LeetCode 精选 TOP 面试题（3）

98. 验证二叉搜索树

思路

(深度优先遍历) $O(n)$

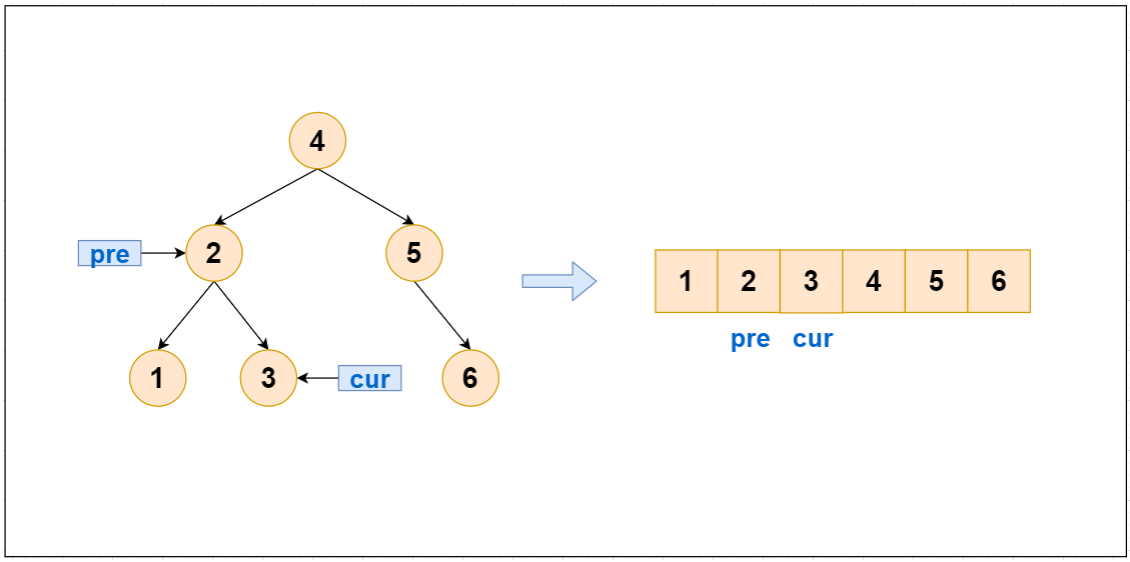
二叉搜索树是一种节点值之间具有一定数量级次序的二叉树，对于树中每个节点：

- 若其左子树存在，则其左子树中每个节点的值都**小于**该节点值；
- 若其右子树存在，则其右子树中每个节点的值都**大于**该节点值。

我们知道二叉搜索树**「中序遍历」**得到的值构成的序列一定是升序的的。因此我们可以对二叉树进行中序遍历，判断当前节点是否大于中序遍历的前一个节点，如果大于，说明说明这个序列是升序的，整棵树是二叉搜索树，否则不是。

二叉树的中序遍历顺序为：左根右

图示：



过程：

- 1、我们定义一个节点变量`pre`用来记录中序遍历的前一个节点。
- 2、中序遍历二叉树，在遍历过程中判断当前节点是否大于中序遍历的前一个节点。如果**大于**不做任何处理，如果**小于等于**说明不满足二叉搜索树的性质，返回`false`。

细节：

- 1、`pre`节点的初始值要设置为`null`。
- 2、具体实现过程看代码。

时间复杂度分析：树中每个节点仅被遍历一遍，所以时间复杂度是 $O(n)$。

c++代码

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
```

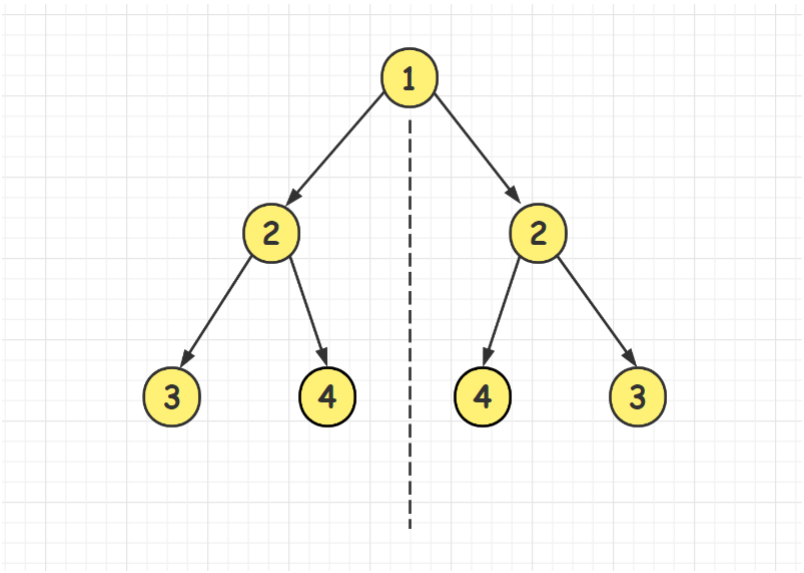
```
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x),
left(left), right(right) {}
 * };
 */
class Solution {
public:
    TreeNode* pre = nullptr;
    bool isValidBST(TreeNode* root) {
        if(!root)  return true;
        if(!isValidBST(root->left))  return false;
        if(pre && root->val <= pre->val)  return false;
        pre = root;
        return isValidBST(root->right);
    }
};
```

101. 对称二叉树

思路

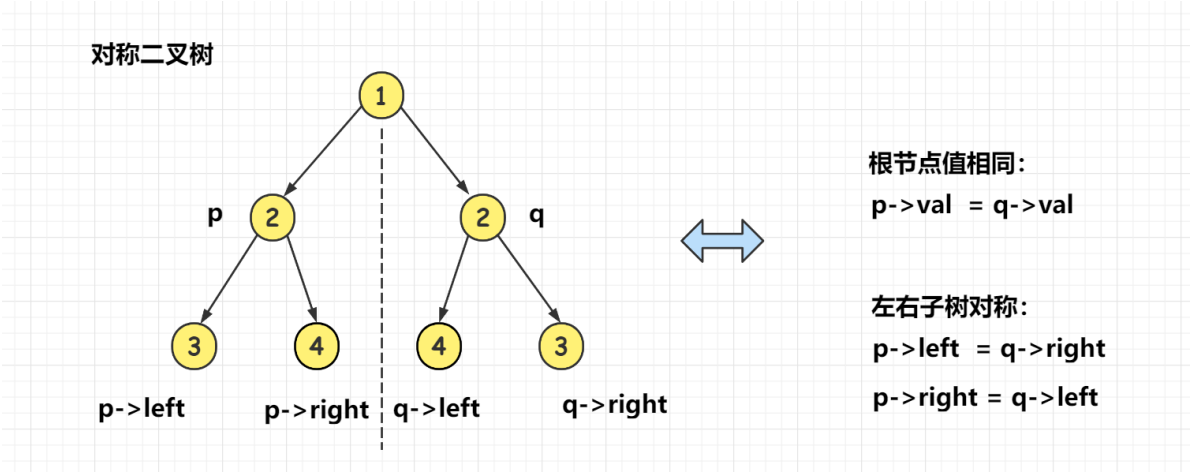
(递归) $O(n)$

样例如图



两个子树对称当且仅当：

1. 两个子树的根节点值相等；
2. 第一棵子树的左子树和第二棵子树的右子树对称，且第一棵子树的右子树和第二棵子树的左子树对称；



过程如下

1、我们定义两个指针 `p` 和 `q` ,让 `p` 和 `q` 指针一开始分别指向左子树和右子树。

2、同步移动这两个指针来遍历这棵树，每次检查当前 `p` 和 `q` 节点的值是否相等，如果相等再判断左右子树是否对称。

递归边界

- `p` 和 `q` 节点都为空时，左右子树都为空，返回 `true`
- `p` 和 `q` 节点只有一个为空时，左右子树不对称，返回 `false`
- `p` 和 `q` 节点值不相等，左右子树不对称，返回 `false`

时间复杂度分析： 从上到下每个节点仅被遍历一遍，所以时间复杂度是 $O(n)$。

c++代码

```
    /**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x),
left(left), right(right) {}
 * };
 */
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        if(!root) return true;
        return dfs(root->left, root->right);
    }

    bool dfs(TreeNode* p, TreeNode* q){
        if(!p && !q)  return true;
        if(!p || !q)  return false;
        if(p->val != q->val) return false;
        return dfs(p->left, q->right) && dfs(p->right, q->left);
    }
};
```

102. 二叉树的层序遍历

思路

(BFS) $O(n)$

我们从根节点开始按宽度优先的顺序遍历整棵树，每次先扩展左儿子，再扩展右儿子。

这样我们会：

1. 先扩展根节点；
2. 再依次扩展根节点的左右儿子，也就是从左到右扩展第二层节点；
3. 再依次从左到右扩展第三层节点；
4. 依次类推

然后在遍历过程中我们给每一层加一个结尾标记`NULL`,当我们访问到一层的结尾时，由于`BFS`的特点，我们刚好把下一层都加到了队列中。这个时候就可以给这层加上结尾标记`NULL`了,每次遍历到一层的结尾`NULL`时，就将这一层添加到结果中。

时间复杂度分析： 每个节点仅会被遍历一次，因此时间复杂度为$O(n)$。

c++代码

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x),
left(left), right(right) {}
 * };
 */
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> res;
        vector<int> path;
        queue<TreeNode*> q;
        q.push(root);
        q.push(nullptr);
        while(q.size()){
            auto t = q.front();
            q.pop();
            if(!t){
                if(path.empty()) break; //如果当前层没有元素，直接结束（防止进入死
循环）
                res.push_back(path);
                path.clear();
                q.push(nullptr);
            }else{
                path.push_back(t->val);
                if(t->left)  q.push(t->left);
                if(t->right) q.push(t->right);
            }
        }
```

```
        return res;
    }
};
```

103. 二叉树的锯齿形层序遍历

思路

(BFS) $O(n)$
我们从根节点开始按宽度优先的顺序遍历整棵树，每次先扩展左儿子，再扩展右儿子。

这样我们会：

1. 先扩展根节点；
2. 再依次扩展根节点的左右儿子，也就是从左到右扩展第二层节点；
3. 再依次从左到右扩展第三层节点；
4. 依次类推

然后在遍历过程中我们给每一层加一个结尾标记`NULL`,当我们访问到一层的结尾时，由于`BFS`的特点，我们刚好把下一层都加到了队列中。这个时候就可以给这层加上结尾标记`NULL`了。

再给每一层加一个标记，奇数行为从左到右，偶数行为从右到左。

c++代码

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x),
left(left), right(right) {}
 * };
 */
class Solution {
public:
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        vector<vector<int>> res;
        vector<int> path;
        queue<TreeNode*> q;
        bool flag = true;
        q.push(root);
        q.push(nullptr);
        while(q.size()){
            auto t = q.front();
            q.pop();
            if(!t){
                if(path.empty())  break;
                if(!flag) reverse(path.begin(), path.end());
                res.push_back(path);
                path.clear();
                q.push(nullptr);
                flag = !flag;
            }else{
                path.push_back(t->val);
```

```
                if(t->left)  q.push(t->left);
                if(t->right) q.push(t->right);
            }
        }
        return res;
    }
};
```

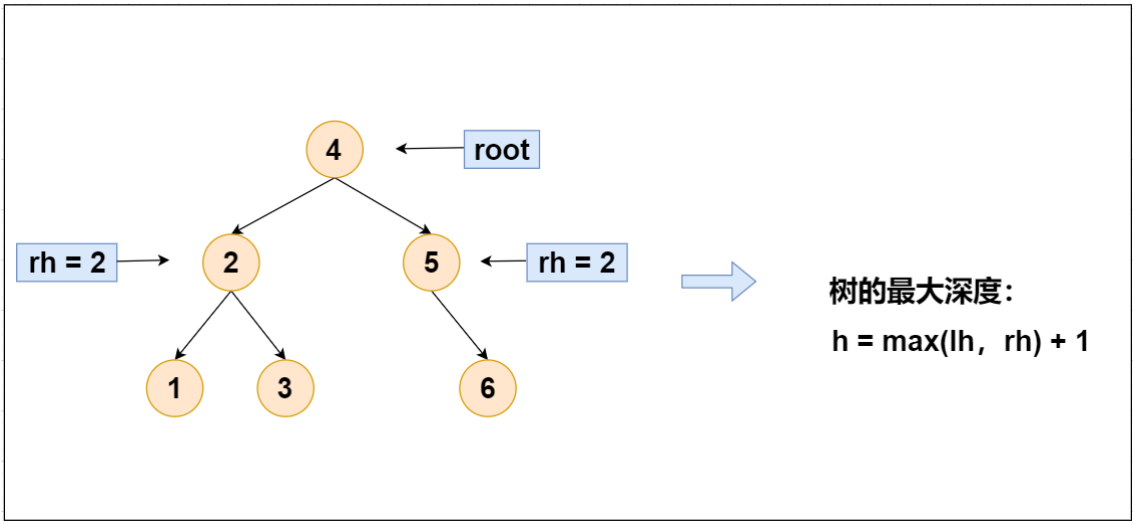
104. 二叉树的最大深度

思路

(递归) $O(n)$

当前树的最大深度等于左右子树的最大深度加`1`，也就是说如果我们知道了左子树和右子树的最大深度`lh`和`rh`，那么该二叉树的最大深度即为`max(lh,rh) + 1`

图示：



递归设计：

- 1、递归边界：当前节点为空时，树的深度为`0`
- 2、递归返回值：返回当前子树的深度，即`max(lh,rh) + 1`

时间复杂度分析：树中每个节点只被遍历一次，所以时间复杂度是 $O(n)$。

c++代码

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x),
left(left), right(right) {}
 * };
 */
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if(!root) return 0;
```

```
        int lh = maxDepth(root->left), rh = maxDepth(root->right);
        return max(lh, rh) + 1;
    }
};
```

105. 从前序与中序遍历序列构造二叉树

思路

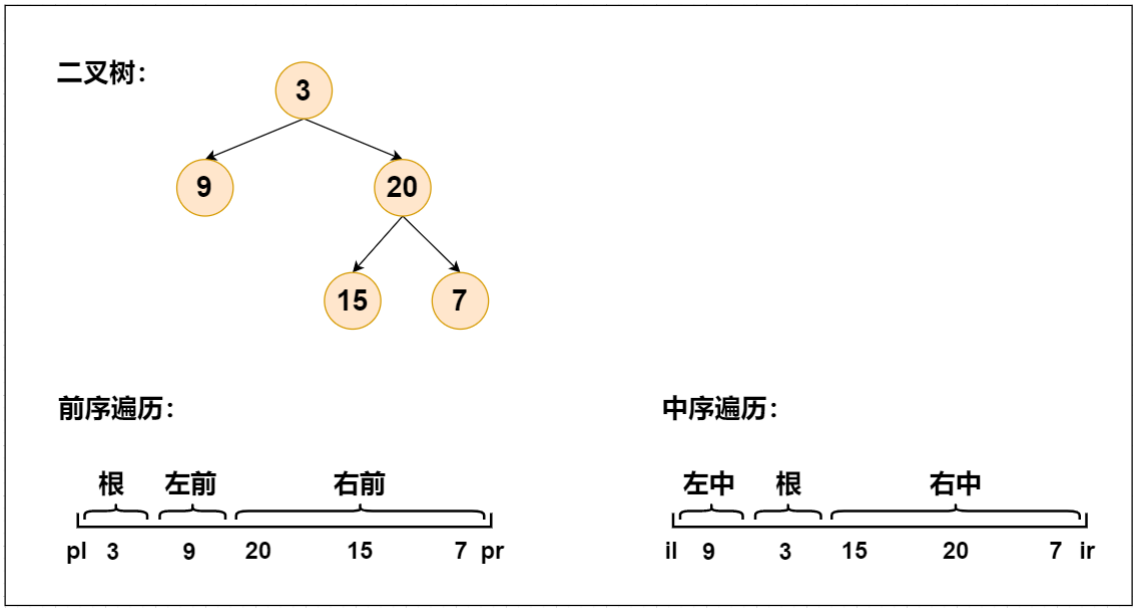
(递归)

二叉树：

- 二叉树前序遍历的顺序为：根左右
- 二叉树中序遍历的顺序为：左根右

我们递归建立整棵二叉树：先创建根节点，然后递归创建左右子树，并让指针指向两棵子树。

图示：



具体步骤如下：

- 1、先利用前序遍历找根节点，前序遍历的第一个数，就是根节点的值；
- 2、在中序遍历中找到根节点的位置 `pos`，则 `pos` 左边是左子树的中序遍历，右边是右子树的中序遍历；
- 3、假设左子树的中序遍历的长度是 `k`，则在前序遍历中，根节点后面的 `k` 个数，是左子树的前序遍历，剩下的数是右子树的前序遍历；
- 4、有了左右子树的前序遍历和中序遍历，我们可以先递归创建出根节点，然后再递归创建左右子树，再将这两颗子树接到根节点的左右位置；

细节1：如何在中序遍历中对根节点快速定位？

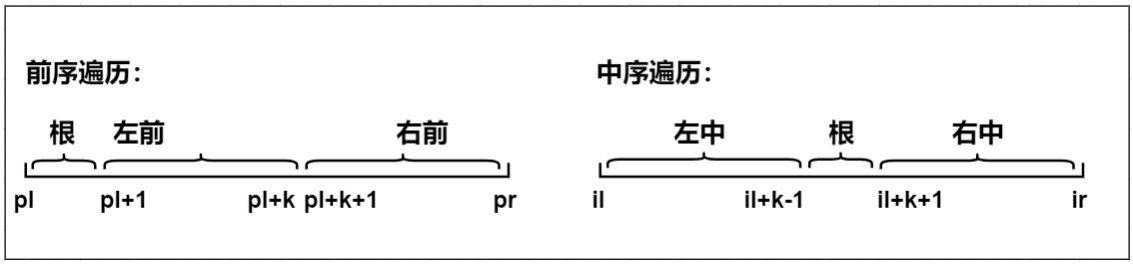
一种简单的方法是直接扫描整个中序遍历的结果并找出根节点，但这样做的时间复杂度较高。我们可以考虑使用哈希表来帮助我们快速地定位根节点。对于哈希映射中的每个键值对，键表示一个元素（节点的值），值表示其在中序遍历中的出现位置。

细节2：如何确定左右子树的前序遍历和中序遍历范围？

- 1、根据哈希表找到中序遍历的根节点位置，我们记作`pos`

- 2、用`pos-il`(`il`为中序遍历左端点)得到中序遍历的长度`k`，由于一棵树的前序遍历和中序遍历的长度相等，因此前序遍历的长度也为`k`。有了前序和中序遍历的长度，根据如上具体步骤`2`，`3`，我们就能很快确定左右子树的前序遍历和中序遍历范围。

  如图所示：



  `pl`,`pr`对应一棵子树的前序遍历区间的左右端点，`il`,`ir`对应一棵子树的中序遍历区间的左右端点。

时间复杂度分析： $O(n)$，其中 n 是树中的节点个数。

c++代码

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    unordered_map<int, int> hash;
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        int n = inorder.size();
        for(int i = 0; i < n; i++){
            hash[inorder[i]] = i;
        }
        return dfs(preorder, 0, n - 1, inorder, 0, n - 1);
    }
    TreeNode* dfs(vector<int>& pre, int pl, int pr, vector<int>& in, int il, int ir){
        if(pl > pr) return nullptr;
        int pos = hash[pre[pl]];
        int k = pos - il;
        TreeNode* root = new TreeNode(pre[pl]);
        root->left = dfs(pre, pl + 1, pl + k, in, il, il + k - 1);
        root->right = dfs(pre, pl + k + 1, pr, in, il + k + 1, ir);
        return root;
    }
};
```

108. 将有序数组转换为二叉搜索树

思路

(二叉搜索树，递归) $O(n)$

二叉搜索树的中序遍历为有序的，为左根右。

递归建立整棵二叉树：

- 1、每次以中点为根，以左半部分为左子树，右半部分为右子树。
- 2、先分别递归建立左子树和右子树，然后令根节点的指针分别指向两棵子树。

时间复杂度分析：一共建立 n 个节点，在递归函数中，建立每个节点的复杂度是 $O(1)$，所以总时间复杂度是 $O(n)$。

c++代码

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x),
left(left), right(right) {}
 * };
 */
class Solution {
public:
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        int n = nums.size();
        return dfs(nums, 0, n - 1);
    }
    TreeNode* dfs(vector<int>& nums, int l, int r){
        if(l > r) return nullptr;
        int k = (l + r) / 2;
        TreeNode* root = new TreeNode(nums[k]);
        root->left = dfs(nums, l, k - 1);
        root->right = dfs(nums, k + 1, r);
        return root;
    }
};
```

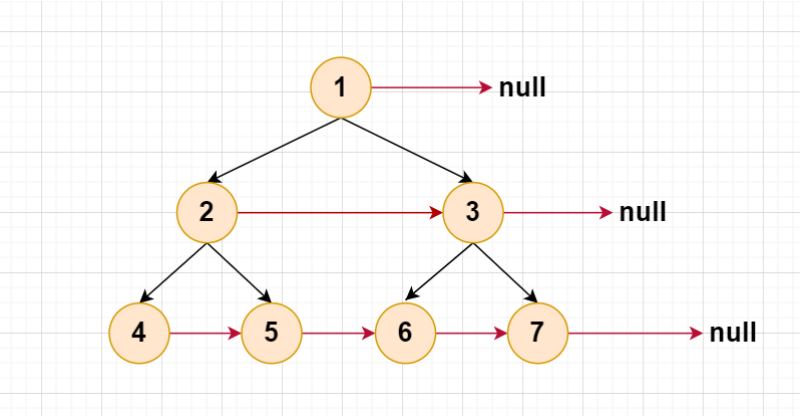
116. 填充每个节点的下一个右侧节点指针

思路

(BFS，树的遍历) $O(n)$

从根节点开始宽度优先遍历，每次遍历一层，遍历时按从左到右的顺序，对于每个节点，先让左儿子指向右儿子，最后让这一层最右侧的节点指向`NULL`。直到我们遍历到叶节点所在的层为止。

我们举个例子，**如图所示：**



对于`2`这个节点：

- 1、先让左儿子指向右儿子，即`2->left->next = 2->right`。
- 2、然后让右儿子指向下一个节点的左儿子，即`2->right->next = 2->next->left`。

具体实现过程如下：

1、记录根节点`source = root`，从根节点`root`开始遍历。

2、如果`root->lfet`存在，我们遍历该层的所有节点，每层从最左边节点开始，假设当前遍历的节点为`p`。

- 先让`p`的左儿子指向右儿子，即`p->left->next = p->right`。
- 如果`p->next`存在，我们就让`p`的右儿子指向下一个节点（即`p->next`节点）的左儿子。

3、每遍历完一层节点，让`root = root->left`。

4、最后返回`source`节点。

时间复杂度分析： 每个节点仅会遍历一次，所以总时间复杂度是 $O(n)$。

c++代码

```
/*
// Definition for a Node.
class Node {
public:
    int val;
    Node* left;
    Node* right;
    Node* next;

    Node() : val(0), left(NULL), right(NULL), next(NULL) {}

    Node(int _val) : val(_val), left(NULL), right(NULL), next(NULL) {}

    Node(int _val, Node* _left, Node* _right, Node* _next)
        : val(_val), left(_left), right(_right), next(_next) {}
};
*/

class Solution {
public:
    Node* connect(Node* root) {
        if(!root) return NULL;
        Node* source = root;
        while(root->left){
```

```
            for(Node* p = root; p; p = p->next){
                p->left->next = p->right;
                if(p->next)  p->right->next = p->next->left;
            }
            root = root->left;
        }
        return source;
    }
};
```

118. 杨辉三角

思路

(动态规划) $O(n^2)$

状态表示： `f[i][j]`表示第`i`行第`j`列应该填的数字。

状态计算： `f[i][j] = f[i - 1][j - 1] + f[i - 1][j]`

时间复杂度分析： $O(n^2)$。

c++代码

```
class Solution {
public:
    vector<vector<int>> generate(int n) {
        vector<vector<int>> f;
        for(int i = 0; i < n; i++){
            vector<int> line(i + 1);  //每行的元素个数为i + 1
            line[0] = line[i] = 1;    //每行首尾元素为1
            for(int j = 1; j < i; j++)
                line[j] = f[i - 1][j - 1] + f[i - 1][j];  //填写一行
            f.push_back(line);
        }
        return f;
    }
};
```

121. 买卖股票的最佳时机

思路

(数组) $O(n)$

- 1、当枚举到`i`时，`minv`维护的是`[0, i]`最小的价格，`price[i] - minv`是在当前点`i`买入的最大收益，
- 2、计算所有点的最大收益取最大值

时间复杂度 $O(n)$

c++代码

```cpp
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int res = 0, minv = INT_MAX;
        for(int i = 0; i < prices.size(); i++){
            minv = min(minv, prices[i]);
            res = max(res, prices[i] - minv);
        }
        return res;
    }
};
```

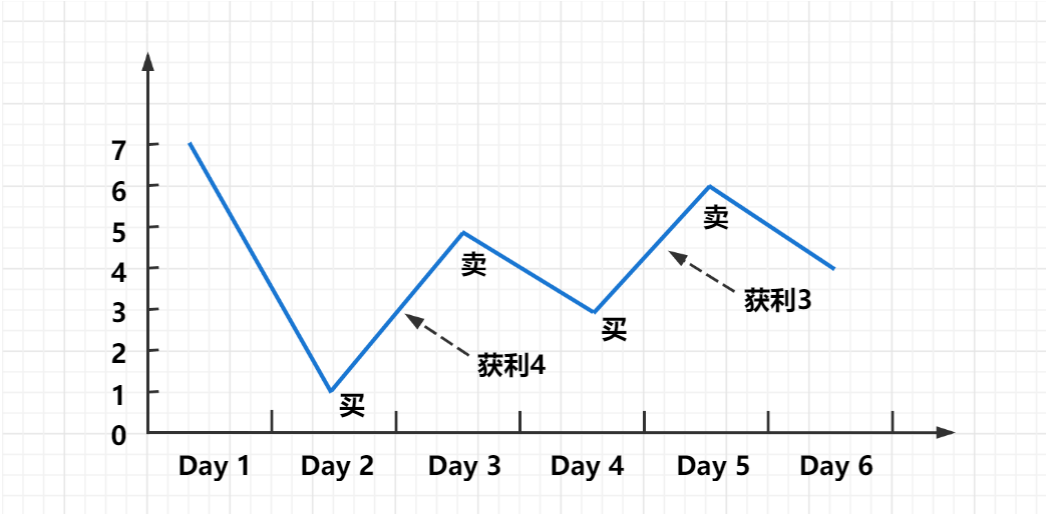
122. 买卖股票的最佳时机 II

思路

(贪心) $O(n)$

对于`[i,j]`这一区间，我们知道: `p[j] - p[i]= (p[i+1] - p[i]) + (p[i+2] - p[i+1])+ ... + (p[j] - p[j-1])` 。如果我们每个长度为 `1`的区间都取正数，这样我们得到的`p[j] - p[i]`价值一定是最大的。

因此我们考虑相邻两天的股票价格，如果后一天的股票价格大于前一天的，那么在进行买入卖出操作后，即可获利。而且这样在不考虑交易次数的前提下，这样的贪心一定能获得最大的利润。



具体过程如下:

- 1、枚举整个数组，如果发现`preprices[i + 1] > prices[i]`，我们就在第`i`天买入，第`i+1`天卖出，并将利润`preprices[i + 1] -prices[i]`记录到答案`res`中。
- 2、返回总利润`res`。

时间复杂度分析: $O(n)$，n是数组的长度。

c++代码

```cpp
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int res = 0;
        for(int i = 0 ; i + 1 < prices.size(); i++){
            if(prices[i + 1] > prices[i])
                res += prices[i + 1] - prices[i];
        }
        return res;
    }
};
```

124. 二叉树中的最大路径和

思路

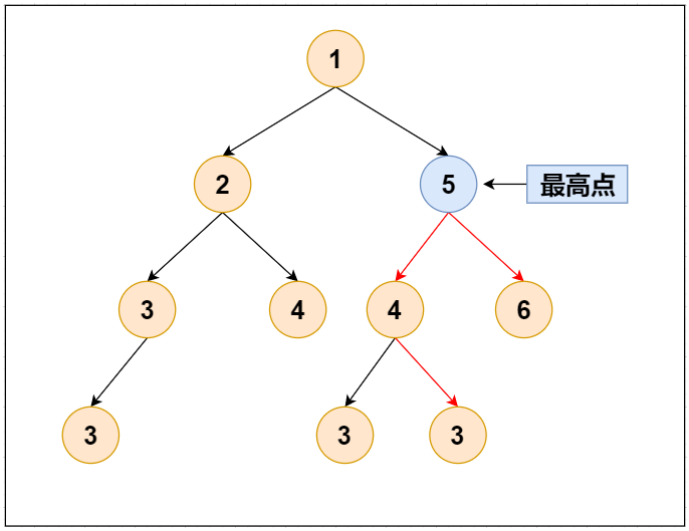
(递归，树的遍历) $O(n^2)$

路径

在这道题目中，路径是指从树中某个节点开始，沿着树中的边走，走到某个节点为止，路过的所有节点的集合。**路径的权值和是指路径中所有节点的权值的总和。**

对于一棵树，我们可以将其划分为很多的子树，如下图所示，虚线矩形围起来的子树。我们把这颗子树的蓝色节点称为该子树最高节点。用最高节点可以将整条路径分为两部分：从该节点向左子树延伸的路径，和从该节点向右子树延伸的部分。

如图所示：



我们可以递归遍历整棵树，递归时维护从每个子树从最高节点开始往下延伸的最大路径和。

- 对于每个子树的最高节点，递归计算完左右子树后，我们将左右子树维护的两条最大路径，和该点拼接起来，就可以得到以这个点为最高节点子树的最大路径。（这条路径一定是：**左子树路径->最高节点->右子树路径**）
- 然后维护从这个点往下延伸的最大路径：从左右子树的路径中选择权值大的一条延伸即可。（只能从**左右子树之间选一条路径**）

最后整颗树的最大路径和为：根节点值+左子树最大路径和+右子树最大路径和，即`left_max + right_max + root->val`

注意：

如果某条路径之和小于`0`，那么我们选择不走该条路径，因此其路径之和应和`0`之间取最大值。

时间复杂度分析： 每个节点仅会遍历一次，所以时间复杂度是 $O(n)$。

c++代码

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x),
left(left), right(right) {}
 * };
 */
class Solution {
public:
    int res = INT_MIN;
    int maxPathSum(TreeNode* root) {
        dfs(root);
        return res;
    }

    int dfs(TreeNode* root){
        if(!root)  return 0;
        int left = max(0, dfs(root->left)), right = max(0, dfs(root-
>right));
        res = max(res, root->val + left + right);
        return root->val + max(left, right);
    }
};
```

125. 验证回文串

思路

(线性扫描) $O(n)$

具体过程如下：

- 1、定义两个指针分别从前往后开始，往中间扫描。
- 2、每次迭代两个指针分别向中间靠近一步，靠近的过程中忽略除了字母和数字的其他字符。
- 3、然后判断两个指针所指的字符是否相等，如果不相等，说明不是回文串。
- 4、当两个指针相遇时，说明原字符串是回文串。

时间复杂度分析： 每个字符仅会被扫描一次，所以时间复杂度是 $O(n)$。

c++代码

```
class Solution {
public:
    bool check(char c){
        return c >= '0' && c <= '9' || c >= 'a' && c <= 'z' || c >= 'A' && c
<= 'Z';
    }

    bool isPalindrome(string s) {
```

```
        int l = 0, r = s.size() - 1;
        while(l < r){
            while(l < r && !check(s[l])) l++;  //左指针跳过其他字符
            while(l < r && !check(s[r])) r--;  //右指针跳过其他字符
            if(l < r && tolower(s[l]) != tolower(s[r])) return false;  //转为小写比较
            l++, r--;
        }
        return true;
    }
};
```

127. 单词接龙

思路

(最短路，BFS) $O(n^2L)$

我们对问题进行抽象：

将单词看做点，如果两个单词可以相互转化，则在相应的点之间连一条无向边。那问题就变成了求从起点到终点的最短路。

然后考虑如何建图，这里我们选择:

- 枚举所有单词对，然后判断是否可以通过改变一个字母相互转化，时间复杂度 $O(n^2L)$;

由于边权都相等，所以可以用BFS求最短路。

时间复杂度分析：

- 1、建图，通过上述分析可知，时间复杂度是 $O(n^2L)$;
- 2、求最短路用的是BFS，每个节点仅会遍历一次，每个点遍历时需要$O(L)$的计算量，所以时间复杂度是 $O(nL)$;

所以总时间复杂度是 $O(26nL2)$。

c++代码

```
class Solution {
public:
    int ladderLength(string beginWord, string endWord, vector<string>& wordList) {
        unordered_set<string> S;
        unordered_map<string, int> dist;
        queue<string> q;
        dist[beginWord] = 1;
        q.push(beginWord);
        for (auto word: wordList) S.insert(word);

        while (q.size()) {
            auto t = q.front();
            q.pop();
            string r = t;
            for (int i = 0; i < t.size(); i ++ ) {
                t = r;
                for (char j = 'a'; j <= 'z'; j ++ )
                    if (r[i] != j) {
                        t[i] = j;
                        if (S.count(t) && dist.count(t) == 0) {
```

```
                            dist[t] = dist[r] + 1;
                            if (t == endWord) return dist[t];
                            q.push(t);
                        }
                    }
                }
            }
            return 0;
        }
};
```

128. 最长连续序列

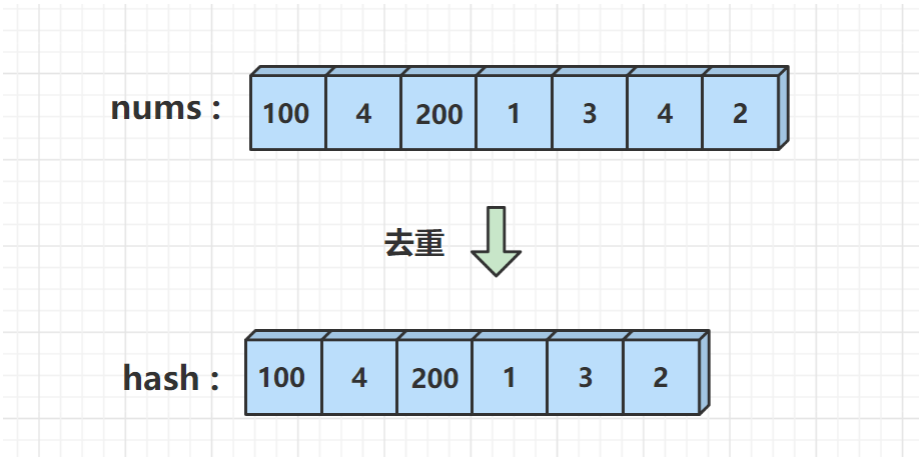
思路

(哈希) $O(n)$

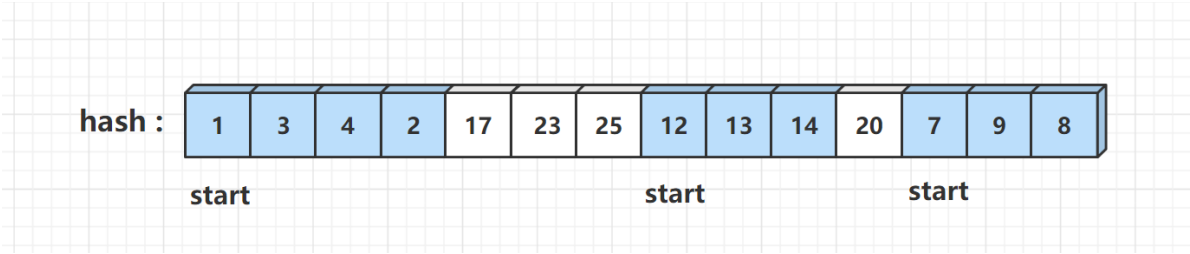
在一个未排序的整数数组 `nums` 中，找出最长的数字连续序列，朴素的做法是：枚举 `nums` 中的每一个数 `x`，并以 `x` 起点，在 `nums` 数组中查询 `x + 1`，`x + 2`，，，`x + y` 是否存在。假设查询到了 `x + y`，那么长度即为 `y - x + 1`，不断枚举更新答案即可。

如果每次查询一个数都要遍历一遍 `nums` 数组的话，时间复杂度为$O(n)$，其实我们可以用一个哈希表来存贮数组中的数，这样查询的时间就能优化为$O(1)$。

数组哈希去重



为了保证$O(n)$的时间复杂度，避免重复枚举一段序列，我们要从序列的起始数字向后枚举。也就是说如果有一个 `x`，`x+1`，`x+2`,,, `x+y` 的连续序列，我们只会以 `x` 为起点向后枚举，而不会从 `x+1`，`x+2`,,,向后枚举。



如何每次只枚举连续序列的起始数字 `x`？

其实只需要每次在哈希表中检查是否存在 `x - 1` 即可。如果 `x - 1` 存在，说明当前数 `x` 不是连续序列的起始数字，我们跳过这个数。

具体过程如下：

- 1、定义一个哈希表 `hash`，将 `nums` 数组中的数都放入哈希表中。

- 2、遍历哈希表`hash`，如果当前数`x`的前驱`x-1`不存在，我们就以当前数`x`为起点向后枚举。
- 3、假设最长枚举到了数`y`，那么连续序列长度即为`y-x+1`。
- 4、不断枚举更新答案。

时间复杂度分析： `while`循环最多执行n次，因此时间复杂度为$O(n)$。

c++代码

```
class Solution {
public:
    int longestConsecutive(vector<int>& nums) {
        unordered_set<int> hash;
        int res = 0;
        for(int x : nums) hash.insert(x);
        for(int x : hash){
            if(!hash.count(x - 1)){
                int y = x;
                while(hash.count(y + 1)) y++;
                res = max(res, y - x + 1);
            }
        }
        return res;
    }
};
```

130. 被围绕的区域

思路

(Flood Fill, 深度优先遍历) $O(n^2)$

边界上的`'O'`不会被包围，所有不在边界上且不与边界相连的`'O'`都会被攻占。因此我们可以逆向考虑，先统计出哪些区域不会被攻占，然后将其它区域都变成`'X'`即可。

具体过程如下：

- 1、从外层出发，`dfs`深度搜索，将不被包围的`'O'`变成`'#'`。
- 2、最后枚举整个数组，把`'O'`和`'X'`变成`'X'`，`'#'`变成`'O'`。

时间复杂度分析： $O(n^2)$。

c++代码

```
class Solution {
public:
    vector<vector<char>> board;  //全局变量
    int dx[4] = {-1, 0, 1, 0}, dy[4] = {0, 1, 0, -1};
    int n, m;
    void solve(vector<vector<char>>& _board) {
        board = _board;
        n = board.size(), m = board[0].size();
        if(!n) return ;

        for(int i = 0; i < n; i++){  // 左右边界
            if(board[i][0] == 'O')     dfs(i, 0);
            if(board[i][m - 1] == 'O') dfs(i, m - 1);
        }

```

```
        for(int i = 0; i < m; i++){ //上下边界
            if(board[0][i] == 'O')     dfs(0, i);
            if(board[n - 1][i] == 'O') dfs(n - 1, i);
        }

        for(int i = 0; i < n; i++)
            for(int j = 0; j < m; j++)
                if(board[i][j] == '#')  board[i][j] = 'O';
                else board[i][j] = 'X';
        _board =  board;
    }
    void dfs(int x, int y){
        board[x][y] = '#';
        for(int i = 0; i < 4; i++){
            int a = x + dx[i], b = y + dy[i];
            if(a >= 0 && a < n && b >= 0 && b < m && board[a][b] == 'O'){
                dfs(a, b);
            }
        }
    }
};


```

131. 分割回文串

思路

(动态规划 + dfs) $O(2^n * n)$

预处理

状态表示： `f[i][j]`表示字符串`s`在区间`[i, j]`的子串是否为一个回文串。`f[i][j]`有两种状态，如果是，则`f[i][j] = ture`，如果不是，则`f[i][j] = false`。

状态计算：

对于字符串`s`的区间子串`s[i,,,j]`，我们去判断`s[i]`和`s[j]`是否相等：

- 1、如果`s[i] == s[j]`，那么此时`f[i][j]`的状态就取决于`f[i + 1][j - 1]`，即`f[i][j] = f[i + 1][j - 1]`。
- 2、如果`s[i] != s[j]`，那么可以肯定`s[i,,,j]`一定不是回文串，即`f[i][j] = false`。

因此，**状态转移方程为：**

边界：

`i == j`时，`f[i][j] = true`。

`(j - i) <= 2`时，即子串`s[i,,,j]`三个或者两个字符长度时，`f[i][j] = true`

实现细节：

为了保证`f[i + 1][j - 1]`的状态先于`f[i][j]`被计算出来，我们需要先从小到大枚举`j`，再枚举`i`。

递归

递归函数设计：

```
void dfs(string& s, int u)
```

`s`代表字符串，`u`是当前回文串的起始位置。对于每一个`u`，都去枚举下一个回文串的最终位置`i`，若`[u,i]`组成的字符串是一个回文串，则记录`[u,i]`组成的回文串，并递归到下一层`i + 1`的位置。

边界：

当`u == s.size()`时，表示已经分割完整个字符串，将当前记录过的回文串链表加入到`res`中

时间复杂度分析： $O(2^n * n)$

c++代码

```
class Solution {
public:
    vector<vector<string>> res;
    vector<string> path;
    vector<vector<bool>> f;
    vector<vector<string>> partition(string s) {
        int n = s.size();
        f = vector<vector<bool>>(n, vector<bool>(n));
        for(int j = 0; j < n; j++)
            for(int i = 0; i <= j; i++){
                if(i == j) f[i][j] = true;
                else if(s[i] == s[j]){
                    if(j - i <= 2 || f[i + 1][j - 1]) f[i][j] = true;
                }
            }
        dfs(s, 0);
        return res;
    }
    void dfs(string& s, int u){
        if(u == s.size()){
            res.push_back(path);
            return;
        }
        for(int i = u; i < s.size(); i++){
            if(f[u][i]){
                string t = s.substr(u, i - u + 1);
                path.push_back(t);
                dfs(s, i + 1);
                path.pop_back();
            }
        }
    }
};
```

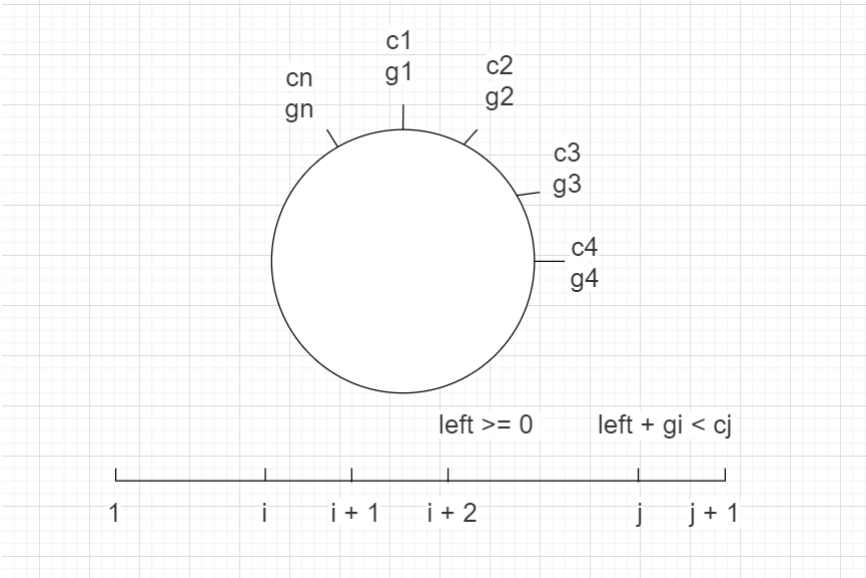
134. 加油站

思路

(贪心) $O(n)$

假如从第`i`个站开始走，没法走一圈回来，而是最多走到第`j`个站，那么从`i...j`之间的一个站点`k`开始走，也没有可能超过`j`这个站，因为从`i`走到`k`的时候是有`>=0`的剩余油量的，但是从第`k`个站出发的话就没有这些油量了，所以更不可能走超过`j`。

因此，如果没法到达第`j`个站，那么我们可以直接排除`i + 1, i + 2,,,j`这些不合法加油站，直接从`j+1`加油站开始作为下一次的起点。



具体过程如下：

- 1、枚举每个起点`i`，初始化剩余油量`left = 0`，每次走的站点数`j = 0`。
- 2、下一个要到达的站点`k = (i + j) % n`。
- 3、到达下一个站点的油量变化为`left += gas[k] - cost[k]`。
- 4、如果`left < 0`，说明当前起点不合法，我们直接`break`。
- 5、如果`j == n`，说明我们又回到了起点，我们返回合法起点`i`，否则下一次起点为`i + j + 1`。

时间复杂度分析： $O(n)$

c++代码

```
class Solution {
public:
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
        int n = gas.size();
        for(int i = 0; i < n; ){// 枚举每个起点
            int left = 0, j;  //剩余油量, 每次走的站点数
            for(j = 0; j < n; j++){
                int k = (i + j) % n;  //下一个要到达的站点
                left += gas[k] - cost[k];
                if(left < 0) break;   //当前起点不合法
            }
            if(j == n) return i;  //返回合法起点
            i = i + j + 1;   //下一次起点为j + 1
        }
        return -1;
    }
};
```

136. 只出现一次的数字

思路

(位运算) $O(n)$

异或运算有以下三个性质:

- 1、`0^0 = 0`，`0^1 = 1`，`0`异或任何数 = 任何数
- 2、`1^0 = 1`，`1^1 = 0`，`1`异或任何数 = 任何数取反

- 3、任何数异或自己=把自己置`0`，即 `a^a=0`

因此这道题可以用位运算来做，过程如下：

- 1、两个相同的元素经过异或之后会变为`0`。
- 2、将数组所有元素异或在一起即可得到出现 `1` 次的元素值。

时间复杂度分析： $O(n)$，其中 n 是数组长度。

c++代码

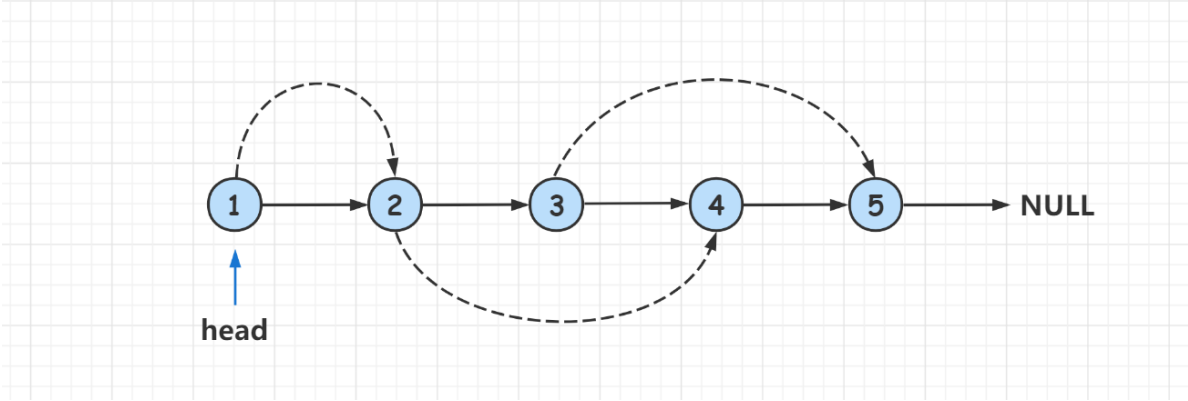
```
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        int res = 0;
        for(int x : nums){
            res ^= x;
        }
        return res;
    }
};
```

138. 复制带随机指针的链表

思路

(迭代) $O(n)$

题目要求我们复制一个长度为`n`的链表，该链表除了每个节点有一个指针指向下一个节点外，还有一个额外的指针指向链表中的任意节点或者`null`，如下图所示：

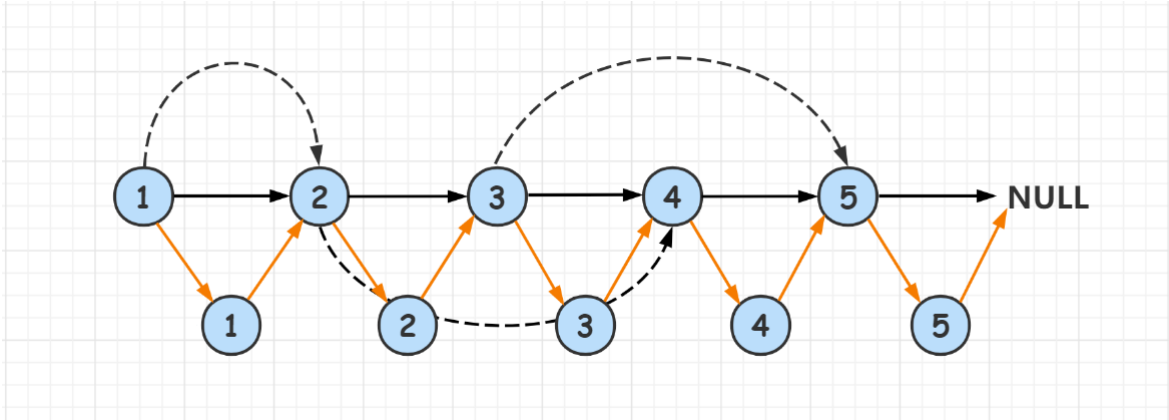


如何去复制一个带随机指针的链表?

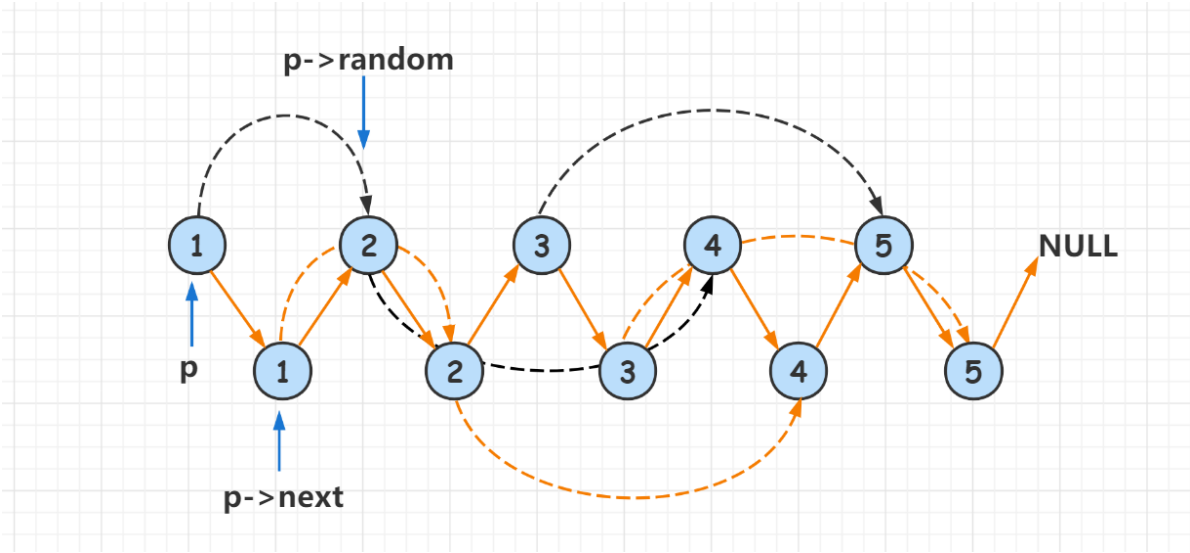
首先我们可以忽略`random`指针，然后对原链表的每个节点进行复制，并追加到原节点的后面，而后复制`random`指针。最后我们把原链表和复制链表拆分出来，并将原链表复原。

图示过程如下：

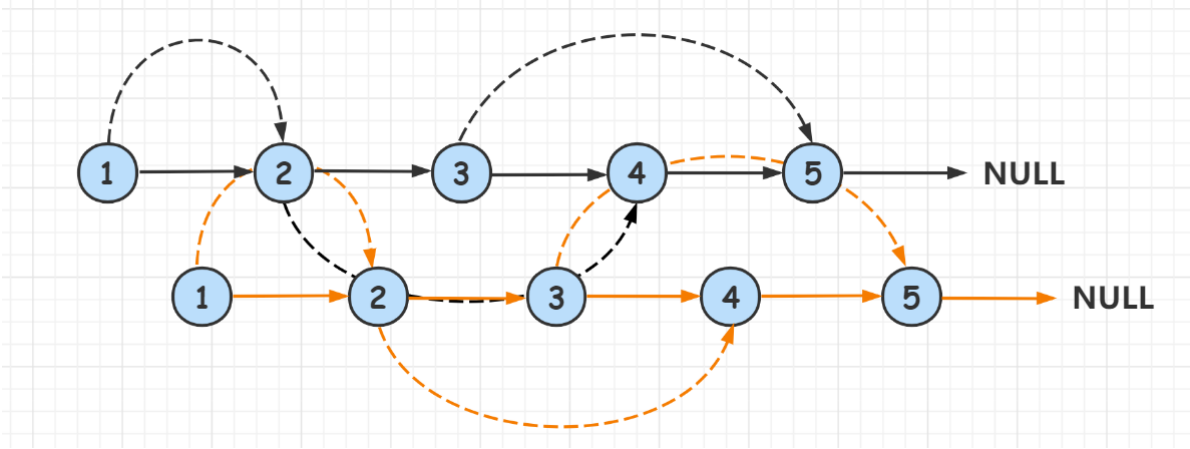
1、在每个节点的后面加上它的复制，并将原链表和复制链表连在一起。



2、 从前往后遍历每一个原链表节点，对于有`random`指针的节点`p`，我们让它的`p->next->random = p->random->next`，这样我们就完成了对原链表`random`指针的复刻。



3、最后我们把原链表和复制链表拆分出来，并将原链表复原。



具体过程如下：

- 1、定义一个`p`指针，遍历整个链表，复制每个节点，并将原链表和复制链表连在一起。
- 2、再次遍历整个链表，执行`p->next->random = p->random->next`，复制`random`指针。
- 3、定义虚拟头节点`dummy`用来指向复制链表的头节点， 将两个链表拆分并复原原链表。

时间复杂度分析： $O(n)$，其中 n 是链表的长度。

c++代码

```
/*
// Definition for a Node.
```

```cpp
class Node {
public:
    int val;
    Node* next;
    Node* random;

    Node(int _val) {
        val = _val;
        next = NULL;
        random = NULL;
    }
};
*/

class Solution {
public:
    Node* copyRandomList(Node* head) {
        for(auto p = head; p; p = p->next->next)  //复制每个节点，并将原链表和复制链表连在一起。
        {
            auto q = new Node(p->val);
            q->next = p->next;
            p->next = q;
        }

        for(auto p = head; p; p = p->next->next)   //复制random指针
        {
            if(p->random)
              p->next->random = p->random->next;
        }

        //拆分两个链表，并复原原链表
        auto dummy = new Node(-1), cur = dummy;
        for(auto p = head; p; p = p->next)
        {
            auto q = p->next;
            cur = cur->next = q;
            p->next = q->next;
        }

        return dummy->next;
    }
};
```

139. 单词拆分

思路

(动态规划) $O(n^3)$

状态表示： `f[i]`表示字符串`s`的前`i`个字符是否可以拆分成`wordDict`，其值有两个`true`和`false`。

状态计算： 假设当前遍历到了第`i`个字符，依据最后一次拆分成的字符串`str`划分集合，最后一次拆分成的字符串`str`可以为`s[0 ~ i - 1]`，`s[1 ~ i - 1]`，，，`s[j ~ i - 1]`。

状态转移方程： `f[i] = ture` 的条件是：`f[j] = ture`并且`s[j, i - 1]`在`hash`表中存在。

初始化： `f[0] = true`，表示空串合法。

实现细节：

为了快速判断字符串`s`拆分出来的子串在`wordDict`中出现，我们可以用一个哈希表存贮`wordDict`中的每个`word`。

时间复杂度分析： 状态枚举$O(n^2)$，状态计算$O(n)$，因此时间复杂度为$O(n^3)$。

c++代码

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        int n =s.size();
        unordered_set<string> hash;
        for(string word : wordDict){
            hash.insert(word);
        }
        vector<bool>f(n + 1, false);
        f[0] = true;
        for(int i = 1; i <= n; i++){
            for(int j = 0; j < i; j++)
                if(f[j] && hash.find(s.substr(j, i - j)) != hash.end()){
                    f[i] = true;
                    break;
                }
        }
        return f[n];
    }
};
```

140. 单词拆分 II

思路

(回溯 + DP)

我们先来看看`LeetCode139`单词拆分 的思路:

状态表示： `f[i]`表示字符串`s`的前`i`个字符是否可以拆分成`wordDict`，其值有两个`true`和`false`。

状态计算： 依据最后一次拆分成的字符串`str`划分集合，最后一次拆分成的字符串`str`可以为`s[0 ~ i - 1]`，`s[1 ~ i - 1]`，，，`s[j ~ i - 1]`。

状态转移方程： `f[i] = ture` 的条件是：`f[j] = ture`并且`s[j, i - 1]`在`hash`表中存在。

初始化： `f[0] = true`，表示空串且合法。

时间复杂度分析： 状态枚举$O(n^2)$，状态计算$O(n)$，因此时间复杂度为$O(n^3)$。

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        unordered_set<string> hash;   //存贮单词
        vector<bool> f(s.size() + 1, false);
        f[0] = true;  //初始化
        for(string word : wordDict){
```

```
            hash.insert(word);
        }
        for(int i = 1; i <= s.size(); i++){
            for(int j = 0; j < i; j++){  //for(int j = 1; j <= i; j++)
                if(f[j] && hash.find(s.substr(j, i - j)) != hash.end()){
                    f[i] = true;
                    break;  //只要有一个子集满足就ok了
                }
            }
        }
        return f[s.size()];
    }
};
```

我们通过`DP`可以判断出一个字符串是否可以被合法的拼接，如果其可以合法拼接，则进行`dfs`，找出所有的情况。

`dfs`函数设计

```
void dfs(string s, int u, string path)
```

- `u`表示下一个单词开始的位置。
- `path`用来记录路径。

`dfs(s, u, path)`表示，从`s[u]`开始拼接下一个单词`s[u, i]`。如果`s[u, i]`出现在`wordBreak`中，并且`f[u] = true`，即`s`的前`u`个字符可以被合法的拼接，则我们将`s[u, i]`拼接到`path`中，并从`i + 1`位置继续递归到下一层。

c++代码

```
class Solution {
public:
    vector<string> res; //记录答案
    unordered_set<string> hash;  //哈希表
    vector<bool> f;
    vector<string> wordBreak(string s, vector<string>& wordDict) {
        int n = s.size();
        f.resize(n + 1);
        f[0] = true;
        for(string word : wordDict){
            hash.insert(word);
        }
        for(int i = 1; i <= n; i++)
            for(int j = 0; j < i; j++){
                if(f[j] && hash.count(s.substr(j, i - j))){
                    f[i] = true;
                    break;
                }
            }
        if(!f[n]) return res;
        dfs(s, 0, "");
        return res;
    }

    void dfs(string& s, int u, string path){
        if(u == s.size()){
```

```
            path.pop_back(); // 去除多余的空格
            res.push_back(path);
            return;
        }
        for(int i = u; i < s.size(); i++){  //[u, i]
            if(f[u] && hash.count(s.substr(u, i - u + 1))){
                dfs(s, i + 1, path + s.substr(u, i - u + 1) + ' ');
            }
        }
    }
};
```

141. 环形链表

思路

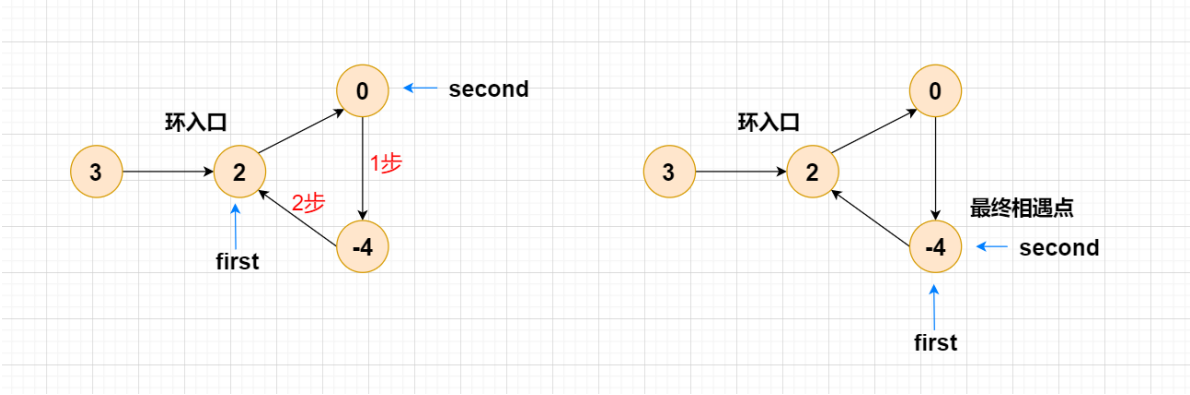
(链表，指针扫描) $O(n)$

用两个指针从头开始扫描，第一个指针每次走一步，第二个指针每次走两步。如果走到 `null`，说明不存在环；否则如果两个指针相遇，则说明存在环。

为什么呢?

假设链表存在环，则当第一个指针走到环入口时，第二个指针已经走到环上的某个位置，距离环入口还差 `x` 步。由于第二个指针每次比第一个指针多走一步，所以第一个指针再走 `x`步，两个指针就相遇了。

如下图所示:



第二个指针还差 `2` 步就可以到达环入口，但是第二个指针每次比第一个指针多走 `1` 步，因此第一个指针再走 `2` 步，两个指针就会相遇。

时间复杂度分析:

第一个指针在环上走不到一圈，所以第一个指针走的总步数小于链表总长度。而第二个指针走的路程是第一个指针的两倍，所以总时间复杂度是$O(n)$。

c++代码

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
```

```cpp
class Solution {
public:
    bool hasCycle(ListNode *head) {
        if(!head || !head->next) return false;
        auto first = head, second = head;
        while(second){
            first = first->next, second = second->next;
            if(!second) return false;
            second = second->next;
            if(second == first) return true;
        }
        return false;
    }
};
```