            s += 'A' + b - 1;
            n = a;
        }
    }
    reverse(s.begin(), s.end());
    return s;
}
```

Excel Sheet Column Number

Idea: Use reversing process of Excel Sheet Column Title.

```cpp
int titleToNumber(string s)
{
    int n = 0;
    for (int i = 0; i < s.size(); i++)
    {
        int x = s[i] - 'A' + 1;
        n = n * 26 + x;
    }
    return n;
}
```

Integer to Roman

Idea: Every digit converted to roman. The code can handle `num <= 3999` only.

```
string intToRoman(int num)
{
    string rst;
    int count = 0;
    char table[4][2] = { {'I', 'V'}, {'X', 'L'}, {'C', 'D'}, {'M', '0'} };
    int t = 1;
    while (t * 10 <= num)
    {
        t *= 10;
        count++;
    }
    while (count >= 0)
    {
        int x = (num / t) % 10;
        t /= 10;
        if (x == 4 || x == 9)
        {
            rst += table[count][0];
            rst += x == 4 ? table[count][1] : table[count + 1][0];
        }
        else
        {
            if (x > 4)
            {
                rst += table[count][1];
                x -= 5;
            }
            while (x--)
                rst += table[count][0];
        }
        count--;
    }
    return rst;
}
```

Roman to Integer

Idea: Compare `s[i]` and `s[i + 1]`, if `s[i] < s[i + 1]`, we add `-s[i]`; else we add `s[i]`. The last element can be directly added.

```cpp
int romanToInt(string s)
{
    int hash[26] = {0};
    hash['X' - 'A'] = 10;
    hash['I' - 'A'] = 1;
    hash['C' - 'A'] = 100;
    hash['L' - 'A'] = 50;
    hash['V' - 'A'] = 5;
    hash['D' - 'A'] = 500;
    hash['M' - 'A'] = 1000;
    int n = s.size(), sum = 0;
    for (int i = 0; i < n - 1; i++)
    {
        int a = hash[s[i] - 'A'], b = hash[s[i + 1] - 'A'];
        sum += a < b ? -a : a;
    }
    sum += hash[s[n - 1] - 'A'];
    return sum;
}
```

Integer to English Words

Idea: Every 1000 to convert.

```
string numberToWords(int num)
{
    if (num == 0)
        return "Zero";
    else
        return int_string(num).substr(1);
}
string int_string(int n)
{
    if (n >= 1000000000)
        return int_string(n / 1000000000) + " Billion"
            + int_string(n - n / 1000000000 * 1000000000);
    else if (n >= 1000000)
        return int_string(n / 1000000) + " Million"
            + int_string(n - n / 1000000 * 1000000);
    else if (n >= 1000)
        return int_string(n / 1000) + " Thousand"
            + int_string(n - n / 1000 * 1000);
    else if (n >= 100)
        return int_string(n / 100) + " Hundred"
            + int_string(n - n / 100 * 100);
    else if (n >= 20)
        return string(" ") + str_x0[n / 10]
            + int_string(n - n / 10 * 10);
    else if (n >= 10)
        return string(" ") + str_1x[n - 10];
    else if (n == 0)
        return "";
    else
        return string(" ") + str_x[n];
}
string str_x0[10] = {"", "", "Twenty", "Thirty", "Forty", "Fifty",
    "Sixty", "Seventy", "Eighty", "Ninety"};
string str_1x[10] = {"Ten", "Eleven", "Twelve", "Thirteen", "Fourteen", "Fifteen",
    "Sixteen", "Seventeen", "Eighteen", "Nineteen"};
string str_x[10] = {"", "One", "Two", "Three", "Four", "Five",
    "Six", "Seven", "Eight", "Nine"};
```

Count Primes

Idea: Define `dp[i]` as `i` being a non-prime (`1: non-prime, 0: prime`). Initialize `dp` as all `0`s. Loop `dp` from `i = 3` to `i == sqrt(n)` , if `dp[i] == 0` , then set all of its multiple non-prime.

```cpp
int countPrimes(int n)
{
    if (n <= 1)
        return 0;
    vector<int> dp(n + 1, 0);
    for (int i = 3; i <= sqrt(n); i += 2)
    {
        if (dp[i] == 1)
            continue;
        int k = i << 1;
        for (int j = i * i; j <= n ; j += k)
            dp[j] = 1;
    }
    int rst = n > 2;
    for (int i = 3; i < n; i += 2)
        rst += 1 - dp[i];
    return rst;
}
```

Reverse Bits

Idea: see code.

```cpp
uint32_t reverseBits(uint32_t n)
{
    n = (n >> 16) | (n << 16);
    n = ((n & 0xff00ff00) >> 8) | ((n & 0x00ff00ff) << 8);
    n = ((n & 0xf0f0f0f0) >> 4) | ((n & 0x0f0f0f0f) << 4);
    n = ((n & 0xcccccccc) >> 2) | ((n & 0x33333333) << 2);
    n = ((n & 0xaaaaaaaa) >> 1) | ((n & 0x55555555) << 1);
    return n;
}
```

Counting Bits

Idea: The number `n & (n - 1)` has one bit less than `n`. So if we want to get `NumBits(n)`, we only need to get `NumBits( n & (n - 1) )`.

```cpp
vector<int> countBits(int num)
{
    vector<int> dp(num + 1, 0);
    for (int i = 1; i <= num; i++)
        dp[i] = dp[i & (i - 1)] + 1;
    return dp;
}
```

Design

Peeking Iterator

Idea: Use a variable to denote whether the real iterator and the virtual iterator are the same(because implementing `peek` by `next` will cause the two iterators different.)

1. At the very beginning, the two iterators are the same.

2. If the two iterators are not the same:
    1. Once a `peek` called, we will return the value it stored last time
    2. Once a `next` called, we will return the value it stored last time, then set the two iterators the same.
    3. Once a `hasNext` called, it must return true.

3. If the two iterators are the same:
    1. Once a `peek` called, we will use `next` to get the value, return it and set the two iterators different
    2. Once a `next` called, just return `next`.
    3. Once a `hasNext` called, just return `hasNext`.

```cpp
// Below is the interface for Iterator, which is already defined for you.
// **DO NOT** modify the interface for Iterator.
class Iterator {
    struct Data;
    Data* data;
public:
    Iterator(const vector<int>& nums);
    Iterator(const Iterator& iter);
    virtual ~Iterator();
    // Returns the next element in the iteration.
    int next();
    // Returns true if the iteration has more elements.
    bool hasNext() const;
};


class PeekingIterator : public Iterator {
public:
    PeekingIterator(const vector<int>& nums) : Iterator(nums) {
        // Initialize any member here.
        // **DO NOT** save a copy of nums and manipulate it directly.
        // You should only use the Iterator interface methods.
        isSame = true;
    }

    // Returns the next element in the iteration without advancing the iterator.
    int peek() {
        if (isSame)
        {
            value = Iterator::next();
            isSame = false;
            return value;
        }
        else
            return value;
    }

    // hasNext() and next() should behave the same as in the Iterator interface.
    // Override them if needed.
    int next() {
        if (isSame)
            return Iterator::next();
        else
        {
            isSame = true;
            return value;
        }
    }

    bool hasNext() const {
        if (isSame)
            return Iterator::hasNext();
        else
            return true;
    }

    bool isSame;
    int value;
};
```

Binary Search Tree Iterator

Idea: In-order traversal in iterative version.

```
class BSTIterator
{
public:
    BSTIterator(TreeNode *root)
    {
        TreeNode *p = root;
        while (p)
        {
            S.push(p);
            p = p->left;
        }
    }

    /** @return whether we have a next smallest number */
    bool hasNext()
    {
        return !S.empty();
    }

    /** @return the next smallest number */
    int next()
    {
        if (!hasNext())
            return INT_MAX;
        TreeNode *p = S.top();
        S.pop();
        int rst = p->val;
        p = p->right;
        while (p)
        {
            S.push(p);
            p = p->left;
        }
        return rst;
    }

    stack<TreeNode*> S;
};
```

Min Stack

Idea: two stacks.

```
class MinStack
{
public:
    void push(int x)
    {
        S1.push(x);
        if (S2.empty() || x <= S2.top())
            S2.push(x);
    }
    void pop()
    {
        if (S1.empty())
            return;
        int x1 = S1.top();
        S1.pop();
        int x2 = S2.top();
        if (x2 >= x1)
            S2.pop();
    }
    int top()
    {
        return S1.top();
    }
    int getMin()
    {
        return S2.top();
    }
    stack<int> S1, S2;
};
```

LRU Cache (Classic!!!)

Idea: Double linked list + Hash Map.

1. If a key does not exist, add it to the end of the linked list.
2. If the capacity overflows, delete the head.
3. If get a key, put the node to the end.

```
class LRUCache{
public:
    class Node
    {
    public:
        Node() : prev(NULL), next(NULL) {}
        Node(int v) : prev(NULL), next(NULL), val(v) {}
        Node *prev, *next;
        int val, key;
    };
    LRUCache(int capacity_) {
        capacity = capacity_;
        head = NULL;
        tail = NULL;
    }

    int get(int key)
    {
        auto it = hashmap.find(key);
        if (it == hashmap.end())
            return -1;
        Node* x = it->second;
        int rst = x->val;
        if (x == tail)
            return rst;
        if (x == head)
        {
            head = x->next;
            head->prev = NULL;
        }
        else
        {
            x->next->prev = x->prev;
            x->prev->next = x->next;
        }
        tail->next = x;
        x->next = NULL;
        x->prev = tail;
        tail = x;
        return rst;
    }

    void set(int key, int value)
    {
        auto it = hashmap.find(key);
        if (it == hashmap.end())
        {
            if (capacity == 0)
                return;
            Node* node = new Node(value);
            node->key = key;
            if (head == NULL)
            {
                head = node;
                tail = node;
            }
            else
            {
                tail->next = node;
                node->prev = tail;
```

```
                tail = node;
            }
            hashmap.insert({ key, node });
            if (hashmap.size() > capacity)
            {
                Node* temp = head;
                hashmap.erase(temp->key);
                head = temp->next;
                head->prev = NULL;
                delete temp;
            }
        }
        else
        {
            it->second->val = value;
            get(key);
        }
    }

    int capacity;
    unordered_map<int, Node*> hashmap;
    Node *head, *tail;
};
```