```python
import numpy as np
import sys


def zero_one_knapsack(values, weights, capacity):
    n = len(values)
    # costs = [[0] * (capacity + 1) for _ in range(n)]
    c = np.zeros((n, capacity + 1))

    for i in range(0, n):
        for j in range(0, capacity + 1):
            if weights[i] > j:
                c[i][j] = c[i - 1][j]
            else:
                c[i][j] = max(c[i - 1][j], values[i] + c[i - 1][j - weights[i]])

    return [c[n - 1][capacity], get_used_items(weights, c)]


# w = list of item weight or cost
# c = the cost matrix created by the dynamic programming solution
def get_used_items(weights, c):
    i = len(c) - 1
    current_w = len(c[0]) - 1

    # set everything to not marked
    marked = [0 for _ in range(i + 1)]

    while i >= 0 and current_w >= 0:
        if (i == 0 and c[i][current_w] > 0) or c[i][current_w] != c[i - 1][current_w]:
            marked[i] = 1
            current_w = current_w - weights[i]
        i -= 1

    return marked


def main():
    if len(sys.argv) != 3:
        print("Usage knapsack.py name1-weight1-val1,name2-weight2-val2,... max weight")
        print("Example:")
        print("knapsack.py iphone-1-500,clock-1-40,laptop-3-2000,guitar-2-1500 4")
        quit()

    items = sys.argv[1].split(',')
    w = []
    v = []
    names = []

    for item in items:
        nums = item.split('-')
        names.append(nums[0])
        w.append(int(nums[1]))
        v.append(int(nums[2]))

    max_cost = int(sys.argv[2])
    answer = zero_one_knapsack(v, w, max_cost)
    print("Knapsack can hold %d pounds, for $%d profit." % (max_cost, answer[0]))
    print("by taking item(s): ")

    for i in range(len(answer[1])):
        if answer[1][i] != 0:
            print(" - " + names[i])


if __name__ == '__main__':
    main()
```

File - /Users/john/Projects/code-catalog-python/catalog/suggested/dynamic_programming/matrix_chain.py

```python
def matrix_chain(d):
    """Return solution to the matrix chain problem.

    d is a list of n+1 numbers describing the dimensions of a chain of
    n matrices such that kth matrix has dimensions d[k]-by-d[k+1].

    Return an n-by-n table such that N[i][j] represents the minimum number of
    multiplications needed to compute the product of Ai through Aj inclusive.
    """
    n = len(d) - 1  # number of matrices
    N = [[0] * n for i in range(n)]  # initialize n-by-n result to zero
    for b in range(1, n):  # number of products in subchain
        for i in range(n - b):  # start of subchain
            j = i + b  # end of subchain
            N[i][j] = min(N[i][k] + N[k + 1][j] + d[i] * d[k + 1] * d[j + 1]
                          for k in range(i, j))
    return N
```

```
"""
Calculates the edit distance in 2 strings
"""


def get_edit_distance(str1, str2):
    len1 = len(str1)
    len2 = len(str2)

    distances = [[0] * (len1 + 1) for _ in range(len2 + 1)]

    edit_distance = get_edit_distance_util(str1, str2, distances, len1, len2)

    return edit_distance


def get_edit_distance_util(str1, str2, distances, len1, len2):
    for i in range(1, len2 + 1):
        distances[i][0] = i

    for j in range(1, len1 + 1):
        distances[0][j] = j

    for i in range(1, len2 + 1):
        for j in range(1, len1 + 1):
            if str1[j - 1] == str2[i - 1]:
                distances[i][j] = distances[i - 1][j - 1]
            else:
                deletion = distances[i][j - 1] + 1
                insertion = distances[i - 1][j] + 1
                substitution = distances[i - 1][j - 1] + 1
                distances[i][j] = min(deletion, insertion, substitution)

    return distances[len2][len1]


def main():
    assert get_edit_distance("abc", "abcd") == 1
    assert get_edit_distance("abc", "abc") == 0
    assert get_edit_distance("", "") == 0
    assert get_edit_distance("", "a") == 1
    assert get_edit_distance("", "abcde") == 5
    assert get_edit_distance("a", "") == 1
    assert get_edit_distance("a", "ab") == 1
    assert get_edit_distance("a", "bc") == 2
    assert get_edit_distance("abcdef", "fedcba") == 6


if __name__ == '__main__':
    main()
```

```python
"""
Compute the maximum subvector of an array of numbers
"""


def maximum_subvector(nums):
    max_so_far = 0
    max_ending_here = 0

    for n in nums:
        max_ending_here = max(max_ending_here + n, 0)
        max_so_far = max(max_so_far, max_ending_here)

    return max_so_far


def main():
    nums = [7, -4, 3, 5, -2, 1, 0, -12, 3, 5, -1, 4, -8]

    max_sub = maximum_subvector(nums)
    assert max_sub == 11


if __name__ == '__main__':
    main()
```

```
"""
Longest common substring
"""


def longest_common_substring(s1, s2):
    m = [[0] * (1 + len(s2)) for _ in range(1 + len(s1))]

    longest, x_longest = 0, 0

    for x in range(1, 1 + len(s1)):
        for y in range(1, 1 + len(s2)):
            if s1[x - 1] == s2[y - 1]:
                m[x][y] = m[x - 1][y - 1] + 1
                if m[x][y] > longest:
                    longest = m[x][y]
                    x_longest = x
            else:
                m[x][y] = 0

    return s1[x_longest - longest: x_longest]


def main():
    assert longest_common_substring('my family', "I'm famous") == ' fam'
    assert longest_common_substring('hello world', "shell corp") == 'hell'
    assert longest_common_substring('just in time', "halifax") == 'i'
    assert longest_common_substring('abc', "xyz") == ''
    assert longest_common_substring('superman', 'wonder woman') == 'man'


if __name__ == '__main__':
    main()
```

```python
"""
Longest common subsequence
"""


def longest_common_subsequence(s1, s2):
    lengths = [[0 for _ in range(len(s2) + 1)] for _ in range(len(s1) + 1)]

    # row 0 and column 0 are initialized to 0 already
    for i, x in enumerate(s1):
        for j, y in enumerate(s2):
            if x == y:
                lengths[i + 1][j + 1] = lengths[i][j] + 1
            else:
                lengths[i + 1][j + 1] = max(lengths[i + 1][j], lengths[i][j + 1])

    # read the substring out from the matrix
    result = []
    x, y = len(s1), len(s2)
    while x != 0 and y != 0:
        if lengths[x][y] == lengths[x - 1][y]:
            x -= 1
        elif lengths[x][y] == lengths[x][y - 1]:
            y -= 1
        else:
            assert s1[x - 1] == s2[y - 1]
            result.append(s1[x - 1])
            x -= 1
            y -= 1

    return ''.join(reversed(result))


def main():
    assert longest_common_subsequence('my family', "I'm famous") == 'm fam'
    assert longest_common_subsequence('hello world', "shell corp") == 'hell or'
    assert longest_common_subsequence('just in time, max', "halifax") == 'iax'
    assert longest_common_subsequence('abc', "xyz") == ''
    assert longest_common_subsequence('superman', 'wonder woman') == 'erman'


if __name__ == '__main__':
    main()
```