LeetCode Practice

Outline

Divide and Conquer

Dynamic Programming

Median of Two Sorted Array (#4)

Median of Two Sorted Array (#4)

- Hint 1: Merging and find k-th element will result in O(m + n) time

Median of Two Sorted Array (#4)

- Hint 1: Merging and find k-th element will result in $O(m + n)$ time
- Hint 2: For any number k, can you determine what position is it in the merged array?

Median of Two Sorted Array (#4)

- Hint 1: Merging and find k-th element will result in O(m + n) time
- Hint 2: For any number k, can you determine what position is it in the merged array?

You can do the following:

- The middle element of the first array is the $\frac{m}{2}$-th element. Then determine its position in the 2nd array with binary search, say k. Now, it's $\frac{m}{2} + k$-th element in the merged array. If this is smaller than $\frac{m+n}{2}$, we should proceed with the right half of the first array, otherwise, left half.

Median of Two Sorted Array (#4) Solution

```cpp
class Solution {
public:
  int findKth(vector<int>& nums1, vector<int>& nums2, int s1, int e1, int k) {
    if (s1 == e1)
      return nums2[k - e1 - 1];
    int mid = (s1 + e1) / 2;
    auto it = lower_bound(nums2.begin(), nums2.end(), nums1[mid]);
    int rank = mid + distance(nums2.begin(), it) + 1;
    if (rank == k)
      return nums1[mid];
    if (rank > k)
      return findKth(nums1, nums2, s1, mid, k);
    return findKth(nums1, nums2, mid + 1, e1, k);
  }

  double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
    int sz = nums1.size() + nums2.size();
    if (sz % 2 == 0) {
      return (findKth(nums1, nums2, 0, nums1.size(), sz / 2) +
              findKth(nums1, nums2, 0, nums1.size(), sz / 2 + 1)) / 2.0;
    }
    return findKth(nums1, nums2, 0, nums1.size(), sz / 2 + 1);
  }
};
```

Edit Distance (#72)

Edit distance refers to a group of dynamic programming problems that mostly contains alignment of two or more sequences. The edit distance between two string can be described with the following formula:

$$dist(i,j) = \begin{cases} dist(i-1,j-1) \text{if } A_i = B_j \\ min(dist(i-1,j), dist(i,j-1), dist(i-1,j-1)) + 1 \end{cases} \tag{1}$$

Edit Distance (#72) Solution

```java
class Solution {
    public int minDistance(String word1, String word2) {
        int[][] dist = new int[word1.length() + 1][word2.length() + 1];
        for (int i = 0; i < word1.length(); i++) {
            for (int j = 0; j < word2.length(); j++) {
                dist[i + 1][j + 1] = Integer.MAX_VALUE;
            }
        }
        // Usually, using additional [0][0] can simplify subscript initialization.
        dist[0][0] = 0;
        for (int i = 0; i < word1.length(); i++)
            dist[i + 1][0] = i + 1;
        for (int i = 0; i < word2.length(); i++)
            dist[0][i + 1] = i + 1;
        for (int i = 0; i < word1.length(); i++) {
            for (int j = 0; j < word2.length(); j++) {
                if (word1.charAt(i) == word2.charAt(j)) {
                    dist[i + 1][j + 1] = dist[i][j];
                } else {
                    dist[i + 1][j + 1] = Math.min(Math.min(dist[i][j + 1],
                            dist[i + 1][j]), dist[i][j]) + 1;
                }
            }
        }
        return dist[word1.length()][word2.length()];
    }
}
```

Wildcard Matching (#44)

Wildcard Matching (#44)

- Hint 1: It's similar to edit distance. Can you think of the formula?

Wildcard Matching (#44)

- Hint 1: It's similar to edit distance. Can you think of the formula?
- Hint 2: When aligning two sequences, what would happen between A_i, B_j in the following case:
  - "abc*" and "abc"
  - "ab" and "cb"
  - "ab?" and "abc"

Wildcard Matching (#44)

- Hint 1: It's similar to edit distance. Can you think of the formula?
- Hint 2: When aligning two sequences, what would happen between A_i, B_j in the following case:
  - "abc*" and "abc"
  - "ab" and "cb"
  - "ab?" and "abc"
- Hint 3: You can simplify cases like "**" to "*"

Wildcard Matching (#44)

- Hint 1: It's similar to edit distance. Can you think of the formula?
- Hint 2: When aligning two sequences, what would happen between A_i, B_j in the following case:
  - "abc*" and "abc"
  - "ab" and "cb"
  - "ab?" and "abc"
- Hint 3: You can simplify cases like "**" to "*"
- Hint 4: The formula is:

$$match(i,j) = \begin{cases} false \text{ if } A_i \neq B_j \wedge A_i \neq * \wedge A_i \neq ? \\ match(i-1, j-1) \text{ if } A_i = B_j \vee A_i = ? \\ match(i, j-1) \vee match(i-1, j) \text{ if } A_i = * \end{cases} \quad (2)$$

Wildcard Matching (#44) Solution

```java
public boolean isMatch(String s, String p) {
    boolean[][] dp = new boolean[s.length() + 1][p.length() + 1];
    dp[0][0] = true;
    for (int i = 0; i < p.length(); i++) {
        dp[0][i + 1] = p.charAt(i) == '*' && dp[0][i];
    }
    for (int i = 0; i < s.length(); i++) {
        for (int j = 0; j < p.length(); j++) {
            dp[i + 1][j + 1] =
                    (dp[i][j] && (s.charAt(i) == p.charAt(j) || p.charAt(j) == '?'))
                            || (p.charAt(j) == '*' && (dp[i][j + 1] || dp[i + 1][j]));
        }
    }
    return dp[s.length()][p.length()];
}
```

Interleaving String (#97)

Interleaving String (#97)

- Hint 1: Imagine we have a prefix of $S3, S1, S2$, say $S3', S1', S2'$. What happens if the last character of $S3'$ equals the last character of $S1'$ or $S2'$?

Interleaving String (#97)

- Hint 1: Imagine we have a prefix of $S3, S1, S2$, say $S3', S1', S2'$. What happens if the last character of $S3'$ equals the last character of $S1'$ or $S2'$?
- Hint 2: Let i, j be the length of the prefix $S1', S2'$. The last character of $S3'$ at this point is $S3_(i + j - 1)$. What is the formula?

Interleaving String (#97)

- Hint 1: Imagine we have a prefix of $S3, S1, S2$, say $S3', S1', S2'$. What happens if the last character of $S3'$ equals the last character of $S1'$ or $S2'$?
- Hint 2: Let i, j be the length of the prefix $S1', S2'$. The last character of $S3'$ at this point is $S3_(i + j - 1)$. What is the formula?
- Hint 3: The formula is:

```
interleave(i, j) =
  true  (i = 0, j = 0)
  interleave(0, j - 1) && S2[j-1] == S3[j - 1] (i = 0)
  interleave(i - 1, 0) && S1[i-1] == S3[i - 1] (j = 0)
  interleave(i - 1, j) || interleave(i, j - 1) (S3[i+j-1] == S1[i-1] == S2[j-1])
  interleave(i - 1, j) (S3[i+j-1] == S1[i-1])
  interleave(i, j - 1) (S3[i+j-1] == S2[j-1])
```

Interleaving String (#97) Solution

```java
public boolean isInterleave(String s1, String s2, String s3) {
    if (s1.length() + s2.length() != s3.length()) {
        return false;
    }
    boolean[][] dp = new boolean[s1.length() + 1][s2.length() + 1];
    dp[0][0] = true;
    for (int i = 0; i < s1.length(); i++) {
        dp[i + 1][0] = dp[i][0] && s1.charAt(i) == s3.charAt(i);
    }
    for (int j = 0; j < s2.length(); j++) {
        dp[0][j + 1] = dp[0][j] && s2.charAt(j) == s3.charAt(j);
    }
    for (int i = 0; i < s1.length(); i++) {
        for (int j = 0; j < s2.length(); j++) {
            if (s3.charAt(i + j + 1) == s1.charAt(i) && s3.charAt(i + j + 1) == s2.charAt(j)) {
                dp[i + 1][j + 1] = dp[i][j + 1] || dp[i + 1][j];
                continue;
            }
            if (s3.charAt(i + j + 1) == s1.charAt(i)) {
                dp[i + 1][j + 1] = dp[i][j + 1];
                continue;
            }
            if (s3.charAt(i + j + 1) == s2.charAt(j)) {
                dp[i + 1][j + 1] = dp[i + 1][j];
                continue;
            }
            dp[i + 1][j + 1] = false;
        }
    }
    return dp[s1.length()][s2.length()];
}
```

Minimum ASCII Delete Sum for Two Strings (#712)

Minimum ASCII Delete Sum for Two Strings (#712)

- Hint 1: Similar to edit distance. What will happen if $S1_i == S2_j$ or $S1_i \neq S2_j$?

Minimum ASCII Delete Sum for Two Strings (#712)

- Hint 1: Similar to edit distance. What will happen if $S1_i == S2_j$ or $S1_i \neq S2_j$?
- Hint 2: The formula is:

```
minimum(i, j) =
  minimum(i - 1, j - 1) (S1[i] == S2[j])
  min(minimum(i - 1, j) + S1[i], minimum(i, j - 1) + S2[j]) (otherwise)
```

Minimum ASCII Delete Sum for Two Strings (#712) Solution

```
class Solution {
    public int minimumDeleteSum(String s1, String s2) {
        int[][] dp = new int[s1.length() + 1][s2.length() + 1];
        for (int i = 0; i < s1.length(); i++) {
            dp[i + 1][0] = dp[i][0] + s1.charAt(i);
        }
        for (int i = 0; i < s2.length(); i++) {
            dp[0][i + 1] = dp[0][i] + s2.charAt(i);
        }
        for (int i = 0; i < s1.length(); i++) {
            for (int j = 0; j < s2.length(); j++) {
                if (s1.charAt(i) == s2.charAt(j)) {
                    dp[i + 1][j + 1] = dp[i][j];
                    continue;
                }
                dp[i + 1][j + 1] = Math.min(dp[i][j + 1] + s1.charAt(i),
                        dp[i + 1][j] + s2.charAt(j));
            }
        }
        return dp[s1.length()][s2.length()];
    }
}
```

Regular Expression Matching (#10)

Regular Expression Matching (#10)

- Hint 1: Todo

Regular Expression Matching (#10)

```
// TODO
```

Best Time to Buy and Sell Stock I (#121)

Best Time to Buy and Sell Stock I (#121)

- Hint 1: At Day i if you decide to sell, what is the maximum price you could achieve?

Best Time to Buy and Sell Stock I (#121)

- Hint 1: At Day i if you decide to sell, what is the maximum price you could achieve?
- Hint 2: We can keep a minimum value seen so far and check is $P_i - min$ is greater than current maximum.

Best Time to Buy and Sell Stock I (#121) Solution

```java
public int maxProfit(int[] prices) {
    int minPrice = Integer.MAX_VALUE;
    int maxProfit = 0;
    for (int p : prices) {
        if (p - minPrice > maxProfit)
            maxProfit = p - minPrice;
        if (minPrice > p)
            minPrice = p;
    }
    return maxProfit;
}
```

Best Time to Buy and Sell Stock II (#122)

Best Time to Buy and Sell Stock II (#122)

- Hint 1: How to represent the status of your current holdings?

Best Time to Buy and Sell Stock II (#122)

- Hint 1: How to represent the status of your current holdings?
- Hint 2: You can use two states: bought and sold. How should they transfer upon seeing a new price? e.g. What will happen if bought –> sold at p? Or sold –> bought at p?

Best Time to Buy and Sell Stock II (#122)

- Hint 1: How to represent the status of your current holdings?
- Hint 2: You can use two states: bought and sold. How should they transfer upon seeing a new price? e.g. What will happen if bought –> sold at p? Or sold –> bought at p?
- Hint 3: At price p, we could have:

```
sold = max(bought + p, sold)
bought = max(bought, sold - p)
```

Best Time to Buy and Sell Stock II (Solution)

```
public int maxProfit(int[] prices) {
    int maxBought = Integer.MIN_VALUE;
    int maxSold = 0;

    for (int p : prices) {
        if (maxBought + p > maxSold)
            maxSold = maxBought + p;
        if (maxSold - p > maxBought)
            maxBought = maxSold - p;
    }
    return maxSold;
}
```

Best Time to Buy and Sell Stock III (#123)

Best Time to Buy and Sell Stock III (#123)

- Hint 1: What status can you have? Is it still two Bought / Sold or more?

Best Time to Buy and Sell Stock III (#123)

- Hint 1: What status can you have? Is it still two Bought / Sold or more?
- Hint 2: We can use the following states:
  - Bought1, in 1st transaction, holding 1 stock.
  - Sold1, 1 transaction completed and not holding anything.
  - Bought2, in 2nd transaction, holding 1 stock.
  - Sold2, 2 transaction completed and not holding anything.

Best Time to Buy and Sell Stock III (#123)

- Hint 1: What status can you have? Is it still two Bought / Sold or more?
- Hint 2: We can use the following states:
  - Bought1, in 1st transaction, holding 1 stock.
  - Sold1, 1 transaction completed and not holding anything.
  - Bought2, in 2nd transaction, holding 1 stock.
  - Sold2, 2 transaction completed and not holding anything.
- Hint 3: The state transfer would be: 0 - (buy) -> Bought1 - (sell) -> Sold1 - (buy) -> Bought2 -> (sell) -> Sold2
  At each price P, the above sequence could happen and we'll take the max of each.

Best Time to Buy and Sell Stock III (#123) Solution

```java
public int maxProfit(int[] prices) {
    int maxBought_1 = Integer.MIN_VALUE;
    int maxSold_1 = 0;
    int maxBought_2 = Integer.MIN_VALUE;
    int maxSold_2 = 0;

    for (int p : prices) {
        maxBought_1 = Math.max(maxBought_1, -p);
        if (maxBought_1 + p > maxSold_1)
            maxSold_1 = maxBought_1 + p;
        if (maxSold_1 - p > maxBought_2)
            maxBought_2 = maxSold_1 - p;
        if (maxBought_2 + p > maxSold_2)
            maxSold_2 = maxBought_2 + p;
    }
    return maxSold_2;
}
```

Best Time to Buy and Sell Stock IV (#188)

Best Time to Buy and Sell Stock IV (#188)

- Hint 1: Similar to III, what are the states?

Best Time to Buy and Sell Stock IV (#188)

- Hint 1: Similar to III, what are the states?
- Hint 2: Now we have k states instead of 2. How do you represent them?

Best Time to Buy and Sell Stock IV (#188)

- Hint 1: Similar to III, what are the states?
- Hint 2: Now we have k states instead of 2. How do you represent them?
- Hint 3: Still the states could be represented as:

```
maxBought[0] = max(maxBought[0], -p)
maxBought[i] = max(maxSold[i - 1] - p, maxBought[i])
maxSold[i] = max(maxBought[i] + p, maxSold[i])
```

Best Time to Buy and Sell Stock IV (#188) Solution

```java
public int maxProfit(int k, int[] prices) {
    if (k == 0 || prices.length == 0) {
        return 0;
    }
    // When k > prices.length / 2, this problem is simplified to
    // Best Time to Buy and Sell Stock II as you can complete as
    // many transactions as you like. This is here only to handle
    // LeetCode's corner cases.
    if (k > prices.length / 2) {
        int maxBought = Integer.MIN_VALUE;
        int maxSold = 0;

        for (int p : prices) {
            if (maxBought + p > maxSold)
                maxSold = maxBought + p;
            if (maxSold - p > maxBought)
                maxBought = maxSold - p;
        }
        return maxSold;
    }

    int[] maxBought = new int[k];
    int[] maxSold = new int[k];
    Arrays.fill(maxBought, Integer.MIN_VALUE);
    for (int p : prices) {
        maxBought[0] = Math.max(maxBought[0], -p);
        for (int i = 0; i < k - 1; i++) {
            maxSold[i] = Math.max(maxBought[i] + p, maxSold[i]);
            maxBought[i + 1] = Math.max(maxSold[i] - p, maxBought[i + 1]);
        }
        maxSold[k - 1] = Math.max(maxSold[k - 1], maxBought[k - 1] + p);
    }
    return maxSold[k - 1];
}
```

Paint House II

Paint House II

- Hint 1: What are the states? Do you need to examine all colors in each step?

Paint House II

- Hint 1: What are the states? Do you need to examine all colors in each step?
- Hint 2: You don't have to examine all colors in each step – using the colors with lowest 2 values would be sufficient, since the next set of lowest values would exactly come from these 2 values + current price.

Paint House II

- Hint 1: What are the states? Do you need to examine all colors in each step?
- Hint 2: You don't have to examine all colors in each step – using the colors with lowest 2 values would be sufficient, since the next set of lowest values would exactly come from these 2 values + current price.
- Hint 3: The formula is:

```
let i = 0..k such that prices(n - 1, i) is smallest
    j = 0..k such that prices(n - 1, j) is second smallest
prices(n, k) =
  prices(n - 1, i) + cost[n][k], if i != k
  prices(n - 1, j) + cost[n][k], if i == k
```

Do you need O(nk) storage space?

Paint House II (Solution)

```java
public int minCostII(int[][] costs) {
    if (costs.length == 0)
        return 0;

    int [] cost = new int[costs[0].length];

    for (int i = 0; i < costs[0].length; i++)
        cost[i] = costs[0][i];

    for (int i = 1; i < costs.length; i++) {
        int[] prices = costs[i];

        // Find the lowest 2 cost.
        int minCost1 = Integer.MAX_VALUE, minColor1 = -1;
        int minCost2 = Integer.MAX_VALUE;
        for (int j = 0; j < cost.length; j++) {
            if (cost[j] < minCost1) {
                minCost2 = minCost1;
                minCost1 = cost[j];
                minColor1 = j;
                continue;
            }
            if (cost[j] < minCost2) {
                minCost2 = cost[j];
            }
        }

        for (int j = 0; j < prices.length; j++) {
            if (j == minColor1) {
                cost[j] = minCost2 + prices[j];
            } else {
                cost[j] = minCost1 + prices[j];
            }
        }
    }

    return Arrays.stream(cost).min().orElse(-1);
}
```

Max Consecutive Ones (#485)

Max Consecutive Ones (#485)

- Hint 1: It's quite similar to Stock. What is the state you'll need to keep?

Max Consecutive Ones (#485)

- Hint 1: It's quite similar to Stock. What is the state you'll need to keep?
- Hint 2: You can keep two numbers: current consecutive ones and a max.

Max Consecutive Ones (#485) Solution

```
public int findMaxConsecutiveOnes(int[] nums) {
    int max = 0;
    int current = 0;
    for (int x : nums) {
        if (x == 0) current = 0;
        else current += 1;
        max = Math.max(max, current);
    }
    return max;
}
```

Knapsack Styled DP

Knapsack problems are pseudo-polynomial time. They require DP over the value domain of some of the parameters. The characteristic of the problems of this kind is they are often quite small on value range. For example, in subset sum, the largest number is usually in terms of 100s.

Coin Change II (#518)

Coin Change II (#518)

- Hint 1: What is the sub-problem?

Coin Change II (#518)

- Hint 1: What is the sub-problem?
- Hint 2: For a specific coin, I can use it or not use it. What is the difference?

Coin Change II (#518)

- Hint 1: What is the sub-problem?
- Hint 2: For a specific coin, I can use it or not use it. What is the difference?
- Hint 3: The formula is:

```
// # of ways to make value k from coins 0..n:
coin(n, k) =
  // We don't use coin[n] or use it
  coin(n - 1, k) + coin(n, k - value[n])
```

Coin Change II (#518) Solution

```
public int change(int amount, int[] coins) {
    int[][] dp = new int[coins.length + 1][amount + 1];
    for (int i = 0; i <= coins.length; i++) {
        dp[i][0] = 1;
    }

    for (int i = 0; i < coins.length; i++) {
        for (int j = 0; j <= amount; j++) {
            int useCoin = (j >= coins[i]) ? dp[i + 1][j - coins[i]] : 0;
            dp[i + 1][j] = useCoin + dp[i][j];
        }
    }
    return dp[coins.length][amount];
}
```

Coin Change I (#322)

Coin Change I (#322)

- Hint 1: Since we have infinite number of coins for each kind, we always have the same set to make any value. So what should be the state?

Coin Change I (#322)

- Hint 1: Since we have infinite number of coins for each kind, we always have the same set to make any value. So what should be the state?
- Hint 2: We can use change as state. What is the formula?

Coin Change I (#322)

- Hint 1: Since we have infinite number of coins for each kind, we always have the same set to make any value. So what should be the state?
- Hint 2: We can use change as state. What is the formula?
- Hint 3: The formula is:

```
changes[i] = min(changes[i - coins[j]] + 1) for j = 0 to coins.length.
```

You'll need to work out the corner cases.

Coin Change I (#322) Solution

```
public int coinChange(int[] coins, int change) {
    int[] changes = new int[change + 1];
    Arrays.fill(changes, Integer.MAX_VALUE);

    changes[0] = 0;
    for (int i = 0; i < coins.length; i++) {
        if (coins[i] <= change) {
            changes[coins[i]] = 1;
        }
    }

    for (int i = 1; i <= change; i++) {
        for (int coin : coins) {
            if (i >= coin && changes[i - coin] != Integer.MAX_VALUE) {
                changes[i] = Math.min(changes[i], changes[i - coin] + 1);
            }
        }
    }

    return changes[change] == Integer.MAX_VALUE ? -1 : changes[change];
}
```

Partition Equal Subset Sum (#416)

Partition Equal Subset Sum (#416)

- Hint 1: This is actually another coin change. What is the change and what are the coins? What are the constraints compared to coin change?

Partition Equal Subset Sum (#416)

- Hint 1: This is actually another coin change. What is the change and what are the coins? What are the constraints compared to coin change?
- Hint 2: The target change is $sum/2$. The contraint is each coin can only be used once. How should you encode such info in the formula?

Partition Equal Subset Sum (#416)

- Hint 1: This is actually another coin change. What is the change and what are the coins? What are the constraints compared to coin change?
- Hint 2: The target change is $sum/2$. The contraint is each coin can only be used once. How should you encode such info in the formula?
- Hint 3: The formula is:

```
// canSum(i, target) represents whether we can select nums[0..i] to
// get the sum target.
canSum(i, target) = canSum(i - 1, target) || canSum(i - 1, target - nums[i])
```

Again, please work out the edge cases.

Partition Equal Subset Sum (#416) Solution

```java
public boolean canPartition(int[] nums) {
    int sum = Arrays.stream(nums).sum();
    if (sum % 2 != 0) {
        return false;
    }
    int target = sum / 2;

    boolean[][] canSum = new boolean[nums.length + 1][target + 1];

    for (int i = 0; i <= nums.length; i++) {
        canSum[i][0] = true;
    }

    for (int i = 1; i <= nums.length; i++) {
        for (int j = 1; j <= target; j++) {
           canSum[i][j] = (j >= nums[i - 1] && canSum[i - 1][j - nums[i - 1]])
                    || canSum[i - 1][j];
        }
    }

    return canSum[nums.length][target];
}
```

House Robber III (#337)

House Robber III (#337)

- Hint 1: This is a little bit tricky. For each root node, you have 2 options rob it or no. If root is robbed, you should not rob its left child and right child. Otherwise, you can choose to rob either child, both children or none. How do you represent the state?

House Robber III (#337)

- Hint 1: This is a little bit tricky. For each root node, you have 2 options rob it or no. If root is robbed, you should not rob its left child and right child. Otherwise, you can choose to rob either child, both children or none. How do you represent the state?

- Hint 2: You can use two hashmap: *hasRoot<TreeNode, Int>*, *noRoot<TreeNode, Int>*. Then you can establish a connection between its a node and its children and get the formula.

House Robber III (#337)

- Hint 1: This is a little bit tricky. For each root node, you have 2 options rob it or no. If root is robbed, you should not rob its left child and right child. Otherwise, you can choose to rob either child, both children or none. How do you represent the state?
- Hint 2: You can use two hashmap: *hasRoot<TreeNode, Int>*, *noRoot<TreeNode, Int>*. Then you can establish a connection between its a node and its children and get the formula.
- Hint 3: The formula is:

```
hasRoot(root) = root.val + noRoot(root.left) + noRoot(root.right);
noRoot(root) = max(noRoot(root.left), hasRoot(root.left)) +
               max(noRoot(root.right), hasRoot(root.right))
```

House Robber III (#337) Solution

```java
// Do a level order tranversal so that we could manipulate nodes bottom-up.
private ArrayList<TreeNode> addNodes(TreeNode root) {
    ArrayList<TreeNode> nodes = new ArrayList<>();
    int index = 0;
    nodes.add(root);
    while (index < nodes.size()) {
        TreeNode cur = nodes.get(index);
        if (cur.left != null) nodes.add(cur.left);
        if (cur.right != null) nodes.add(cur.right);
        index++;
    }
    return nodes;
}

private int getOrZero(TreeNode node, HashMap<TreeNode, Integer> map) {
    if (node != null && map.containsKey(node)) return map.get(node);
    return 0;
}

public int rob(TreeNode root) {
    if (root == null) return 0;

    HashMap<TreeNode, Integer> hasRoot = new HashMap<>();
    HashMap<TreeNode, Integer> noRoot = new HashMap<>();

    ArrayList<TreeNode> nodes = addNodes(root);

    for (int i = nodes.size() - 1; i >= 0; i--) {
        TreeNode node = nodes.get(i);
        int noRootLeft = getOrZero(node.left, noRoot);
        int noRootRight = getOrZero(node.right, noRoot);
        hasRoot.put(node, noRootLeft + noRootRight + node.val);
        int hasRootLeft = getOrZero(node.left, hasRoot);
        int hasRootRight = getOrZero(node.right, hasRoot);
        noRoot.put(node, Math.max(hasRootLeft, noRootLeft)
                + Math.max(hasRootRight, noRootRight));
    }
    return Math.max(getOrZero(root, hasRoot), getOrZero(root, noRoot));
}
```

<<<<<<< HEAD

Unique Paths (#62)

Unique Paths (#62)

- Hint 1: When robot is at (x, y), where can it come from?

Unique Paths (#62)

- Hint 1: When robot is at (x, y), where can it come from?
- Hint 2: The robot can come from (x - 1, y) or (x, y - 1). What is the formula?

Unique Paths (#62)

- Hint 1: When robot is at (x, y), where can it come from?
- Hint 2: The robot can come from (x - 1, y) or (x, y - 1). What is the formula?
- Hint 3: The formula is:

```
pos(x, y) = pos(x - 1, y) + pos(x, y - 1)
```

Unique Paths (#62) Solution

```
public int uniquePaths(int m, int n) {
    int[][] dp = new int[m][n];
    for (int i = 0; i < m; i++) {
        dp[i][0] = 1;
    }
    for (int i = 0; i < n; i++) {
        dp[0][i] = 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 Paths II (#63)

Unique Paths II (#63)

- Hint 1: The same as unique paths I. You can use 2D grid and coordinates as states. What to do with obstacles?

Unique Paths II (#63)

- Hint 1: The same as unique paths I. You can use 2D grid and coordinates as states. What to do with obstacles?
- Hint 2: If grid(x, y) == 1 then pos(x, y) = 0. The rest are the same.

Unique Paths II (#63)

- Hint 1: The same as unique paths I. You can use 2D grid and coordinates as states. What to do with obstacles?
- Hint 2: If grid(x, y) == 1 then pos(x, y) = 0. The rest are the same.
- Hint 3: The formula is:

```
pos(x, y) = 0 if grid(x, y) == 1
pos(x, y) = pos(x - 1, y) + pos(x, y - 1) otherwise
```

Unique Paths II (#63) Solution

```
public int uniquePathsWithObstacles(int[][] obstacleGrid) {
    int m = obstacleGrid.length;
    int n = obstacleGrid[0].length;
    int[][] dp = new int[m][n];
    dp[0][0] = obstacleGrid[0][0] == 0 ? 1 : 0;
    for (int i = 1; i < m; i++) {
        dp[i][0] = obstacleGrid[i][0] == 0 ? dp[i - 1][0] : 0;
    }

    for (int i = 1; i < n; i++) {
        dp[0][i] = obstacleGrid[0][i] == 0 ? dp[0][i - 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];
}
```