1) [0-1 knapsack w/ exactly capacity] 0-1 knapsack, given weights, values, capacity W, find max value, exactly filling capacity W.

Brute force:
```
knapsack(weights, values, W):
    return max(knapsack_rec(weights, values, W, n))

knapsack_rec(weights, values, W, n):
    # base cases
    if W == 0: return 0
    if n == 0: return -inf
    # recurrence
    if weights[n-1] <= W:
        return max(knapsack_rec(..., W, n-1),
                   values[n-1] + knapsack_rec(..., W-weights[n-1], n-1))
    else:
        return knapsack_rec(..., W, n-1)
```

Top-down:
```
knapsack(weights, values, W):
    memo = {(-1, 0): 0}
    for i in range(len(weights)):
        for w in range(W, weights[i]-1, -1):
            if (i-1, w-weights[i]) in memo:
                memo[(i, w)] = max(memo[(i-1, w)],
                                   values[i] + memo[(i-1, w-weights[i])])
    return max(memo[(i, W)] for i in range(len(weights)))
```

Bottom-up:
```
knapsack(weights, values, W):
    dp = [[-inf]*(W+1) for i in range(len(weights))]
    for i in range(len(weights)):
        dp[i][0] = 0
    for i in range(1, len(weights)):
        for w in range(1, W+1):
            if w >= weights[i]:
                dp[i][w] = max(dp[i-1][w], dp[i-1][w-weights[i]] + values[i])
            else:
                dp[i][w] = dp[i-1][w]
    return dp[n][W]
```

2) [Find max sum in array w/ gaps of size k] Given array, find max sum of elements, no two adjacent elements are more than k apart.

Brute force:
```
maxsum(arr, k):
    return max(maxsum_rec(arr, k, i) for i in range(len(arr)))

maxsum_rec(arr, k, i):
    if i == -1: return 0
    if i == 0: return arr[0]
    if i == 1: return max(arr[0], arr[1])
    return max(maxsum_rec(arr, k, i-1), arr[i] + maxsum_rec(arr, k, i-k-1))
```

Top-down:
```
maxsum(arr, k):
    memo = {-1: 0, 0: arr[0], 1: max(arr[0], arr[1])}
    for i in range(2, len(arr)):
        memo[i] = max(memo[i-1], arr[i] + memo[i-k-1])
    return max(memo[i] for i in range(len(arr)))
```

Bottom-up:
```
maxsum(arr, k):
    dp = [-inf]*len(arr)
    dp[0] = arr[0]
    dp[1] = max(arr[0], arr[1])
    for i in range(2, len(arr)):
        dp[i] = max(dp[i-1], arr[i] + dp[i-k-1])
    return dp[-1]
```

3) [Longest Palindrome] Given a string, find the longest palindrome, return the length of the longest palindrome (100% efficiency, 1ms) runtime of 28 ms, uses 14.8mb

Brute force:
```
longest_palindrome(s):
    if len(s) == 1: return 1
    if len(s) == 2: return 2
    if s == s[::-1]:
        return len(s)
    return max(longest_palindrome(s[1:]), longest_palindrome(s[:-1]))
```

Top down:
```
longest_palindrome(s):
    memo = {1: 1, 2: 2}
    for i in range(3, len(s)+1):
        if s[:i] == s[:i][::-1]:
            memo[i] = i
        else:
            memo[i] = max(memo[i-1], memo[i-1])
    return max(memo.values())
```

Bottom-up:
```
longest_palindrome(s):
    dp = [1]*len(s)
    for i in range(1, len(s)):
        if s[:i+1] == s[:i+1][::-1]:
            dp[i] = i+1
        else:
            dp[i] = max(dp[i-1], dp[i-1])
    return max(dp)
```

4) [Longest Increasing Subsequence] Given an array of integers, find the length of the longest increasing subsequence (100% efficiency) runtime of 28 ms, uses 14.8mb

Brute force:
```
lis(arr):
    if len(arr) == 1: return 1
    if len(arr) == 2: return 2
    if arr == sorted(arr):
        return len(arr)
    return max(lis(arr[1:]), lis(arr[:-1]))
```

Top down:
```
lis(arr):
    memo = {1: 1, 2: 2}
    for i in range(3, len(arr)+1):
        if arr[:i] == sorted(arr[:i]):
            memo[i] = i
        else:
            memo[i] = max(memo[i-1], memo[i-1])
    return max(memo.values())
```

Bottom-up:
```
lis(arr):
    dp = [1]*len(arr)
    for i in range(1, len(arr)):
        if arr[:i+1] == sorted(arr[:i+1]):
            dp[i] = i+1
        else:
            dp[i] = max(dp[i-1], dp[i-1])
    return max(dp)
```

5) [Longest Palindrome] Given a string, find the longest palindrome, return the length of the longest palindrome (100% efficiency) runtime of 28 ms, uses 14.8mb

Brute force:
```
longest_palindrome(s):
    if len(s) == 1: return 1
    if len(s) == 2: return 2
    if s == s[::-1]:
        return len(s)
    return max(longest_palindrome(s[1:]), longest_palindrome(s[:-1]))
```

Top down:
```
longest_palindrome(s):
    memo = {1: 1, 2: 2}
    for i in range(3, len(s)+1):
        if s[:i] == s[:i][::-1]:
            memo[i] = i
        else:
            memo[i] = max(memo[i-1], memo[i-1])
    return max(memo.values())
```

Bottom-up:
```
longest_palindrome(s):
    dp = [1]*len(s)
    for i in range(1, len(s)):
        if s[:i+1] == s[:i+1][::-1]:
            dp[i] = i+1
        else:
            dp[i] = max(dp[i-1], dp[i-1])
    return max(dp)
```

6) [Longest Increasing Subsequence] Given an array of integers, find the length of the longest increasing subsequence (100% efficiency) runtime of 28 ms, uses 14.8mb

Brute force:
```
lis(arr):
    if len(arr) == 1: return 1
    if len(arr) == 2: return 2
    if arr == sorted(arr):
        return len(arr)
    return max(lis(arr[1:]), lis(arr[:-1]))
```

Top down:
```
lis(arr):
    memo = {1: 1, 2: 2}
    for i in range(3, len(arr)+1):
        if arr[:i] == sorted(arr[:i]):
            memo[i] = i
        else:
            memo[i] = max(memo[i-1], memo[i-1])
    return max(memo.values())
```

Bottom-up:
```
lis(arr):
    dp = [1]*len(arr)
    for i in range(1, len(arr)):
        if arr[:i+1] == sorted(arr[:i+1]):
            dp[i] = i+1
        else:
            dp[i] = max(dp[i-1], dp[i-1])
    return max(dp)
```

7) [Longest Palindrome] Given a string, find the longest palindrome, return the length of the longest palindrome (100% efficiency) runtime of 28 ms, uses 14.8mb

Brute force:
```
longest_palindrome(s):
    if len(s) == 1: return 1
    if len(s) == 2: return 2
    if s == s[::-1]:
        return len(s)
    return max(longest_palindrome(s[1:]), longest_palindrome(s[:-1]))
```

Top down:
```
longest_palindrome(s):
    memo = {1: 1, 2: 2}
    for i in range(3, len(s)+1):
        if s[:i] == s[:i][::-1]:
            memo[i] = i
        else:
            memo[i] = max(memo[i-1], memo[i-1])
    return max(memo.values())
```

Bottom-up:
```
longest_palindrome(s):
    dp = [1]*len(s)
    for i in range(1, len(s)):
        if s[:i+1] == s[:i+1][::-1]:
            dp[i] = i+1
        else:
            dp[i] = max(dp[i-1], dp[i-1])
    return max(dp)
```

8) [Longest Increasing Subsequence] Given an array of integers, find the length of the longest increasing subsequence (100% efficiency) runtime of 28 ms, uses 14.8mb

Brute force:
```
lis(arr):
    if len(arr) == 1: return 1
    if len(arr) == 2: return 2
    if arr == sorted(arr):
        return len(arr)
    return max(lis(arr[1:]), lis(arr[:-1]))
```

Top down:
```
lis(arr):
    memo = {1: 1, 2: 2}
    for i in range(3, len(arr)+1):
        if arr[:i] == sorted(arr[:i]):
            memo[i] = i
        else:
            memo[i] = max(memo[i-1], memo[i-1])
    return max(memo.values())
```

Bottom-up:
```
lis(arr):
    dp = [1]*len(arr)
    for i in range(1, len(arr)):
        if arr[:i+1] == sorted(arr[:i+1]):
            dp[i] = i+1
        else:
            dp[i] = max(dp[i-1], dp[i-1])
    return max(dp)
```

9) [Longest Palindrome] Given a string, find the longest palindrome, return the length of the longest palindrome (100% efficiency) runtime of 28 ms, uses 14.8mb

Brute force:
```
longest_palindrome(s):
    if len(s) == 1: return 1
    if len(s) == 2: return 2
    if s == s[::-1]:
        return len(s)
    return max(longest_palindrome(s[1:]), longest_palindrome(s[:-1]))
```

Top down:
```
longest_palindrome(s):
    memo = {1: 1, 2: 2}
    for i in range(3, len(s)+1):
        if s[:i] == s[:i][::-1]:
            memo[i] = i
        else:
            memo[i] = max(memo[i-1], memo[i-1])
    return max(memo.values())
```

Bottom-up:
```
longest_palindrome(s):
    dp = [1]*len(s)
    for i in range(1, len(s)):
        if s[:i+1] == s[:i+1][::-1]:
            dp[i] = i+1
        else:
            dp[i] = max(dp[i-1], dp[i-1])
    return max(dp)
```

10) [Longest Palindrome] Given a string, find the longest palindrome, return the length of the longest palindrome (100% efficiency) runtime of 28 ms, uses 14.8mb

Brute force:
```
longest_palindrome(s):
    if len(s) == 1: return 1
    if len(s) == 2: return 2
    if s == s[::-1]:
        return len(s)
    return max(longest_palindrome(s[1:]), longest_palindrome(s[:-1]))
```

Top down:
```
longest_palindrome(s):
    memo = {1: 1, 2: 2}
    for i in range(3, len(s)+1):
        if s[:i] == s[:i][::-1]:
            memo[i] = i
        else:
            memo[i] = max(memo[i-1], memo[i-1])
    return max(memo.values())
```

Bottom-up:
```
longest_palindrome(s):
    dp = [1]*len(s)
    for i in range(1, len(s)):
        if s[:i+1] == s[:i+1][::-1]:
            dp[i] = i+1
        else:
            dp[i] = max(dp[i-1], dp[i-1])
    return max(dp)
```

11) [Longest Palindrome] Given a string, find the longest palindrome, return the length of the longest palindrome (100% efficiency) runtime of 28 ms, uses 14.8mb

Brute force:
```
longest_palindrome(s):
    if len(s) == 1: return 1
    if len(s) == 2: return 2
    if s == s[::-1]:
        return len(s)
    return max(longest_palindrome(s[1:]), longest_palindrome(s[:-1]))
```

Top down:
```
longest_palindrome(s):
    memo = {1: 1, 2: 2}
    for i in range(3, len(s)+1):
        if s[:i] == s[:i][::-1]:
            memo[i] = i
        else:
            memo[i] = max(memo[i-1], memo[i-1])
    return max(memo.values())
```

Bottom-up:
```
longest_palindrome(s):
    dp = [1]*len(s)
    for i in range(1, len(s)):
        if s[:i+1] == s[:i+1][::-1]:
            dp[i] = i+1
        else:
            dp[i] = max(dp[i-1], dp[i-1])
    return max(dp)
```

12) [Longest Palindrome] Given a string, find the longest palindrome, return the length of the longest palindrome (100% efficiency) runtime of 28 ms, uses 14.8mb

Brute force:
```
longest_palindrome(s):
    if len(s) == 1: return 1
    if len(s) == 2: return 2
    if s == s[::-1]:
        return len(s)
    return max(longest_palindrome(s[1:]), longest_palindrome(s[:-1]))
```

Top down:
```
longest_palindrome(s):
    memo = {1: 1, 2: 2}
    for i in range(3, len(s)+1):
        if s[:i] == s[:i][::-1]:
            memo[i] = i
        else:
            memo[i] = max(memo[i-1], memo[i-1])
    return max(memo.values())
```

Bottom-up:
```
longest_palindrome(s):
    dp = [1]*len(s)
    for i in range(1, len(s)):
        if s[:i+1] == s[:i+1][::-1]:
            dp[i] = i+1
        else:
            dp[i] = max(dp[i-1], dp[i-1])
    return max(dp)
```

13) [Longest Palindrome] Given a string, find the longest palindrome, return the length of the longest palindrome (100% efficiency) runtime of 28 ms, uses 14.8mb

Brute force:
```
longest_palindrome(s):
    if len(s) == 1: return 1
    if len(s) == 2: return 2
    if s == s[::-1]:
        return len(s)
    return max(longest_palindrome(s[1:]), longest_palindrome(s[:-1]))
```

Top down:
```
longest_palindrome(s):
    memo = {1: 1, 2: 2}
    for i in range(3, len(s)+1):
        if s[:i] == s[:i][::-1]:
            memo[i] = i
        else:
            memo[i] = max(memo[i-1], memo[i-1])
    return max(memo.values())
```

Bottom-up:
```
longest_palindrome(s):
    dp = [1]*len(s)
    for i in range(1, len(s)):
        if s[:i+1] == s[:i+1][::-1]:
            dp[i] = i+1
        else:
            dp[i] = max(dp[i-1], dp[i-1])
    return max(dp)
```

14) [Longest Palindrome] Given a string, find the longest palindrome, return the length of the longest palindrome (100% efficiency) runtime of 28 ms, uses 14.8mb

Brute force:
```
longest_palindrome(s):
    if len(s) == 1: return 1
    if len(s) == 2: return 2
    if s == s[::-1]:
        return len(s)
    return max(longest_palindrome(s[1:]), longest_palindrome(s[:-1]))
```

Top down:
```
longest_palindrome(s):
    memo = {1: 1, 2: 2}
    for i in range(3, len(s)+1):
        if s[:i] == s[:i][::-1]:
            memo[i] = i
        else:
            memo[i] = max(memo[i-1], memo[i-1])
    return max(memo.values())
```

Bottom-up:
```
longest_palindrome(s):
    dp = [1]*len(s)
    for i in range(1, len(s)):
        if s[:i+1] == s[:i+1][::-1]:
            dp[i] = i+1
        else:
            dp[i] = max(dp[i-1], dp[i-1])
    return max(dp)
```

15) [Longest Palindrome] Given a string, find the longest palindrome, return the length of the longest palindrome (100% efficiency) runtime of 28 ms, uses 14.8mb

Brute force:
```
longest_palindrome(s):
    if len(s) == 1: return 1
    if len(s) == 2: return 2
    if s == s[::-1]:
        return len(s)
    return max(longest_palindrome(s[1:]), longest_palindrome(s[:-1]))
```

Top down:
```
longest_palindrome(s):
    memo = {1: 1, 2: 2}
    for i in range(3, len(s)+1):
        if s[:i] == s[:i][::-1]:
            memo[i] = i
        else:
            memo[i] = max(memo[i-1], memo[i-1])
    return max(memo.values())
```

Bottom-up:
```
longest_palindrome(s):
    dp = [1]*len(s)
    for i in range(1, len(s)):
        if s[:i+1] == s[:i+1][::-1]:
            dp[i] = i+1
        else:
            dp[i] = max(dp[i-1], dp[i-1])
    return max(dp)
```

16) [Longest Palindrome] Given a string, find the longest palindrome, return the length of the longest palindrome (100% efficiency) runtime of 28 ms, uses 14.8mb

Brute force:
```
longest_palindrome(s):
    if len(s) == 1: return 1
    if len(s) == 2: return 2
    if s == s[::-1]:
        return len(s)
    return max(longest_palindrome(s[1:]), longest_palindrome(s[:-1]))
```

Top down:
```
longest_palindrome(s):
    memo = {1: 1, 2: 2}
    for i in range(3, len(s)+1):
        if s[:i] == s[:i][::-1]:
            memo[i] = i
        else:
            memo[i] = max(memo[i-1], memo[i-1])
    return max(memo.values())
```

Bottom-up:
```
longest_palindrome(s):
    dp = [1]*len(s)
    for i in range(1, len(s)):
        if s[:i+1] == s[:i+1][::-1]:
            dp[i] = i+1
        else:
            dp[i] = max(dp[i-1], dp[i-1])
    return max(dp)
```

17) [Longest Palindrome] Given a string, find the longest palindrome, return the length of the longest palindrome (100% efficiency) runtime of 28 ms, uses 14.8mb

Brute force:
```
longest_palindrome(s):
    if len(s) == 1: return 1
    if len(s) == 2: return 2
    if s == s[::-1]:
        return len(s)
    return max(longest_palindrome(s[1:]), longest_palindrome(s[:-1]))
```

Top down:
```
longest_palindrome(s):
    memo = {1: 1, 2: 2}
    for i in range(3, len(s)+1):
        if s[:i] == s[:i][::-1]:
            memo[i] = i
        else:
            memo[i] = max(memo[i-1], memo[i-1])
    return max(memo.values())
```

Bottom-up:
```
longest_palindrome(s):
    dp = [1]*len(s)
    for i in range(1, len(s)):
        if s[:i+1] == s[:i+1][::-1]:
            dp[i] = i+1
        else:
            dp[i] = max(dp[i-1], dp[i-1])
    return max(dp)
```

18) [Longest Palindrome] Given a string, find the longest palindrome, return the length of the longest palindrome (100% efficiency) runtime of 28 ms, uses 14.8mb

Brute force:
```
longest_palindrome(s):
    if len(s) == 1: return 1
    if len(s) == 2: return 2
    if s == s[::-1]:
        return len(s)
    return max(longest_palindrome(s[1:]), longest_palindrome(s[:-1]))
```

Top down:
```
longest_palindrome(s):
    memo = {1: 1, 2: 2}
    for i in range(3, len(s)+1):
        if s[:i] == s[:i][::-1]:
            memo[i] = i
        else:
            memo[i] = max(memo[i-1], memo[i-1])
    return max(memo.values())
```

Bottom-up:
```
longest_palindrome(s):
    dp = [1]*len(s)
    for i in range(1, len(s)):
        if s[:i+1] == s[:i+1][::-1]:
            dp[i] = i+1
        else:
            dp[i] = max(dp[i-1], dp[i-1])
    return max(dp)
```

19) [Longest Palindrome] Given a string, find the longest palindrome, return the length of the longest palindrome (100% efficiency) runtime of 28 ms, uses 14.8mb

Brute force:
```
longest_palindrome(s):
    if len(s) == 1: return 1
    if len(s) == 2: return 2
    if s == s[::-1]:
        return len(s)
    return max(longest_palindrome(s[1:]), longest_palindrome(s[:-1]))
```

Top down:
```
longest_palindrome(s):
    memo = {1: 1, 2: 2}
    for i in range(3, len(s)+1):
        if s[:i] == s[:i][::-1]:
            memo[i] = i
        else:
            memo[i] = max(memo[i-1], memo[i-1])
    return max(memo.values())
```

Bottom-up:
```
longest_palindrome(s):
    dp = [1]*len(s)
    for i in range(1, len(s)):
        if s[:i+1] == s[:i+1][::-1]:
            dp[i] = i+1
        else:
            dp[i] = max(dp[i-1], dp[i-1])
    return max(dp)
```

20) [Longest Palindrome] Given a string, find the longest palindrome, return the length of the longest palindrome (100% efficiency) runtime of 28 ms, uses 14.8mb

Brute force:
```
longest_palindrome(s):
    if len(s) == 1: return 1
    if len(s) == 2: return 2
    if s == s[::-1]:
        return len(s)
    return max(longest_palindrome(s[1:]), longest_palindrome(s[:-1]))
```

Top down:
```
longest_palindrome(s):
    memo = {1: 1, 2: 2}
    for i in range(3, len(s)+1):
        if s[:i] == s[:i][::-1]:
            memo[i] = i
        else:
            memo[i] = max(memo[i-1], memo[i-1])
    return max(memo.values())
```

Bottom-up:
```
longest_palindrome(s):
    dp = [1]*len(s)
    for i in range(1, len(s)):
        if s[:i+1] == s[:i+1][::-1]:
            dp[i] = i+1
        else:
            dp[i] = max(dp[i-1], dp[i-1])
    return max(dp)
```