LC783 Minimum Distance Between BST Nodes

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:

    int minDiffInBST(TreeNode* root) {
        prev = nullptr;
        mindiff = root->val;

        inorder(root);

        return mindiff;
    }

    void inorder(TreeNode* node) {
        if (node == nullptr) return;

        inorder(node->left);

        if (prev) mindiff = std::min(mindiff, node->val - prev->val);
        prev = node;

        inorder(node->right);
    }

private:
    TreeNode* prev;
    int mindiff;
};
```

HackerRank Reverse a doubly linked list

```
/*
 * For your reference:
 *
 * DoublyLinkedListNode {
 *     int data;
 *     DoublyLinkedListNode* next;
 *     DoublyLinkedListNode* prev;
 * };
 *
 */

DoublyLinkedListNode* rev(DoublyLinkedListNode* curr,
DoublyLinkedListNode* prev) {
    if (curr == nullptr) return prev;

    auto next = curr->next;
    curr->next = prev;
    curr->prev = next;

    return rev(next, curr);
}

DoublyLinkedListNode* reverse(DoublyLinkedListNode* head) {
    return rev(head, nullptr);
}
```

LC687 Longest Univalue Path

```
/**
 * Definition of TreeNode:
 * class TreeNode {
 * public:
 *     int val;
 *     TreeNode *left, *right;
 *     TreeNode(int val) {
 *         this->val = val;
 *         this->left = this->right = NULL;
 *     }
 * }
 */

class Solution {
public:
    /**
     * @param root:
     * @return: the length of the longest path where each node in the path has
the same value
     */
    int longestUnivaluePath(TreeNode * root) {
        int longest = 0;

        longestpath1sided(root, longest);
        return longest;

    }

    int longestpath1sided(TreeNode* node, int& longest) {
        if (node == nullptr) return 0;

        // postorder search
        int llength = longestpath1sided(node->left, longest);
        int rlength = longestpath1sided(node->right, longest);

        if (node->left  != nullptr && node->val == node->left->val ) llength +=
1; else llength = 0;
        if (node->right != nullptr && node->val == node->right->val) rlength +=
1; else rlength = 0;

        // update the longest univalue path found
        longest = std::max(longest, llength + rlength);

        // return the longest single-sided univalue path starting at node
        return std::max(llength, rlength);

    }
};
```

LC698 Partition to K Equal Sum Subsets

```
class Solution {
public:
    /**
     * @param nums: a list of integer
     * @param k: an integer
     * @return: return a boolean, denote whether the array can be divided into k
non-empty subsets whose sums are all equal
     */
    bool partitiontoEqualSumSubsets(vector<int> &nums, int k) {
        // write your code here

        int sum = std::accumulate(nums.cbegin(), nums.cend(), 0);

        if (sum % k != 0) return false;

        int target = sum / k;
        std::vector<bool> visited(nums.size(), false);

        return pESS(nums, target, visited, k, 0, 0);

    }

    bool pESS(std::vector<int>& nums, int target, std::vector<bool>& visited, int
k, int start, int cursum) {
        if (k == 1) return true;
        if (cursum > target) return false;
        if (cursum == target) return pESS(nums, target, visited, k - 1, 0, 0);

        for (auto i = start; i != nums.size(); ++i) {
            if (visited[i]) continue;

            visited[i] = true;
            if (pESS(nums, target, visited, k, i + 1, cursum + nums[i])) return
true;

            visited[i] = false;
        }

        return false;

    }
};
```

LC794 Valid Tic-Tac-Toe State

```
class Solution {
public:
    /**
     * @param board: the given board
     * @return: True if and only if it is possible to reach this board position
during the course of a valid tic-tac-toe game
     */
    bool validTicTacToe(vector<string> &board) {
        int numX = 0, numO = 0;

        for (auto& row : board)
            for (auto& ch : row) {
               if      (ch == 'X') ++numX;
               else if (ch == 'O') ++numO;
            }

        // check the numbers of X's and O's
        if ( numX != numO && numX != numO + 1) return false;

        int numSameX = 0, numSameO = 0;

        // check \ diagonal
        if (board[0][0] == board[1][1] && board[0][0] == board[2][2] &
board[1][1] != ' ') {
            if   (board[1][1] == 'X') ++numSameX;
            else ++numSameO;
        }
        // check / diagonal
        else if (board[0][2] == board[1][1] && board[0][2] == board[2][0] &
board[1][1] != ' ') {
            if   (board[1][1] == 'X') ++numSameX;
            else ++numSameO;
        }
        else {
            // check rows
            for (auto& row : board) {
                if      (row == "XXX") ++numSameX;
                else if (row == "OOO") ++numSameO;
            }

            // check cols
            for (int j = 0; j != 3; ++j)
                if (board[0][j] == board[1][j] && board[0][j] == board[2][j]) {
                    if      (board[0][j] == 'X') ++numSameX;
                    else if (board[0][j] == 'O') ++numSameO;
                }
        }

        if (numX == numO && numSameX == 0 || numX > numO && numSameO == 0) return
true;
```

```
        else return false;
    }
};
```

LC726 Number of Atoms

```
class Solution {
public:
    /**
     * @param formula: a string
     * @return: return a string
     */

    using citerator = std::string::const_iterator;
    using map = std::map<std::string, int>;

    string countOfAtoms(string &formula) {
        auto i = formula.cbegin();

        map counts = cOA(i, formula.cend());

        assert(i == formula.cend());

        std::string ans;
        for (auto& kv : counts) {
            ans += kv.first;
            if (kv.second > 1) ans += std::to_string(kv.second);
        }

        return ans;
    }

    map cOA(citerator& i, citerator cend) {
        map counts;

        while (i != cend) {
            if      (*i == '(') {
                map temp = cOA(++i, cend);
                int multiplier = getnumber(i, cend);

                // add counts from inside the bracket
                for (auto& kv : temp)
                    counts[kv.first] += kv.second * multiplier;
            }
            else if (*i == ')') {
                ++i;
                return counts;
            }
            else {
                auto name = getname(i, cend);
                counts[name] += getnumber(i, cend);
            }
        }

        return counts;
    }
```

```
    std::string const getname(citerator& i, citerator cend) {
        assert(std::isupper(*i));

        auto j = i + 1;
        while (j != cend && std::islower(*j)) ++j;
        std::string const name(i, j);

        // must update the position
        i = j;

        return name;
    }

    int getnumber(citerator& i, citerator cend) {
        // *i is either number, (, alphabet or cend
        auto j = i;

        while(j != cend && std::isdigit(*j)) ++j;

        if (j == i) return 1;

        int num = std::stoi(std::string(i, j));

        // must update the position
        i = j;

        return num;
    }
};
```