LeetCode Practice

Outline

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

```
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

```
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

```
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
```

Single Sequence Styled DP

There is only one sequence. Current state is often determined by 1 or more previous states.

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)

```java
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)

```
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

```java
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;
}
```

Climbing Stairs (#70)

Climbing Stairs (#70)

- Hint 1: For current step, what status will it depend on?

Climbing Stairs (#70)

- Hint 1: For current step, what status will it depend on?
- Hint 2: It only depends on 2 previous stairs: i - 1 and i - 2.

Climbing Stairs (#70)

- Hint 1: For current step, what status will it depend on?
- Hint 2: It only depends on 2 previous stairs: i - 1 and i - 2.
- Hint 3: The formula is:

```
step(i) = step(i - 1) + step(i - 2)
```

So it only depends on 2 variables. And yes, it's same as getting n-th element from Fibonacci sequence.

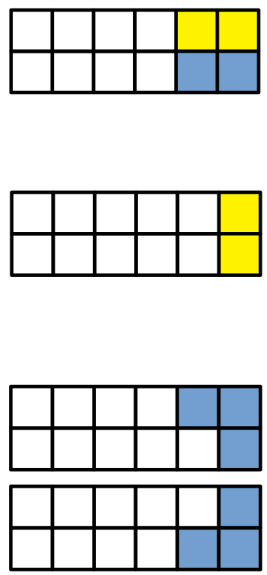
Climbing Stairs (#70) Solution

```
public int climbStairs(int n) {
    int a0 = 0;
    int a1 = 1;
    for (int i = 0; i < n; i++) {
        int a2 = a0 + a1;
        a0 = a1;
        a1 = a2;
    }
    return a1;
}
```

Domino and Tromino Tiling (#790)

Domino and Tromino Tiling (#790)

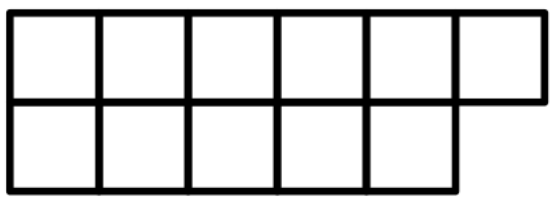
- Hint 1: Consider the following graph. How many types of states are there?



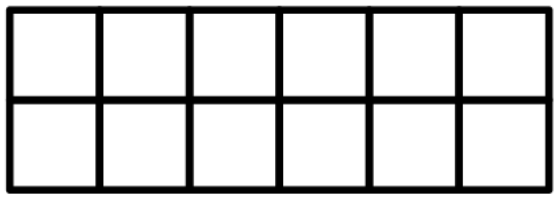
Domino and Tromino Tiling (#790)

- Hint 2: We can classify them into two categories: OneOut and Smooth. How can you construct the subproblem for both types?

OneOut

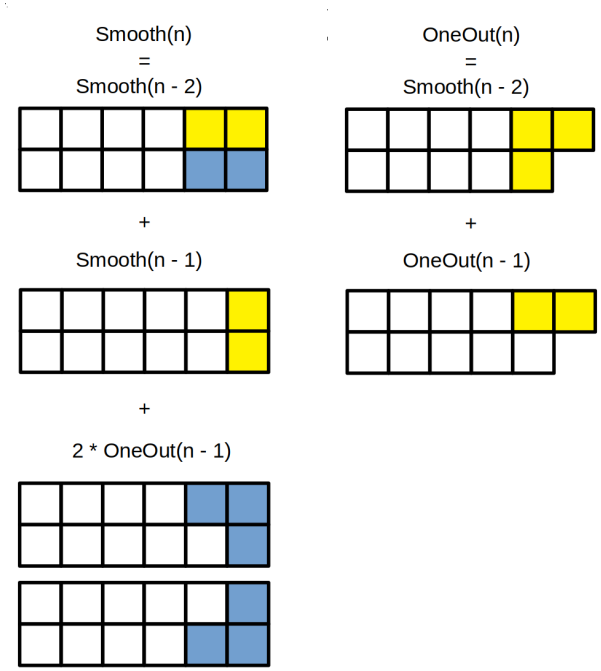


Smooth



Domino and Tromino Tiling (#790)

- Hint 3: You can have the following formula:



Domino and Tromino Tiling (#790) Solution

```java
public int numTilings(int N) {
    final int MOD = 1000000007;

    if (N == 0) return 0;
    if (N == 1) return 1;
    if (N == 2) return 2;

    // Use long to avoid overflow during addition.
    long[] smooth = new long[N];
    long[] oneOut = new long[N];

    smooth[0] = 1;
    smooth[1] = 2;

    oneOut[1] = 1;

    for (int i = 2; i < N; i++) {
        smooth[i] = (smooth[i - 1] + smooth[i - 2] + 2 * oneOut[i - 1]) % MOD;
        oneOut[i] = (smooth[i - 2] + oneOut[i - 1]) % MOD;
    }

    return (int) smooth[N - 1];
}
```

Bonus Point: Can you solve it with O(1) space?

Domino and Tromino (#790) Solution

Notice that smooth_i / oneOut_i only depends on smooth_{i-1}, smooth_{i-2} and oneOut_{i-1}.

```
public int numTilings(int N) {
    if (N == 0) return 0;
    if (N == 1) return 1;

    final int MOD = 1000000007;

    long smooth0 = 1;
    long smooth1 = 2;
    long oneOut0 = 1;

    for (int i = 2; i < N; i++) {
        long smooth2 = (smooth1 + smooth0 + 2 * oneOut0) % MOD;
        oneOut0 = (smooth0 + oneOut0) % MOD;
        smooth0 = smooth1;
        smooth1 = smooth2;
    }

    return (int) smooth1;
}
```

Frog Jump (#403)

Frog Jump (#403)

- Hint 1: Whether the frog can reach stone_i depends solely on stones (0..i-1).

Frog Jump (#403)

- Hint 1: Whether the frog can reach $stone_i$ depends solely on stones (0..i-1).
- Hint 2: If the frog jumps from $stone_j$ to $stone_i$, it must have jumped from another $stone_k$ to $stone_j$, where $|i - j - (j - k)| <= 1$.

Frog Jump (#403)

- Hint 1: Whether the frog can reach stone_i depends solely on stones (0..i-1).
- Hint 2: If the frog jumps from stone_j to stone_i, it must have jumped from another stone_k to stone_j, where |i - j - (j - k)| <= 1.
- Hint 3: One way to express this is:

```
jump(i) = {j | j = 0..i - 1 and (i - j - 1 in jump[j] or
               i - j in jump[j] or i - j + 1 in jump[j])}
jump(0) = {0}
The frog can reach the end iff jump(last) is not empty.
```

Frog Jump (#403) Solution

```
public boolean canCross(int[] stones) {
    ArrayList<HashSet<Integer>> dp = new ArrayList<>();
    for (int ignored : stones) dp.add(new HashSet<>());
    dp.get(0).add(0);
    for (int i = 1; i < stones.length; i++) {
        for (int j = 0; j < i; j++) {
            int step = stones[i] - stones[j];
            if (dp.get(j).contains(step - 1) ||
                    dp.get(j).contains(step) ||
                    dp.get(j).contains(step + 1)) {
                dp.get(i).add(step);
            }
        }
    }
    return !dp.get(stones.length - 1).isEmpty();
}
```

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

```java
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

```java
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];
}
```

Tree Style DP

This should not be very common. Each tree node represents one optimal value when we apply the operation within that subtree.

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));
}
```

Coordinate Style DP

This normally consists of a grid-like structure, with coordinates representing the states.

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

```java
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];
}
```

Triangle (#120)

Triangle (#120)

- Hint 1: Similar to unique paths, you can still use coordinates as state.

Triangle (#120)

- Hint 1: Similar to unique paths, you can still use coordinates as state.
- Hint 2: The formula is:

```
a(i, j) = min(a(i - 1, j - 1), a(i - 1, j)) + triangle(i, j)
```

Triangle (#120)

- Hint 1: Similar to unique paths, you can still use coordinates as state.
- Hint 2: The formula is:

```
a(i, j) = min(a(i - 1, j - 1), a(i - 1, j)) + triangle(i, j)
```

- Hint 3: For layer N, you may only care about layer N - 1, which saves you from using $O(n^2)$ space

Triangle (#120) Solution

```java
public int minimumTotal(List<List<Integer>> triangle) {
    int[] dp = new int[triangle.size()];
    for (int i = 0; i < triangle.size(); i++) {
        int[] tmp = new int[triangle.size()];
        List<Integer> row = triangle.get(i);
        for (int j = 0; j < row.size(); j++) {
            if (i == 0 || j == 0) tmp[j] = dp[j] + row.get(j);
            else if (j == row.size() - 1) tmp[j] = dp[j - 1] + row.get(j);
            else tmp[j] = Math.min(dp[j], dp[j - 1]) + row.get(j);
        }
        dp = tmp;
    }
    int min = Integer.MAX_VALUE;
    for (int a : dp) min = Math.min(min, a);
    return min;
}
```

Dungeon Game (#174)

Dungeon Game (#174)

- Hint 1: Traditionally if we walk forward, we don't know what the start HP is. It's not easy to go right / down. It's probably better to go backwards, since we know at the point we reach the princes, we should have at least 1 HP to spare.

Dungeon Game (#174)

- Hint 1: Traditionally if we walk forward, we don't know what the start HP is. It's not easy to go right / down. It's probably better to go backwards, since we know at the point we reach the princes, we should have at least 1 HP to spare.

- Hint 2: The formula is:

```
// minHp represents the minimum Hp we need *before* we step on cell[i][j].
minHp(i, j) = min(minHp(i + 1, j) ? dungeon(i, j), minHp(i, j + 1) ? dungeon(i, j))
```

Can you guess what ? should represent?

Dungeon Game (#174)

- Hint 1: Traditionally if we walk forward, we don't know what the start HP is. It's not easy to go right / down. It's probably better to go backwards, since we know at the point we reach the princes, we should have at least 1 HP to spare.

- Hint 2: The formula is:

```
// minHp represents the minimum Hp we need *before* we step on cell[i][j].
minHp(i, j) = min(minHp(i + 1, j) ? dungeon(i, j), minHp(i, j + 1) ? dungeon(i, j))
```

Can you guess what ? should represent?

- Hint 3: ? could be represented as the following function:

```
// lastCell is either [i+1][j] or [i][j+1]. currentCell is dungeon[i][j].
// This says that if we want to move from currentCell to lastCell, the Hp we need
// before we step onto [i][j] so that we can finally reach the bottom / right cell.
private int getValue(int lastCell, int currentCell) {
    // If [i][j] is negative, we'll need to add that to our budget.
    if (currentCell < 0) return lastCell - currentCell;
    // If lastCell's required amount is less than the amount we can gain from
    // current cell, we only need to be alive before we step on it.
    if (lastCell <= currentCell) return 1;
    // Otherwise, we can charge up at current cell to the point that lastCell
    // requires.
    return lastCell - currentCell;
}
```

Dungeon Game (#174) Solution

```
private int getValue(int lastCell, int currentCell) {
    if (currentCell < 0) return lastCell - currentCell;
    if (lastCell <= currentCell) return 1;
    return lastCell - currentCell;
}

public int calculateMinimumHP(int[][] dungeon) {
    int[][] minHp = new int[dungeon.length][dungeon[0].length];
    int h = dungeon.length - 1;
    int w = dungeon[0].length - 1;
    for (int i = h; i >= 0; i--) {
        for (int j = w; j >= 0; j--) {
            if (i == h && j == w) minHp[i][j] = getValue(1, dungeon[i][j]);
            else if (i == h) minHp[i][j] = getValue(minHp[i][j + 1], dungeon[i][j]);
            else if (j == w) minHp[i][j] = getValue(minHp[i + 1][j], dungeon[i][j]);
            else minHp[i][j] = Math.min(getValue(minHp[i + 1][j], dungeon[i][j]),
                        getValue(minHp[i][j + 1], dungeon[i][j]));
        }
    }
    return minHp[0][0];
}
```

Higher Ordered DPs

I'm calling this type "higher ordered DP" because it generally requires $O(n^3)$ to solve. One example of it would be matrix multiplication:
Given matrix chain ABCDEFG, calculate the product of it with least number of multiplications. Different number of multiplications could be achieved by adding parentheses. For example, (A(B(CD)E)(FG)) and ((ABCD)(EFG)) may require different number of multiplications.
Read
`https://home.cse.ust.hk/~dekai/271/notes/L12/L12.pdf`
for more information on matrix multiplication.

Burst Balloons (#312)

Burst Balloons (#312)

- Hint 1: If I burst balloons between (i, j) and iand j becomes neighbor, assume the sequence is optimal, do any subsequent operations affect it?

Burst Balloons (#312)

- Hint 1: If I burst balloons between (i, j) and iand j becomes neighbor, assume the sequence is optimal, do any subsequent operations affect it?
- Hint 2: The formula:

```
max(i, j) = max{max(i, k) + max(k, j) + balloons[i] * balloons[k] * balloons[j]}.
```

- Hint 3: It helps to prepend 1 and append 1 to both sides of the array.

Burst Balloons (#312) Solution

```java
public int maxCoins(int[] nums) {
    int[] balloons = new int[nums.length + 2];
    balloons[0] = balloons[balloons.length - 1] = 1;
    System.arraycopy(nums, 0, balloons, 1, nums.length);
    int[][] dp = new int[balloons.length][balloons.length];
    // As indicated in Page 19 of https://home.cse.ust.hk/~dekai/271/notes/L12/L12.pdf,
    // we'll need to calculate sequences ordered by length. So k represents the length
    // of the sequence.
    for (int k = 2; k <= balloons.length - 1; k++) {
        for (int i = 0; i < balloons.length - k; i++) {
            int end = i + k;
            for (int j = i + 1; j <= end - 1; j++) {
                int score = dp[i][j] + dp[j][end] + balloons[i] * balloons[j] * balloons[end];
                if (score > dp[i][end]) dp[i][end] = score;
            }
        }
    }
    return dp[0][balloons.length - 1];
}
```

Monotonic Stack

It's generally used for linear data structure. You maintain a (stack-like) structure that is increasing or decreasing. When you encounter an element that's smaller / larger than this one, you keep popping the stack until the element is larger than / smaller than the current top of the stack. In the process of popping, you can do various operations on it. Since each element enters the stack and is popped off the stack once, it's still linear.

Min Stack (#155)

Min Stack (#155)

- Hint 1: You'll need to keep all the minimum numbers that you encountered so far. How do you do that?

Min Stack (#155)

- Hint 1: You'll need to keep all the minimum numbers that you encountered so far. How do you do that?
- Hint 2: You can use 2 stacks. One regular and one for min numbers. If a newly encountered number is smaller than or equal to the one on top of the min stack, you push it onto the min stack as well.

Min Stack (#155) Solution

```
class MinStack {
    private Stack<Integer> st;
    private Stack<Integer> min;

    /** initialize your data structure here. */
    public MinStack() {
        st = new Stack<>();
        min = new Stack<>();
    }

    public void push(int x) {
        st.push(x);
        if (min.empty() || min.peek() >= x)
            min.push(x);
    }

    public void pop() {
        int e = st.pop();
        if (e == min.peek())
            min.pop();
    }

    public int top() {
        return st.peek();
    }

    public int getMin() {
        return min.peek();
    }
}
```

Remove K Digits (#402)

Remove K Digits (#402)

- Hint 1: What you need to do to make sure the end number is smaller?

Remove K Digits (#402)

- Hint 1: What you need to do to make sure the end number is smaller?
- Hint 2: You can use a stack, when you encounter an element that's smaller than the top of the stack, you keep on popping. Then push the element onto the stack. You'll need to keep track of k as well, since you can only pop k times maximum.

Remove K Digits (#402) Solution

```java
public String removeKdigits(String num, int k) {
    StringBuilder stack = new StringBuilder();
    for (char n : num.toCharArray()) {
        while (stack.length() > 0 && stack.charAt(stack.length() - 1) > n && k > 0) {
            k--;
            stack.deleteCharAt(stack.length() - 1);
        }
        stack.append(n);
    }
    while (k != 0 && stack.length() > 0) {
        stack.deleteCharAt(stack.length() - 1);
        k--;
    }
    while (stack.length() > 0 && stack.charAt(0) == '0') {
        stack.deleteCharAt(0);
    }
    return stack.length() == 0 ? "0" : stack.toString();
}
```

Next Greater Element II (#503)

Next Greater Element II (#503)

- Hint 1: You'll need to keep track the elements you've encountered so far until you meed a greater element. Then assign all the elements that are smaller than that one the index. What can you do?

Next Greater Element II (#503)

- Hint 1: You'll need to keep track the elements you've encountered so far until you meed a greater element. Then assign all the elements that are smaller than that one the index. What can you do?
- Hint 2: You can use a stack that's monotonically decreasing. When you find an element that's greater than the stack top, you keep popping element and assign the index. Just remember the array counts circular so you'll need to take care of it.

Next Greater Element II (#503) Solution

```java
public int[] nextGreaterElements(int[] nums) {
    int[] result = new int[nums.length];
    Arrays.fill(result, -1);
    Stack<Integer> stack = new Stack<>();
    for (int i = 0; i < 2 * nums.length - 1; i++) {
        int index = i % nums.length;
        while (!stack.empty() && nums[index] > nums[stack.peek()]) {
            result[stack.pop()] = nums[index];
        }
        if (result[index] == -1) {
            stack.push(index);
        }
    }
    return result;
}
```

Daily Temperatures (#739)

Daily Temperatures (#739)

- Hint 1: You'll need to keep a list of temperatures until it's not decreasing.

Daily Temperatures (#739)

- Hint 1: You'll need to keep a list of temperatures until it's not decreasing.
- Hint 2: You can use a monotonically decreasing stack. Pop the stack when you see a number that's greater than current value.

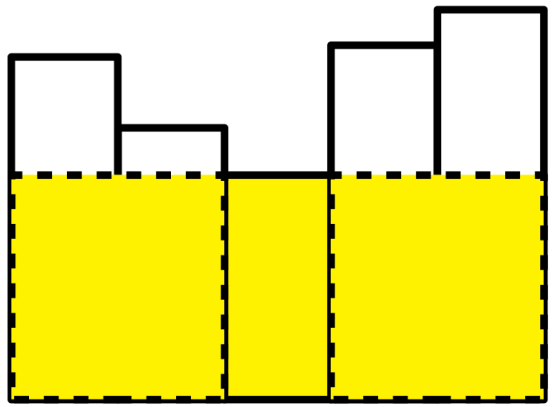
Daily Temperatures (#739) Solution

```java
public int[] dailyTemperatures(int[] temperatures) {
    int[] result = new int[temperatures.length];
    Stack<Integer> stack = new Stack<>();
    for (int i = 0; i < temperatures.length; i++) {
        while (!stack.empty() && temperatures[stack.peek()] < temperatures[i]) {
            int index = stack.pop();
            result[index] = i - index;
        }
        stack.push(i);
    }
    while (!stack.empty()) {
        result[stack.pop()] = 0;
    }
    return result;
}
```

Largest Rectangle in Histogram (#84)

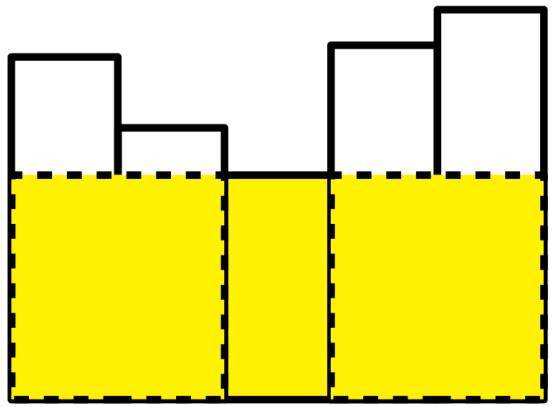
Largest Rectangle in Histogram (#84)

- Hint 1: For each element, how is the area of the rectangle that includes this element is calculated?



Largest Rectangle in Histogram (#84)

- Hint 1: For each element, how is the area of the rectangle that includes this element is calculated?



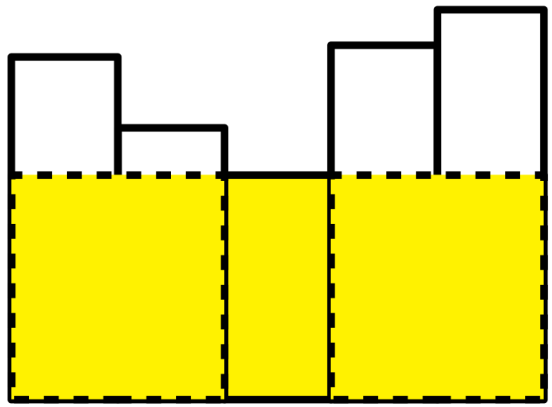
- Hint 2: The area of the rectangle is:

```
((Num of continous rect on left with greater height)
 + (Num of continous rect on right with greater height))
 * rect[i].height
```

How can we calculate this?

Largest Rectangle in Histogram (#84)

- Hint 1: For each element, how is the area of the rectangle that includes this element is calculated?



- Hint 2: The area of the rectangle is:

```
((Num of continous rect on left with greater height)
 + (Num of continous rect on right with greater height))
 * rect[i].height
```

How can we calculate this?

- Hint 3: You can use a monotonic stack to compute the Num of rects! Do it from left to right then from right to left.

Largest Rectangle in Histogram (#84) Solution

```
private void addWidth(int[] heights, int[] widths) {
    Stack<Integer> stack = new Stack<>();
    for (int i = 0; i < heights.length; i++) {
        while (!stack.empty() && heights[stack.peek()] > heights[i]) {
            int index = stack.pop();
            widths[index] += i - index;
        }
        stack.push(i);
    }
    while (!stack.empty()) {
        int index = stack.pop();
        widths[index] += heights.length - index;
    }
}

private int[] reverse(int[] array) {
    for (int i = 0; i < array.length / 2; i++) {
        int tmp = array[i];
        array[i] = array[array.length - i - 1];
        array[array.length - i - 1] = tmp;
    }
    return array;
}

public int largestRectangleArea(int[] heights) {
    Stack<Integer> stack = new Stack<>();
    int[] widths = new int[heights.length];
    int maxArea = 0;
    addWidth(heights, widths);
    addWidth(reverse(heights), reverse(widths));
    for (int i = 0; i < heights.length; i++) {
        maxArea = Math.max(heights[i] * (widths[i] - 1), maxArea);
    }
    return maxArea;
}
```

Sliding Window Maximum (#239)

Sliding Window Maximum (#239)

- Hint 1: To keep the maximum value for an interval, a stack-like structure could be used. But is stack enough?

Sliding Window Maximum (#239)

- Hint 1: To keep the maximum value for an interval, a stack-like structure could be used. But is stack enough?
- Hint 2: You'll need to remove from bottom of stack – so might be using LinkedList.

Sliding Window Maximum (#239)

- Hint 1: To keep the maximum value for an interval, a stack-like structure could be used. But is stack enough?
- Hint 2: You'll need to remove from bottom of stack – so might be using LinkedList.
- *Hint 3: First, we pop the last element from the window. If that is at the bottom of stack, we remove it as well. Then for a new element, we keep popping the stack until we find an element that's greater than this element. This still takes linear time and the maximum value is preserved.

Sliding Window Maximum (#239) Solution

```java
public int[] maxSlidingWindow(int[] nums, int k) {
    if (nums.length == 0) return new int[]{};

    int[] result = new int[nums.length - k + 1];
    LinkedList<Integer> list = new LinkedList<>();
    for (int i = 0; i < k; i++) {
        while (!list.isEmpty() && list.getLast() < nums[i]) {
            list.removeLast();
        }
        list.addLast(nums[i]);
    }

    for (int i = k; i < nums.length; i++) {
        result[i - k] = list.getFirst();
        int out = nums[i - k];
        if (out == list.getFirst()) {
            list.removeFirst();
        }
        while (!list.isEmpty() && list.getLast() < nums[i]) {
            list.removeLast();
        }
        list.addLast(nums[i]);
    }
    result[nums.length - k] = list.getFirst();
    return result;
}
```

Two Pointers

Two pointers has two sub-types:

- Two-sided pointers: the 2 pointers go from both sides in-wards.
- Fast-slow pointers: 2 pointers go from the same starting point but one moves faster and the other moves slower.
- Outbound pointers: 2 pointers start at the same point but move out-bound.

In general, two pointers are used if there is some condition that you can decide the direction for the pointers to move.

Linked List Cycle (#141)

Linked List Cycle (#141)

- Hint: You can use a fast and a slow pointer. The fast pointer moves 2 nodes each time and the slow pointer moves 1 node each time. If the two pointers meet, it's the cycle.

Linked List Cycle (#141) Solution

```
public boolean hasCycle(ListNode head) {
    ListNode slow = head;
    ListNode fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        if (slow == fast) return true;
    }
    return false;
}
```

Linked List Cycle II (#142)

Linked List Cycle II (#142)

- Hint 1: When two pointers meet, what's the distance between the two pointers traveled?

Linked List Cycle II (#142)

- Hint 1: When two pointers meet, what's the distance between the two pointers traveled?
- Hint 2: The distance is the length of the circle. Then you can ask the fast pointer to move such distance. Then move each pointer by 1 unit. What's the meet point?

Linked List Cycle II (#142) Solution

```java
public ListNode detectCycle(ListNode head) {
    ListNode slow = head;
    ListNode fast = head;
    int length = 0;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        length++;
        if (slow == fast) {
            slow = fast = head;
            for (int i = 0; i < length; i++) {
                fast = fast.next;
            }
            while (slow != fast) {
                slow = slow.next;
                fast = fast.next;
            }
            return slow;
        }
    }
    return null;
}
```

Two Sum II (#167)

Two Sum II (#167)

- Hint: Start from two sides i, j. If $a_i + a_j <$ target, we should increase i. Otherwise, if $a_i + a_j >$ target, we should decrease j.

Two Sum II (#167) Solution

```java
public int[] twoSum(int[] numbers, int target) {
    int start = 0;
    int end = numbers.length - 1;
    while (start < end) {
        int sum = numbers[start] + numbers[end];
        if (sum == target) {
            return new int[]{start + 1, end + 1};
        }
        if (sum < target) start++;
        else end--;
    }
    return new int[]{};
}
```

Container with Most Water (#11)

Container with Most Water (#11)

- Hint 1: For a container with 2 sides l_1 and l_2, the area is min(height[l_1], height[l_2]) * (l2 - l1).

Container with Most Water (#11)

- Hint 1: For a container with 2 sides l_1 and l_2, the area is min(height[l_1], height[l_2]) * (l2 - l1).
- Hint 2: Say l_1 is smaller than l_2, what is the next position for l_1 so that it's area could be greater than current one?

Container with Most Water (#11)

- Hint 1: For a container with 2 sides l_1 and l_2, the area is min(height[l_1], height[l_2]) * (l2 - l1).
- Hint 2: Say l_1 is smaller than l_2, what is the next position for l_1 so that it's area could be greater than current one?
- Hint 3: The next candidate for l_1 can only be those position a_i such that height[a_i] $>$ l_1. Otherwise, the height is smaller / unchanged and the length is reduced. In such case, the area is always smaller.

Container with Most Water (#11) Solution

```
public int maxArea(int[] height) {
    int start = 0;
    int end = height.length - 1;
    int maxSize = 0;
    while (start < end) {
        int h1 = height[start];
        int h2 = height[end];
        int area = Math.min(h1, h2) * (end - start);
        maxSize = Math.max(area, maxSize);
        if (height[end] < height[start]) {
            while (end > start && height[end] <= h2) end--;
        } else {
            while (start < end && height[start] <= h1) start++;
        }
    }
    return maxSize;
}
```

Trapping Rain Water (#42)

Trapping Rain Water (#42)

- Hint 1: For a pair of indices i and j (i < j), if height[i] < height[j] and assume everything between i and j is smaller than i, it's guaranteed that you can scan from i to j and the total volume would be sum(height[i] - height[t]) for t between i and j. Now, if we have j > k > i that height[k] > height[i], what do you do?

Trapping Rain Water (#42)

- Hint 1: For a pair of indices i and j (i < j), if height[i] < height[j] and assume everything between i and j is smaller than i, it's guaranteed that you can scan from i to j and the total volume would be sum(height[i] - height[t]) for t between i and j. Now, if we have j > k > i that height[k] > height[i], what do you do?

- Hint 2: For i < k < j, if height[k] > height[i], then if height[k] < height[j], you can repeat that from k to j otherwise, you can go from j to k. Thus, you have a two pointers algorithm.

Trapping Rain Water (#42) Solution

```
public int trap(int[] height) {
    int start = 0;
    int end = height.length - 1;
    int total = 0;
    while (start < end) {
        if (height[start] < height[end]) {
            int i = start + 1;
            while (i < end && height[i] < height[start]) {
                total += height[start] - height[i];
                i++;
            }
            start = i;
        } else {
            int i = end - 1;
            while (i > start && height[i] < height[end]) {
                total += height[end] - height[i];
                i--;
            }
            end = i;
        }
    }
    return total;
}
```

Parsing

There are a certain category of problems that require you to come up with a mini parser. The parser is usually a recursive descendant parser – most likely LL(1) parser. So you'll need to first make sure you understand the grammar.
If the grammar is left-recursive, that is, the grammar's transitive closure has the form $T = T \mid \ldots$ You'll need to refactor the grammar so that it's not left recursive anymore.
Please read context-free grammar at:
`https://en.wikipedia.org/wiki/LL_parser`

Steps to Solution

For this type of problem, the recommended steps are:

- Understand / summarize the grammar (write the notation down)
- Identify if it's left-recursive, if yes, you'll need to refactor it
- Write the parser

Note: Actually most problems in this section could be solved in a simpler way than the above steps. But doing so will make the solution more systematic and it's good exercise. That being said, the solution is lengthy and you may need to actually come up with a different solution and/or shorter solution in actual interview (not so hard once you already know the grammar).

Decode String (#394)

Decode String (#394)

- Hint 1: Can you write down the grammar?

Decode String (#394)

- Hint 1: Can you write down the grammar?
- Hint 2: The grammar for the string is:

```
S = E | <number>[S]S* | <char>S*
number = (0-9)+
char = [a-zA-Z]

// E: represents empty string
// S*: represents zero or more S
```

Is it left recursive?

Decode String (#394)

- Hint 1: Can you write down the grammar?
- Hint 2: The grammar for the string is:

```
S = E | <number>[S]S* | <char>S*
number = (0-9)+
char = [a-zA-Z]

// E: represents empty string
// S*: represents zero or more S
```

Is it left recursive?

- Hint 3: No! It's not left recursive and you can safely write down the parser recursively.

Decode String (#394) Solution

First, we'll define the representation of the parsed result. First the base class:

```
private abstract class StringNode {
    abstract public void buildString(StringBuilder builder);
}
```

It has one method that basically generates the final string.

Decode String (#394) Solution

Then we have a couple of inherited classes that represents different types of grammar elements:

```
// Represents StringNode with <number>[...]
private class RepeatNode extends StringNode {
    private final int repeat;
    private final StringNode internal;
    public RepeatNode(int n, StringNode node) {
        repeat = n;
        internal = node;
    }
    @Override
    public void buildString(StringBuilder builder) {
        for (int i = 0; i < repeat; i++) {
            internal.buildString(builder);
        }
    }
}
// Represents node with single char.
private class CharNode extends StringNode {
    private final char character;
    public CharNode(char c) {
        character = c;
    }
    @Override
    public void buildString(StringBuilder builder) {
        builder.append(character);
    }
}
```

Decode String (#394) Solution

Since the grammar writes "S*", we'll need to represent zero or more such element. We'll thus need yet another one:

```
// Represents S*
private class SequentialNode extends StringNode {
    private final ArrayList<StringNode> nodes = new ArrayList<>();
    void addNode(StringNode node) {
        nodes.add(node);
    }
    @Override
    public void buildString(StringBuilder builder) {
        for (StringNode n : nodes) {
            n.buildString(builder);
        }
    }
}
```

Decode String (#394) Solution

Then we'll define the `ParsingState` structure that represents the input stream It's pretty standard across different questions.

```
private class ParsingState {
    final String str;
    int offset = 0;
    ParsingState(String s) {
        str = s;
    }
    boolean endOfStream() {
        return offset >= str.length();
    }

    String take(int n) {
        String result = str.substring(offset, offset + n);
        offset += n;
        return result;
    }

    char peek() {
        return str.charAt(offset);
    }

    String takeWhile(Function<Character, Boolean> pred) {
        int end = offset;
        while (end < str.length() && pred.apply(str.charAt(end))) {
            end++;
        }
        String result = str.substring(offset, end);
        offset = end;
        return result;
    }
}
```

Decode String (#394) Solution

Now we define the actual parsing functions:

```
private Integer parseInt(ParsingState s) {
    String ss = s.takeWhile(c -> c >= '0' && c <= '9');
    return Integer.parseInt(ss);
}

private StringNode parseDecodeString(ParsingState s) {
    SequentialNode result = new SequentialNode();
    if (s.endOfStream()) {
        return result;
    }
    char c = s.peek();
    if (c >= '0' && c <= '9') {
        int num = parseInt(s);
        s.take(1);  // skip '['
        StringNode inner = parseDecodeString(s);
        s.take(1);  // skip ']'
        result.addNode(new RepeatNode(num, inner));
    } else if (c == ']') {
        return result;
    } else {
        char ch = s.take(1).charAt(0);
        result.addNode(new CharNode(ch));
    }
    result.addNode(parseDecodeString(s));
    return result;
}

public String decodeString(String s) {
    StringBuilder builder = new StringBuilder();
    parseDecodeString(new ParsingState(s)).buildString(builder);
    return builder.toString();
}
```

Mini Parser (#385)

Mini Parser (#385)

- Hint 1: Can you write down the grammar? Is the grammar left-recursive?

Mini Parser (#385)

- Hint 1: Can you write down the grammar? Is the grammar left-recursive?
- Hint 2: The grammar is:

```
S=[S(,S)*] | <Integer>
Integer=(-|0|1|2|3..|9)+
```

It's not left recursive since you don't have form:

```
S=S|...
```

Mini Parser (#385) Solution

We can re-use our parsing state:

```
private class ParsingState {
    final String str;
    int offset = 0;
    ParsingState(String s) {
        str = s;
    }
    boolean endOfStream() {
        return offset >= str.length();
    }

    String take(int n) {
        String result = str.substring(offset, offset + n);
        offset += n;
        return result;
    }

    char peek() {
        return str.charAt(offset);
    }

    String takeWhile(Function<Character, Boolean> pred) {
        int end = offset;
        while (end < str.length() && pred.apply(str.charAt(end))) {
            end++;
        }
        String result = str.substring(offset, end);
        offset = end;
        return result;
    }

}
```

Mini Parser (#385) Solution

The parsing part is actually quite trivial:

```
private boolean isDigit(char c) {
    // For the purpose of this problem, we can treat - as part of the number.
    return c == '-' || (c >= '0' && c <= '9');
}

private Integer parseInt(ParsingState s) {
    String ss = s.takeWhile(this::isDigit);
    return Integer.parseInt(ss);
}

private NestedInteger doParse(ParsingState state) {
    NestedInteger result = new NestedInteger();
    if (state.endOfStream()) return result;

    char ch = state.peek();
    if (isDigit(ch)) {
        int value = parseInt(state);
        result.setInteger(value);
        return result;
    }
    if (ch == '[') {
        state.take(1);  // ignore [
        while (state.peek() != ']') {
            NestedInteger tmp = doParse(state);
            result.add(tmp);
            if (state.peek() == ',') state.take(1);
        }
        state.take(1);  // ignore ]
    }
    return result;
}

public NestedInteger deserialize(String s) {
    return doParse(new ParsingState(s));
}
```

Number of Atoms (#726)

Number of Atoms (#726)

- Hint 1: Can you write down the grammar? Is it left-recursive?

Number of Atoms (#726)

- Hint 1: Can you write down the grammar? Is it left-recursive?
- Hint 2: The grammar is:

```
Compound = Form+
Form = Atom<Num>? | (Compound)<Num?>
Atom = <Uppercase><lowercase>?
Num = (0-9)+
```

It's not left-recursive.

Number of Atoms (#726) Solution

We can still reuse our ParsingState class:

```
private class ParsingState {
    final String str;
    int offset = 0;
    ParsingState(String s) {
        str = s;
    }
    boolean endOfStream() {
        return offset >= str.length();
    }

    String take(int n) {
        String result = str.substring(offset, offset + n);
        offset += n;
        return result;
    }

    char peek() {
        return str.charAt(offset);
    }

    String takeWhile(Function<Character, Boolean> predicate) {
        int end = offset;
        while (end < str.length() && predicate.apply(str.charAt(end))) {
            end++;
        }
        String result = str.substring(offset, end);
        offset = end;
        return result;
    }
}
```

Number of Atoms (#726) Solution

We'll need a bunch of helpers as well:

```
private boolean isDigit(char c) {
    // For the purpose of this problem, we can treat - as part of the number.
    return c == '-' || (c >= '0' && c <= '9');
}

private Integer parseInt(ParsingState s) {
    String ss = s.takeWhile(this::isDigit);
    return Integer.parseInt(ss);
}

private boolean isUpperCase(char c) {
    return c >= 'A' && c <= 'Z';
}

private boolean isLowerCase(char c) {
    return c >= 'a' && c <= 'z';
}
```

Number of Atoms (#726) Solution

Now we can define our element class. It has an atomCount method that returns the atom count to a hashmap.

```
interface Element {
    void atomCount(HashMap<String, Integer> result);
}
// Single atom:
private class Atom implements Element {
    private final String atom;
    private final int count;
    Atom(String a, int c) {
        atom = a;
        count = c;
    }
    @Override
    public void atomCount(HashMap<String, Integer> result) {
        result.put(atom, result.getOrDefault(atom, 0) + count);
    }
}
// Complex compound (multiple atoms):
private class Form implements Element {
    private final HashMap<Element, Integer> elements = new HashMap<>();
    void putElement(Element e, Integer c) {
        elements.put(e, c);
    }
    @Override
    public void atomCount(HashMap<String, Integer> result) {
        for (Element e : elements.keySet()) {
            HashMap<String, Integer> r = new HashMap<>();
            e.atomCount(r);
            for (String k : r.keySet()) {
                int count = r.get(k) * elements.get(e);
                result.put(k, result.getOrDefault(k, 0) + count);
            }
        }
    }
}
```

Number of Atoms (#726) Solution

The actual parsing code:

```
private Element parseAtom(ParsingState state) {
    String atom = state.take(1) + state.takeWhile(this::isLowerCase);
    String countStr = state.takeWhile(this::isDigit);
    int count = countStr.isEmpty() ? 1 : Integer.parseInt(countStr);
    return new Atom(atom, count);
}
private Element parseCompound(ParsingState state) {
    Form compound = new Form();
    while (!state.endOfStream() && state.peek() != ')') {
        Element form = parseForm(state);
        compound.putElement(form, 1);
    }
    return compound;
}
private Element parseForm(ParsingState state) {
    Form form = new Form();
    if (isUpperCase(state.peek())) {
        Element atom = parseAtom(state);
        form.putElement(atom, 1);
    } else if (state.peek() == '(') {
        state.take(1);  // ignore (
        Element compound = parseCompound(state);
        state.take(1);  // ignore )
        String countStr = state.takeWhile(this::isDigit);
        int count = countStr.isEmpty() ? 1 : Integer.parseInt(countStr);
        form.putElement(compound, count);
    }
    return form;
}
```

Number of Atoms (#726) Solution

Finally, we can write the count function:

```
public String countOfAtoms(String formula) {
    Element compound = parseCompound(new ParsingState(formula));
    HashMap<String, Integer> atomCounts = new HashMap<>();
    compound.atomCount(atomCounts);
    StringBuilder builder = new StringBuilder();
    List<String> keys = new ArrayList<>(atomCounts.keySet());
    Collections.sort(keys);
    for (String k : keys) {
        builder.append(k);
        int count = atomCounts.get(k);
        if (count > 1) builder.append(count);
    }
    return builder.toString();
}
```

Ternary Expression Parser (LintCode #887, LeetCode: #439)

Ternary Expression Parser (LintCode #887, LeetCode: #439)

- Hint 1: What is the grammar? Is it LL(1) or LL(k)? Is it left recursive?

Ternary Expression Parser (LintCode #887, LeetCode: #439)

- Hint 1: What is the grammar? Is it LL(1) or LL(k)? Is it left recursive?
- Hint 2: One possible form of grarmmar is:

```
S = <boolean>(?S:S)?|<Integer>
```

It's not left recursive.

Ternary Expression Parser (LintCode #887, LeetCode: #439) Solution

Let's define the parsed result. It has an eval function that evaluates the syntax tree:

```
interface Term {
    String eval();
}

// Represents boolean or integer.
private class Value implements Term {
    private final String val;
    Value(String v) { val = v; }
    @Override
    public String eval() { return val; }
}

private class Ternary implements Term {
    private final Term condition, trueExpr, falseExpr;
    Ternary(Term cond, Term t, Term f) {
        condition = cond;
        trueExpr = t;
        falseExpr = f;
    }
    @Override
    public String eval() {
        return condition.eval().equals("T") ? trueExpr.eval() : falseExpr.eval();
    }
}
```

Ternary Expression Parser (LintCode #887, LeetCode: #439) Solution

The parse function is actually very simple and straightforward:

```
private Term parse(ParsingState state) {
    if (state.peek() == 'T' || state.peek() == 'F') {
        Value v = new Value(state.take(1));
        if (!state.endOfStream() && state.peek() == '?') {
            state.take(1);  // drop ?
            Term t = parse(state);
            state.take(1); // drop :
            Term f = parse(state);
            return new Ternary(v, t, f);
        }
        return v;
    }
    if (isDigit(state.peek())) {
        Integer i = parseInt(state);
        return new Value(i.toString());
    }
    return null;
}

public String parseTernary(String expression) {
    return parse(new ParsingState(expression)).eval();
}
```