LeetCode Problems Summary by Tags

DFS

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`

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.

Path Sum (Classic)

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()`.

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.

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;
}
```

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)`

Construct Binary Tree from Inorder and Postorder Traversal

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

Clone Graph (Graph, Check Again!)

Idea: use `unordered_map<int,*> to store the cloned 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!)

Balanced Binary Tree

Idea: Recursive. `bool x = DFS(root, height)`.

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

Idea: Define `int rst = DFS(root, kx)` , where `kx` is the maxval from root to a node, `rst` is the max sum of the tree `root`.

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: Standard Level-order traversal.

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.

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`.

```
// 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);
}
```

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

```
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 element,
    // then we should not select the following same elements, since it will cause duplicate 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);
}
```

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。

```
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;
}
```

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`.

```
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`.

```
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);
}
```

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。

```
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. 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.
- 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.

Linked List

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

Merge Two Sorted Lists (Classic)

```
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

Array

Partition Array (Classic!!!)

Idea: This is the partition step in quick sort.

```cpp
void partition(const 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
}
```

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`.

Tailing Zeros

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

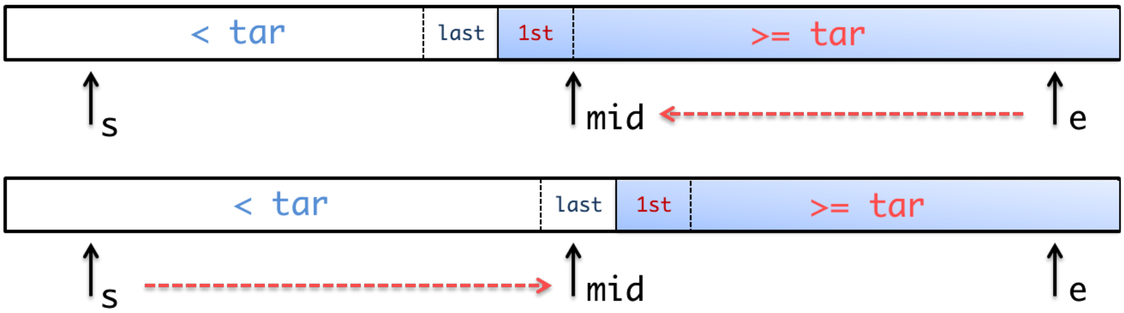
Binary Search

Note: Standard Templates

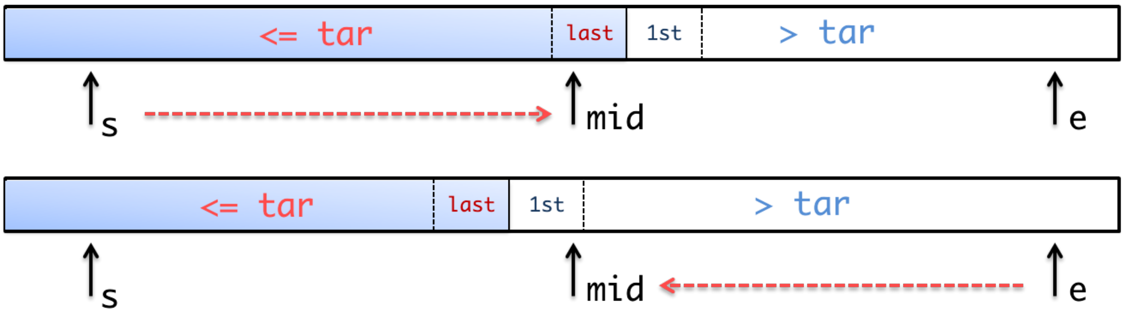
口诀：
一者大于等于TAR，中者大于等于TAR时尾前移；先看s后看e，都是大于等于。
后者小于等于TAR，中者小于等于TAR时头后移；先看e后看s，都是小于等于。
如果没有见到等于，找其对偶者加一减一处理。

Binary Search

1st >= tar || last < tar



last <= tar || 1st > tar



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 -1;
}

// 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;
}
```

Search for a Range

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

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.

Find Minimum in Rotated Sorted Array

Idea: binary search. After while loop stops, check: (1) `s == e`: `nums[s]` (2) `s + 1 == e`: `min(nums[s], nums[e])` (3) `s + 1 < e`: same as (2) (This case means the array is 0-rotated).

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. Check: (1) `s == e` (2) `s + 1 == e` (3) `s + 1 < e`: (3.1) `nums[s] == nums[mid] == nums[e]`: two directions O(n) scanning, find the minimum. if not found, it means all values are equal. (3.2) otherwise: `nums[s]`

Find the Duplicate Number

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

H-Index II **(Classic!!!)**

Sqrt(x)

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

Binary Tree

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.

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;
}
```

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.

```
vector<int> postorderTraversal(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)
        {
            if (crt->right)
                S.push(crt->right);
            S.push(crt);
            crt = crt->left;
        }
        // 2. Pop Step:
        TreeNode* tmp = S.top();
        S.pop();
        crt = tmp->right;
        // 3. Print Step:
        if (crt && !S.empty() && crt == S.top())
        {
            S.pop();
            S.push(tmp);
        }
        else
        {
            rst.push_back(tmp->val);
            crt = 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.

Dynamic Programming

Climbing Stairs

Idea: Coordinate DP

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

Idea: Coordinate DP

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]`.

Unique Binary Search Trees, Unique Binary Search Trees II

Idea: Interval DP. 1D `=>` 2D. `dp[i,j] = sum(dp[i,r-1]*dp[r+1,j]), r = i,...,j`.

Decode Ways

Idea: check the current and the previous characters, consider whether any(both) of them is `'0'`. There will be 4 cases to consider.

Perfect Squares

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

Longest Increasing Subsequence (Classic)

Idea: O(n): `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]`.

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

Triangle (Classic)

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

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.

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]`.

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], if p[j-1] == '*'.`

Note: This problem must be solved with a rolling array, otherwise MLE error.

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]`.

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])`.

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]}`.

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]`).

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]`!

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)`

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。

```
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

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