Санкт-Петербургский Национальный
Исследовательский Университет
Информационных технологий, механики и оптики
Факультет ИКТ

Отчет по лабораторной работе №1

"Жадные алгоритмы. Динамическое программирование"

Студент: Кузнецов
Никита Сергеевич
1 курс
Группа №K3120
Преподаватель: Харьковская
Татьяна Александровна

Санкт-Петербург
2022

Задача 2

Условие:

Вы собираетесь поехать в другой город, расположенный в d км от вашего родного города. Ваш автомобиль может проехать не более m км на полном баке, и вы начинаете с полным баком. По пути есть заправочные станции на расстояниях $\text{stop}_1, stop_2, ..., stop_n$.?

Решение:

Написал простую функцию, где в начале обрабатываю ввод:

```
def how_many_stops(data: list[str]) -> int:
    dist = int(data[0])
    can_drive = int(data[1])
    stops = list(map(int, data[3].split()))
```

Затем, проверяю условие если автомобиль может проехать все расстояние без остановок, а также условие при котором расстояние между остановками больше возможностей бака автомобиля:

```
    if can_drive >= dist:
        return 0

    for i in range(len(stops) - 1):
        if stops[i + 1] - stops[i] > can_drive:
            return -1
```

Ну и в конце, использую жадный алгоритм для того, чтобы посчитать количество раз, которое нужно остановиться автомобилю.

```
    counter = 0
    drive_dist = can_drive
    while can_drive <= dist:
        for i, stop in enumerate(stops):
            if stop > can_drive:
                counter += 1
                stops = stops[i:]
                break
        can_drive += drive_dist
    else:
        last_stop = stops[-1]

    if last_stop < dist:
        counter += 1

    return counter
```

С каждым шагом увеличиваю расстояние, на которое автомобиль может проехать и если он не может доехать до какой-то точки - то прибавляю одну остановку. В самом конце проверяю, чтобы последняя остановка не была до конца маршрута, и если это так - то прибавляю еще одну.

Тесты:

Написал свои тесты для этой задачи. Первые три basic_tests проверяют условия из задачи, а три других придумал сам.

```
class TaskTest(unittest.TestCase):

    def test_basic(self):
```

```
        case1 = "950\n" \
            "400\n" \
            "4\n" \
            "200 375 550 750".splitlines()

        case2 = "10\n" \
                "3\n" \
                "4\n" \
                "1 2 5 9".splitlines()

        case3 = "200\n" \
                "250\n" \
                "2\n" \
                "100 150".splitlines()

        self.assertEqual(how_many_stops(case1), 2, "should be 2")
        self.assertEqual(how_many_stops(case2), -1, "should be -1")
        self.assertEqual(how_many_stops(case3), 0, "should be 0")

    def test_custom(self):
        case1 = "1000\n" \
                "355\n" \
                "5\n" \
                "250 500 750 900".splitlines()

        case2 = "60\n" \
                "10\n" \
                "6\n" \
                "1 11 21 31 41 51".splitlines()

        case3 = "5\n" \
                "1\n" \
                "4\n" \
                "1 2 3 4".splitlines()

        self.assertEqual(how_many_stops(case1), 3, "should be 3")
        self.assertEqual(how_many_stops(case2), 6, "should be 6")
        self.assertEqual(how_many_stops(case3), 4, "should be 4")
```

Результаты теста:

```
Ran 2 tests in 0.002s

OK
```

Вывод:

В ходе решения этой задаче я смог имплементировать простой жадный алгоритм для нахождения минимального количества остановок для автомобиля.

Задача 5

Условие:

Вы организуете веселый конкурс для детей. В качестве призового фонда у вас есть n конфет. Вы хотели бы использовать эти конфеты для раздачи k лучшим местам в конкурсе с естественным ограничением, заключающимся в том, что чем выше место, тем больше конфет. Чтобы осчастливить как можно больше детей, вам нужно найти наибольшее значение k, для которого это возможно.

Решение:

Написал функцию, принимающую на вход количество конфет.

```
def max_prize_amount(candies: int) -> tuple[int, list[int]]:

    if candies < 2:
        return 1, [candies]
```

Если конфет меньше 2, то возвращаю 1 и список, состоящий из одного элемента - кол-ва конфет.
Иначе, создаю список с конфетами, который начинается с единицы, уменьшаю количество конфет на эту единицу. Затем в цикле проверяю, чтобы количество конфет было больше последней добавленной конфеты, и если это так, то добавляю в массив количество, большее этого на единицу, а общее количество конфет уменьшаю на это количество.

```
    candy_per_child = [1]
    candies -= 1

    while candies > candy_per_child[-1]:
        new_candies = candy_per_child[-1] + 1
        candy_per_child.append(new_candies)
        candies -= new_candies

    candy_per_child[-1] += candies

    return len(candy_per_child), candy_per_child
```

В конце добавляю остаток в последний элемент списка и возвращаю сам список и его длину. Для вывода преобразую эти числа в нужный формат:

```
    with open("input.txt", 'r') as f:
    datafile = f.readlines()

    result1, result2 = max_prize_amount(int(datafile[0]))
    with open("output.txt", 'w') as f:
        f.write(f"{str(result1)}\n{' '.join(list(map(str, result2)))}")
```

Тесты:

Написал свои тесты для этой задачи. Первые три - из условия задачи, три других придумал сам.

```
class TaskTest(unittest.TestCase):

    def test_basic(self):
        self.assertEqual(max_prize_amount(6), (3, [1, 2, 3]), "OK")
        self.assertEqual(max_prize_amount(8), (3, [1, 2, 5]), "OK")
        self.assertEqual(max_prize_amount(2), (1, [2]), "OK")

    def test_custom(self):
        self.assertEqual(max_prize_amount(12), (4, [1, 2, 3, 6]), "OK")
        self.assertEqual(max_prize_amount(1), (1, [1]), "OK")
        self.assertEqual(max_prize_amount(25), (6, [1, 2, 3, 4, 5, 10]), "OK")


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

Программа проходит эти тесты:

```
Ran 2 tests in 0.002s

OK
```

Вывод:

Применив жадный алгоритм, получилось провести хороший и качественный праздник для детей, посчитав нужное количество конфет для каждого места.

Задача 12

Условие:

Дана последовательность натуральных чисел a1, a2, ..., an, и известно, что ai i для любого 1 i n. Требуется определить, мож- но ли разбить элементы последовательности на две части таким образом, что сумма элементов в каждой из частей будет равна половине суммы всех элементов последовательности.

Решение:

Начал с создания функции, принимающей на вход массив чисел.

```
def half_subset(array: list[int]) -> tuple[int, list[int]]:
    if sum(array) % 2 > 0:
        return -1, []
```

Если длина массива - это нечетное число, то возвращаем -1 и пустой список, так как поделить такой массив поровну нельзя. Затем записываем нужные нам переменные: число, равняющееся половине суммы чисел массива, копию массива и массив индексов (для вывода):

```
    half = sum(array) // 2
    array_copy = array.copy()
    index_array = []
```

Затем, проходимся по массиву и сравниваем его последний элемент с половиной. Если он меньше или равен ей - то вычитаем его из половины и добавляем индекс в список индексов. И в конце каждого прохода срезаем последний элемент массива.

```
    for i, _ in enumerate(array):
        check = array_copy[-1]
        if check <= half:
            half -= check
            index_array.append(array.index(check) + 1)
        array_copy = array_copy[:-1]

    return len(index_array), sorted(index_array)
```

В конце возвращаю длину получившегося списка индексов и сам список. Записываю ответ в файл.

```
if __name__ == '__main__':
    with open("input.txt", 'r') as f:
        datafile = f.readlines()

    result = half_subset(list(map(int, datafile[1].split())))
    with open("output.txt", 'w') as f:
        f.write(str(result))
```

Тесты:

Написал свои тесты для этой задачи. Первый тест из условия задачи, второй я сделал для 10000 рандомных массивов, с числами от 1 до размера и размерами от 10 до 500 элементов. Затем я сравнивал вывод моей функции с половиной суммы каждого рандомного массива.

```
class TaskTest(unittest.TestCase):

    def test_basic(self):
        self.assertEqual(half_subset([1, 2, 3]), (1, [3]))

    def test_random_numbers(self):
        for i in range(10000):

            size = random.randint(10, 500)
            initial = [random.randint(1, x + 1) for x in range(size)]
            output, index_array = half_subset(initial)

            half = sum(initial) // 2

            if output == -1:
                half = 0

            result_array = [initial[j - 1] for j in index_array]

            self.assertEqual(half, sum(result_array), \
            f"Test case {i} - {sum(result_array)} should be {half}")


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

Все тесты выполнены успешно.

```
Ran 2 tests in 3.391s

OK
```

Вывод:

Применив динамическое программирование удалось решить проблему подмножества элементов, используя время O(n), т.е. всего лишь за один проход массива.

Задача 17

Условие:

Шахматная ассоциация решила оснастить всех сво- их сотрудников такими телефонными номерами, которые бы набирались на кнопочном телефоне ходом коня. Например, ходом коня набирается телефон 340-49-27. При этом телефонный номер не может начинаться ни с цифры 0, ни с цифры 8.

Решение:

Начал решение с записи в словарь всех возможных ходов коня на телефоне:

```
possible_moves = {1: [6, 8],
                  2: [7, 9],
                  3: [4, 8],
                  4: [0, 3, 9],
                  5: [],
                  6: [0, 1, 7],
                  7: [2, 6],
                  8: [1, 3],
                  9: [2, 4],
                  0: [4, 6]}
```

Для решения решил использовать рекурсивную функцию, принимающую на вход одно число (цифра на телефоне), длину номера, а также кэш, который предотвращает повторное вычисление для одних и тех же входных данных.

```
def count_move(num: int, length: int, cache=None) -> int:
    if cache is None:
        cache = {}

    if cache.get((num, length)):
        return cache[(num, length)]

    if length == 1:
        return 1
```

Прописал условие, задающее кэш как словарь, условие, по которому проверяется кэш и так же базовый случай для рекурсивной функции.

Затем, если не базовый случай, то для каждого возможного хода из possible_moves вызывается эта же функция. Всё это записывается в результат, который записывается в кэш и наконец вовзращается.

```
    # else if not in cache and length > 1
    res = sum([count_move(n, length - 1, cache) for n in possible_moves[num]])
    cache[(num, length)] = res
    return res
```

В главной функции, считающей все возможные номера телефонов может быть три случая. Первый - это длина 1, тогда возвращается 8. Второй - если длина четная, тогда вызывается подсчет всех номеров начинающихся с единицы * 5 и с четверки * 2, так как при решении было замечено, что присутствуют определенные шаблоны, которые можно заменить этим умножением. В случае с нечетной длиной - все аналогично, только добавляется еще подсчет номеров, начинающихся с двойки. Вывод делается по модулю 10^9.

```
def knight_move(n: int) -> int:
    if n == 1:
        return 8

    elif n % 2 == 0:
        res = count_move(1, n) * 5 + count_move(4, n) * 2
    else:
        res = count_move(1, n) * 4 + count_move(4, n) * 2 + count_move(2, n)

    return res % 10**9
```

Тесты:

Для тестов использовал сайт https://acmp.ru, где решение прошло все тесты:

Задача	Язык	Результат	Тест	Время	Память
0471	Python	Accepted		0,046	550 Кб
1664	Python	Wrong answer	6	0,046	386 Кб

Вывод:

Используя рекурсию и некоторые оптимизации удалось составить быстрый и эффективный алгоритм, удовлетворяющий требованиям задачи.

Задача 20

Условие:

Слово называется палиндромом, если его первая буква совпадает с последней, вторая – с предпоследней и т.д. Например: «abba», «madam», «x».

Для заданного числа K слово называется почти палиндромом, если в нем можно изменить не более K любых букв так, чтобы получился палиндром. Например, при K = 2 слова «reactor», «kolobok», «madam» являются по- чти палиндромами (подчеркнуты буквы, заменой которых можно получить палиндром).

Подсловом данного слова являются все слова, получающиеся путем вычер- кивания из данного нескольких (возможно, одной или нуля) первых букв и

нескольких последних. Например, подсловами слова «cat» являются слова «c», «a», «t», «ca», «at» и само слово «cat» (а «ct» подсловом слова «cat» не является). Требуется для данного числа K определить, сколько подслов данного слова S являются почти палиндромами.

Решение:

Я разбил решение на три части. Первая - это функция-генератор для поиска всех подстрок строки.

```
def find_all_substrings(string: str):
    for i in range(len(string)):
        for j in range(i, len(string)):
            yield string[i:j + 1]
```

Следующая - определяет является ли строка почти палиндромом.

```
def is_almost_palindrome(k: int, word: str):
    if len(word) == 1:
        return True
    if k >= len(word)//2:
        return True

    counter = 0
    w_len = len(word)
    for i in range(w_len//2):
        if word[i] != word[w_len - i - 1]:
            counter += 1
            if counter > k:
                return False
    return True
```

Если строка состоит из одного символа или k >= половины длины строки, то она возвращает True. Если эти простые условия не были выполнены, то начинается перебор первой половины символов. Если символ, симметричный i-тому не равен ему, то к счетчику прибавляется единица, и если счетчик превысил k, то выводится False, не превысил - True.

Ну и финальная функция, считающая количество почти палиндромов во всех подстроках:

```python
def count_almost_palindromes(k: int, string: str) -> int:
    counter = 0
    for substring in find_all_substrings(string):
        if is_almost_palindrome(k, substring):
            counter += 1
    return counter
```

Тесты:

Для тестов использовал сайт https://acmp.ru, где решение не прошло тест по времени, застряв на 11 тесте, где вероятно крайне большой размер строки.

1664	Python	Wrong answer	6	0,031	386 Кб
0268	Python	Time limit exceeded	11	1,203	1826 Кб
0001	Python	Compilation error			

Вывод:

Так как поиск всех подстрок невозможен другим способом, кроме как за время O(n*n), то я не знаю как можно еще улучшить этот алгоритм, так как очевидно, что тормоза происходят именно на этом моменте. Вероятно, играет роль небольшая производительность языка Python.

Задача 11 (дополнительно)

Условие:

Вам дается набор золотых слитков, и ваша цель - набрать как можно больше золота в свою сумку. Существует только одна копия каждого слитка, и для каждого слитка вы можете либо взять его, либо нет (т.е. вы не можете взять часть слитка).

Решение:

Для решения использую рекурсивную функцию, принимающую на вход вместимость сумки, слитки и максимальный текущий вес.

```python
def find_max_weight(capacity: int, ingots: list[int], max_weight: int = 0) ->
                                                    int:
    # if all ingots processed return max_weight
    if not ingots:
        return max_weight

    # finding heaviest ingot and check if it fits
    heaviest = max(ingots)
    if max_weight + heaviest <= capacity:
        max_weight += heaviest

    # removing heaviest anyway and processing other ingots
    ingots.remove(heaviest)
    return find_max_weight(capacity, ingots, max_weight)
```

Базовым случаем для нее будет вариант, когда не осталось слитков. Если они еще есть - то выделяем самый тяжелый и смотрим подходит ли он по весу. Независимо от того, вмещается ли он в сумку - убираем его из слитков и продолжаем обрабатывать другие.

Тесты:

Для проверки решения написал свои тесты, где первый - это случай из условия. И придумал пять своих.

```
class TaskTest(unittest.TestCase):

    def test_basic(self):
        self.assertEqual(find_max_weight(10, [1, 4, 8]), 9, "OK")

    def test_custom(self):
        self.assertEqual(find_max_weight(20, [1, 4, 8]), 13, "OK")
        self.assertEqual(find_max_weight(50, [10, 40, 8]), 50, "OK")
        self.assertEqual(find_max_weight(45, [1, 40, 8]), 41, "OK")
        self.assertEqual(find_max_weight(200, [10, 40, 80]), 130, "OK")
        self.assertEqual(find_max_weight(2, [1, 41, 1]), 2, "OK")


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

Решение проходит все вышеуказанные тесты.

```
Ran 2 tests in 0.002s

OK
```

Вывод:

Используя рекурсивную функцию удалось написать решение, удовлетворяющее условию и решающее задачу за максимальное время O(n), где n - это количество слитков.

Задача 21 (дополнительно)

Условие:

Петя очень любит программировать. Недавно он решил реализовать популярную карточную игру «Дурак». Но у Пети пока маловато опыта, ему срочно нужна Ваша помощь. Как известно, в «Дурака» играют колодой из 36 карт. В Петиной программе каждая карта представляется в виде строки из двух символов, где первый символ означает ранг (‘6’, ‘7’, ‘8’, ‘9’, ‘T’, ‘J’, ‘Q’, ‘K’, ‘A’) карты, а второй символ означает масть (‘S’, ‘C’, ‘D’, ‘H’). Ранги перечислены в порядке возрастания старшинства. Пете необходимо решить следующую задачу: сможет ли игрок, обладая набором из N карт, отбить M карт, которыми под него сделан ход? Для того чтобы отбиться, игроку нужно покрыть каждую из карт, которыми под него сделан ход, картой из своей колоды. Карту можно покрыть либо старшей картой той же масти, либо картой козырной масти. Если кроющаяся карта сама является козырной, то её можно покрыть только старшим козырем. Одной картой можно покрыть только одну карту.

Решение:

Для решения я реализовал три вспомогательных функции. Первая - проверяет есть ли среди списка карт такая, которая сможет побить данную карту. Также я создал глобальную переменную RANKS, которую использую чтобы определять старшинство одной карты над другой.

```
RANKS = "6789TJQKA"
```

```
def check_same_colour(cards, card_to_beat):
    return [card for card in cards
            if card[1] == card_to_beat[1]
            and RANKS.index(card[0]) > RANKS.index(card_to_beat[0])]
```

Вторая - возвращает все козырные карты из списка карт.

```
def get_trump_cards(cards, trump_colour):
    return [card for card in cards
            if card[1] == trump_colour]
```

Третья - возвращает множество карт, которые могут побить карты противника.

```
def get_beating_cards(my_cards, opponent_cards):
    beating_cards = []
    for op_card in opponent_cards:
        beating_cards += check_same_colour(my_cards, op_card)
    return list(set(beating_cards))
```

Теперь, основная функция. Для начала я распределяю все карты по четырем категориям - мои козыри и остальные карты, козыри противника и остальные карты противника.

```
def can_beat_cards(trump_colour, my_cards, op_cards):
    my_trumps = get_trump_cards(my_cards, trump_colour)
    op_trumps = get_trump_cards(op_cards, trump_colour)

    my_cards = [card for card in my_cards if card not in my_trumps]
    op_cards = [card for card in op_cards if card not in op_trumps]
```

Затем, я получаю список карт, которые могут побить карты противника отдельно для козырей и отдельно для остальных карт.

```
    beating_trumps = get_beating_cards(my_trumps, op_trumps)
    beating_cards = get_beating_cards(my_cards, op_cards)
```

Потом, я смотрю прежде всего на то, смогут ли мои козыри побить козыри противника. Если проверка прошла, то я записываю возможные лишние козыри в список, чтобы не потерять их при дальнейших сравнениях (но записываю только в том случае, если их оказалось больше чем козырей противника).
После, я проверяю могут ли мои обычные карты побить обычные карты противника, если проверка не проходит - то я уже смотрю могут ли мои обычные карты + все мои оставшиеся козыри побить карты противника.

```
    if len(beating_trumps) >= len(op_trumps):

        extra_trumps = beating_trumps[len(beating_trumps)-1:] \
            if len(beating_trumps) > len(op_trumps) \
            else []

        if len(beating_cards) >= len(op_cards):
            return "YES"
        else:
            my_trumps = [trump for trump in my_trumps if trump not in
                                                    beating_trumps] +
                                                    extra_trumps
            if len(beating_cards) + len(my_trumps) >= len(op_cards):
                return "YES"

    return "NO"
```

Ну и заворачиваю это все через обработку ввода/вывода через файлы.

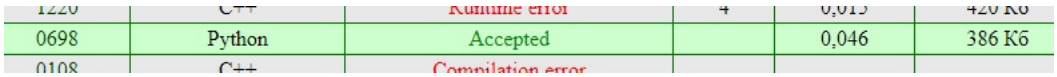
```
with open("input.txt", 'r') as f:
    datafile = f.readlines()

result = can_beat_cards(datafile[0].split()[2],
                        datafile[1].split(),
                        datafile[2].split())

with open("output.txt", 'w') as f:
    f.write(str(result))
```

Тесты:

Для тестов использовал сайт https://acmp.ru, где решение успешно прошло все тесты.



Вывод:

С помощью жадных алгоритмов и разбиения задачи на мелкие подзадачи удалось эффективно решить поставленную в условии проблему битья карт.

Задача 13 (дополнительно)

Условие:

Вы и двое ваших друзей только что вернулись домой после посещения разных стран. Теперь вы хотели бы поровну разделить все сувениры, которые все трое накупили.

Решение:

Для решения написал рекурсивную функцию, которая перебором убавляет от каждой подсуммы и чтобы не повторять действия - сохраняет уже вычисленные значения в кэш.

```
def can_divide(coins, n, first, second, third, cache):
    if first == 0 and second == 0 and third == 0:
        if n < 0:
            return True
        return False

    if n < 0:
        return False
```

Здесь first, second, third - это три подсуммы, coins - монеты, n - количество оставшихся сувениров. Также прописал базовый случай для выхода из рекурсии - когда все три подсуммы достигают нуля или когда количество сувениров уходит в отрицательное значение.
Затем, я проверяю высчитывал ли я уже значения подсуммы и если нет - то рекурсивно нахожу куда лучше всего подходит значение сувенира, и в конце записываю в кэш.

```
    way = (first, second, third)
    if way not in cache:
        to_first = False
        to_second = False
        to_third = False

        if first - coins[n] >= 0:
```

```
            to_first = can_divide(coins, n - 1, first - coins[n], second, third
                                                        , cache)
        if not to_first and second - coins[n] >= 0:
            to_second = can_divide(coins, n - 1, first, second - coins[n],
                                                       third, cache)
        if not to_first and not to_second and third - coins[n] >= 0:
            to_third = can_divide(coins, n - 1, first, second, third - coins[n]
                                                        , cache)

        cache[way] = to_first or to_second or to_third

    return cache[way]
```

Обработка ввода/вывода выглядит следующим образом:

```
if __name__ == '__main__':
    with open("input.txt", 'r') as f:
        datafile = f.readlines()

    n = int(datafile[0]) - 1
    coins = list(map(int, datafile[1].split()))

    summ = sum(coins)
    if summ % 3 != 0:
        result = False
    else:
        third = summ // 3
        result = can_divide(coins, n, third, third, third, {})

    with open("output.txt", 'w') as f:
        f.write("1") if result else f.write("0")
```

Проверяю делится ли сумма значений монет на 3 или нет, и в зависимости от этого либо сразу вывожу результат либо запускаю функцию.

Тесты:

Написал свои тесты для этой задачи. Первые тест - из условия. А второй - основан на рандомных массивах одинаковых монет размером от 5 до 50. И все это сравнивается с количеством сувениров (т.к. они все одинаковые), и если количество делится на три - то значит поделить можно и следовательно будет точно True.

```
class TaskTest(unittest.TestCase):

    def test_basic(self):
        self.assertEqual(can_divide([3, 3, 3, 3], 3, 12 // 3, 12 // 3, 12 // 3,
                                                    {}), False, "OK")
        self.assertEqual(can_divide([40], 0, 40 // 3, 40 // 3, 40 // 3, {}),
                                                   False, "OK")
        self.assertEqual(can_divide([17, 59, 34, 57, 17, 23, 67, 1, 18, 2, 59],
                                                    10, 118, 118, 118, {}), True,
                                                   "OK")
        self.assertEqual(can_divide([1, 2, 3, 4, 5, 5, 7, 7, 8, 10, 12, 19, 25]
                                                   , 12, 36, 36, 36, {}), True, "
                                                   OK")

    def test_custom(self):
        for i in range(10000):
            num = random.randint(1, i+2)
            size = random.randint(5, 50)
            array = [num for _ in range(size)]
            one_third = sum(array) // 3
```

```
            result = can_divide(array, len(array) - 1, one_third, one_third,
                                                    one_third, {})

            self.assertEqual(result, len(array) % 3 == 0, "OK")


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

Все тесты пройдены.

```
Ran 2 tests in 11.468s

OK
```

Вывод:

Учитывая небольшие поставленные ограничения на ввод (до 30 штук) считаю что программа эффективно справляется с поставленной задачей разделения сувениров поровну между друзьями, используя рекурсивный алгоритм.

Выводы:

Для разных задач подходят разные способы решения. Если ограничения на входные данные - небольшие, то можно написать рекурсию, которую необходимо максимально оптимизировать, чтобы не переполнить стек. Если задачу можно упростить, разбив на мелкие подзадачи - то это может значительно ускорить и упростить для понимания решение. Если можно проработать каждый путь развития задачи, то это тоже будет довольно эффективно по времени и памяти. Но многие из этих решений могут значительно затруднить скорость написание кода и значительно увеличить его размер. В то время как жадные алгоритмы или динамическое программирование гораздо более понятны со стороны и занимают меньше времени на реализацию.