```
from itertools import combinations

def knapsack_bruteforce(capacity, items):
    n = len(items)
    max_profit = 0

    for r in range(1, n + 1):
        for combination in combinations(items, r):
            total_weight = sum(item[1] for item in combination)
            total_value = sum(item[0] for item in combination)

            if total_weight <= capacity:
                max_profit = max(max_profit, total_value)

    return max_profit

with open('items.txt', 'r') as f:
    C, N = map(int, f.readline().split())
    data = []
    for _ in range(N):
        value, weight = map(int, f.readline().split())
        data.append((value, weight))

print(f'Размер рюкзака на усложненной выборке: {C}\nКол-во предметов на усложненной выборке: {N}\n{data}')

def knapsack_pack(C, items):

    n = len(items)

    pack = [[0] * (C + 1) for _ in range(n + 1)] # Создаем pack таблицу размерами (n + 1) x (C + 1)

    for i in range(1, n + 1):
```

```
        value, weight = items[i - 1]

        for w in range(C + 1):
            if weight > w:
                pack[i][w] = pack[i - 1][w]
            else:
                pack[i][w] = max(pack[i - 1][w], pack[i - 1][w - weight] + value)

    return pack[n][C]

start_time = time.time()
max_value = knapsack_pack(C, data)
end_time = time.time()

print(f'Время выполнения: {end_time - start_time}')
print(f'Максимальная сумма значений предметов: {max_value}')

def knapsack_recursive(capacity, items, index, memo):

    if index == 0 or capacity == 0:
        return 0

    if (index, capacity) in memo:
        return memo[(index, capacity)]

    value, weight = items[index - 1]

    if weight > capacity:
        memo[(index, capacity)] = knapsack_recursive(capacity, items, index - 1, memo)
    else:
        memo[(index, capacity)] = max(knapsack_recursive(capacity, items, index - 1, memo),  # не берем предмет
            value + knapsack_recursive(capacity - weight, items, index - 1, memo)  # берем предмет
        )
```

```
    return memo[(index, capacity)]

def knapsack(capacity, items):
    memo = {}
    return knapsack_recursive(capacity, items, len(items), memo)

start_time = time.time()
max_value = knapsack(C, data)
end_time = time.time()

print(f'Время выполнения: {end_time - start_time}')
print(f'Максимальная сумма значений предметов: {max_value}')

import heapq

'''
Node представляет узел дерева решений.
Каждый узел указывает, включён ли текущий предмет (на уровне level) в рюкзак или нет.

self.level = level        # Уровень (индекс) текущего предмета
self.profit = profit      # Текущая ценность (сумма значений предметов)
self.weight = weight      # Текущий вес рюкзака
self.bound = bound        # Верхняя оценка возможной максимальной ценности

__lt__ — определяет порядок узлов в очереди с приоритетом.
Мы используем максимальную bound для приоритетной обработки узлов (max-куча).
'''

class Node:
    def __init__(self, level, profit, weight, bound):
        self.level = level
        self.profit = profit
        self.weight = weight
```

```
        self.bound = bound

    def __lt__(self, other):
        return self.bound > other.bound

'''
bound оценивает максимальную возможную ценность рюкзака из текущего состояния узла node.

В цикле while добавляем предметы, пока их общий вес не превышает вместимость рюкзака.

Если ещё есть место, добавляем долю следующего предмета. Она используется исключительно для оценки потенциала узла,
доля служит способом приближения потенциально достижимой максимальной ценности.
'''

def bound(node, n, capacity, items):

    if node.weight >= capacity:
        return 0

    profit_bound = node.profit
    j = node.level + 1
    total_weight = node.weight

    while j < n and total_weight + items[j][1] <= capacity:
        total_weight += items[j][1]
        profit_bound += items[j][0]
        j += 1

    if j < n:
        profit_bound += (capacity - total_weight) * items[j][0] / items[j][1]

    #print(node.level, node.profit, node.weight, profit_bound)
```

```
    return profit_bound

'''
Задача о рюкзаке с использованием метода ветвей и границ (branch and bound),
на основе очереди с приоритетами для отсечения невыгодных путей.

1. Мы сортируем items по убыванию отношения ценность/вес (чтобы быстрее достигать наибольшей ценности).
2. Создаем начальный узел и добавляем его в очередь.
3. Извлекается узел с наибольшей верхней границей, чтобы рассмотреть узлы,
которые потенциально могут привести к более высокому profit.
4. Создаем дочерний узел (следующий уровень и добавляем профит, вес).
5. Проверяем и обновляем максимальной ценность.
6. Если верхняя граница больше текущего max_profit, узел добавляется в очередь для дальнейшего рассмотрения.
7. Если предмет не включается, то создаётся альтернативный узел, где  profit и weight остаются такими же, как у узла u.
Если верхняя граница v.bound этого узла выше текущего max_profit, узел добавляется в очередь.
'''

def knapsack_branch_and_bound(capacity, items):

    n = len(items)
    items = sorted(items, key=lambda x: x[0] / x[1], reverse=True)

    queue = []
    u = Node(-1, 0, 0, 0)
    u.bound = bound(u, n, capacity, items)
    heapq.heappush(queue, u)

    max_profit = 0

    while queue:
        u = heapq.heappop(queue)
```

```
        if u.level == n - 1 or u.bound <= max_profit:
            continue

        v = Node(u.level + 1,
                 u.profit + items[u.level + 1][0],
                 u.weight + items[u.level + 1][1],
                 0)

        if v.weight <= capacity and v.profit > max_profit:
            max_profit = v.profit

        v.bound = bound(v, n, capacity, items)
        if v.bound > max_profit:
            heapq.heappush(queue, v)

        v = Node(u.level + 1, u.profit, u.weight, 0)
        v.bound = bound(v, n, capacity, items)
        if v.bound > max_profit:
            heapq.heappush(queue, v)

    return max_profit

start_time = time.time()
max_value = knapsack_branch_and_bound(C, data)
end_time = time.time()

print(f'Время выполнения на данных: {end_time - start_time}')
print(f'Максимальная сумма значений предметов на данных: {max_value}')
```