МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В. ЛОМОНОСОВА

Экономический Факультет
Кафедра Экономической Информатики

Научно-исследовательская работа по теме:

“Применение методов динамического программирования в задаче о кэшбеке”

Выполнил:
Студент 3 курса э308 группы
Гришин А. Ю.

Преподаватель:
Кандидат экономических наук,
Кандидат физико-математических наук,
Кандидат юридических наук,
Доцент

Сидоренко В.Н.

Москва, 2022

Оглавление

Введение и определение проблемы

Для наиболее полного и точного описания моделей, представленных ниже, необходимо обозначить основные термины и, что более важно, максимально полно изложить идею, лежащую в основе настоящей работы.

Банк - кредитная организация, которая имеет исключительное право осуществлять в совокупности следующие банковские операции: привлечение во вклады денежных средств физических и юридических лиц, размещение указанных средств от своего имени и за свой счет на условиях возвратности, платности, срочности, открытие и ведение банковских счетов физических и юридических лиц.[1]

Кэшбек – cashback – от латинского “деньги назад” = возврат денег. Получение денежных средств от банка на основании договора между клиентом и банком соответственно. Основные термины введены, переходим к их анализу:

С одной стороны, банк имеет дело с деньгами своих клиентов, что делает его зависимым от имеющихся у них средств. Отсюда **вывод**: одна из задач успешного банка – привлечение клиентов. Это возможно, например, посредством предложения клиенту более выгодных условий по вкладам и кредитам.

С другой стороны, банку, как и любому другому экономическому агенту, не выгодно терпеть убытку, излишне тратя средства на привлечение аудитории. То есть нужно “казаться” клиенту честными и делающим все для его выгоды, однако производить такие манипуляции со вкладами, чтобы клиент – будучи Homo Economicus – не смог выигрывать от операций, совершенных в банке/ах. Тут подходит слово “арбитраж”, однако получение дополнительных средств [под “средствами” понимается денежный эквивалент] рассматривается с точки зрения одного человека, следовательно, для наиболее четкого представления в данной работе исследуется связь между банком и его клиентом.

NB! **Промежуточный вывод**: банку нужно быть “честным”, но осмотрительным и расчётливым и, привлекая дополнительного клиента, “извлекать из него” все возможные средства для собственных манипуляций. Одним из способов преумножения числа клиентов многие банки считают проведение акций или учреждение политики возврата денежных средств при совершении покупок от некоторой заданной суммы: кэшбек[2].

Далее для большей конкретики приводим размышления, рассуждения и исследования на тему решения проблемы получения в виде кэшбека наиболее близкой или равной заданной банком сумме величины.

Актуальность

Данная работа актуальна, так как на сегодняшний день в среднем каждый человек является клиентом конкретного/ых банка/ов. Необходимость проведения денежных операций в цифровой среде уже является ежедневной необходимостью, что заметно затрудняет процесс экономии денежных средств. Так как, например, взимание комиссии за перевод суммы выше заданной или изменяющаяся ставка процента, рассчитываемая на дневной основе, понемногу отнимают все большую и большую часть от используемых человеком средств, которые тот мог бы потратить более рационально. Для устранения данной трудности необходимы время, силы и желание разбираться и учиться “обходить” систему. Однако индивид, вращающийся в оживленной бизнес-сфере, несущий большую ответственность за свои действия на работе, не имеет всего вышеперечисленного. Значит, наиболее простое в использовании решение данной проблемы позволит каждому человеку в отдельности сберегать больше, следовательно потребление и/или вклады его тоже увеличатся, так как при прочих равных условиях наиболее рационально для любого Homo Economicus – занятие хрематистикой и удовлетворение собственных потребностей, а не исключительное увеличение сбережений. В результате, удовольствие, получаемое при потреблении некоторого предмета, по мнению Ксенофонта[3], Платона[4] и Аристотеля, приводит к большему довольству условиями собственной жизни, то есть к более производительному труду. А если это верно для конкретного человека, то верно и для любого, так как экономические агенты схожи по тому, что каждый из них приносит вклад в экономическое развитие, что, в свою очередь, увеличивает совокупный ВВП страны, что означает, что конкретная страна более активно развивается как в области экономики, так и в областях науки, техники, промышленности, медицины и так далее.

Цели

Целю текущей работы является попытка дать каждому человеку, связанному с финансовой сферой, возможность сберегать наибольшую часть собственных средств, обращающихся и используемых в цифровом виде. Это позволит существенно увеличить спрос в других секторах экономики, отвечающих за развлечения, плановые покупки, образование, автомобильные сервисы, медицину, банковскую сферу и так далее. Что впоследствии приведет к большей удовлетворенности собственной жизнью и далее к более производительному труду.

NB! **Вывод**: ВВП страны будет увеличиваться.

Происходить данное увеличение будет посредством предложения и использования алгоритма решения задачи, являющейся частью класса задач оптимизации: задачи о кэшбеке.

Задачи

Если: целью данной работы является предоставление возможности решения банковской задачи о кэшбеке каждому человеку, связанному с денежными потоками,

То: необходимо максимально упростить процесс решения данной задачи для пользователя, чтобы тот, не прилагая каких-либо усилий, не позволял банку/ам получать больше предоставляемых с обеих сторон – клиент банку и банк клиенту – средств.

Вывод: написание алгоритма – наиболее точная и удовлетворяющая критериям простоты и удобства использования задача настоящей работы.

NB! Спецификой является применение методов Динамического Программирования, что подразумевает поиск и индуктивное использование инварианта. Следовательно самое главное во всем процессе разработки – достижение наиболее полного понимания поставленной проблемы. Для это необходимо через формальную постановк разложить ее на бинарные составляющие и, используя метод синтеза, сформировать из имеющегося набора данных алгоритм.

Действия: Процесс работы задуманной программы производится в несколько этапов:

1. Организация ввода с клавиатуры последовательности из N чисел, символизирующих набор затрат за период;
2. Организация ввода с клавиатуры числа M, символизирующего максимальную величину кэшбека, который можно получить, имея текущий набор затрат;
3. Организация процесса выбора оптимальных элементов последовательности, позволяющих максимально приблизиться к заданной ранее величине M;
4. Организация процесса вывода оптимальных элементов последовательности и подсчет разницы между фактической и ожидаемой суммами.

Формализация проблемы и построение решения

Для лучшего понимания происходящего формализуем модель в словесной форме, далее в математической, далее в алгоритмической.

Словесная формализация

Представим ситуацию: некоторый банк вводит политику возврата денежных средств. Основные положения политики следующие:

1. В течение периода клиент совершает покупки на определенные суммы, расходуя средства со своего счета;
2. В конце периода клиенту предоставляется возможность вернуть часть затраченных средств. Для этого он может выбрать такие из своих затрат, что они удовлетворяют требованиям, установленным банком. Например: использовать покупки только от 1’500 рублей;
3. ***NB!*** Возможен возврат суммы НЕ БОЛЕЕ, чем *r*-ая часть от общих затрат за период.

Промежуточный вывод: клиент, исходя из своих затрат, может набрать любую сумму не превышающую указанную банком. Это основная трудность.

NB! Далее вместо слов “наиболее близкая сумма” используется сочетание слов “оптимальная сумма/величина”. Иными словами, подобрав неоптимальную сумму, клиент отдает банку больше, чем должен, однако претензий к банку нет, так как клиент сам вычислил сумму возврата.

Общий вывод: необходимо реализовать способ максимальной минимизации разницы между набранной величиной и максимально возможным кэшбеком.

Математическая формализация

Нас интересует решение следующей системы:

$$\begin{cases} x^* = \arg\left(\min\left(M - \sum_{j=1}^{|M_l|} M_{l,j}\right)\right) \\ \sum_{j=1}^{|M_l|} M_{l,j} \le M, l \in \sum_{k=0}^{|G|=N} \binom{|G|}{k} \\ \forall j \in \{1, \dots, N\}, \forall l \in \sum_{k=0}^{|G|=N} \binom{|G|}{k} => M_{l,j} \in Q^+ \end{cases}$$

Откуда она взялась? Что означает каждый ее элемент и каждая из строчек? Рассмотрим детальнее и убедимся в логичности и необходимости каждого выражения:

1. Вводим необходимые пояснения и трактовки:
    a) $\{1, \dots, N\}$ – множество индексов введенных величин, символизирующих затраты за период;
    b) G – множество введенных чисел, затраты за период. $|G| = N$;
    c) $M_l \subset G$ – подмножество элементов исходного множества введенных чисел;
    d) $l \in \sum_{k=0}^{|G|} \binom{|G|}{k} = \sum_{k=0}^{|G|} \frac{|G|!}{(k)!(|G|-k)!}$ – количество сумм, которые можно сформировать, имея N элементов исходного множества. Если представить облако из счетного числа точек, положим *N*, то количество подмножеств, которые можно сформировать из него вычисляется по формуле: $\sum_{k=0}^{N} \frac{N!}{(k)!(N-k)!}$
    e) $M_{l,j}$ – *j*-ый элемент подмножества M_l множества исходных чисел G.
2. $\forall j \in \{1, \dots, N\}, \forall l \in \sum_{k=0}^{|G|=N} \binom{|G|}{k} => M_{l,j} \in Q^+$- каждая сумма затрат неотрицательна и принадлежит множеству рациональных числе. Логично, что для покупки некоторого товара человек тратит фиксированную сумму денежных средств, представленную один из элементов множества рациональных чисел. Иными словами, нельзя при покупке выплатить $\sqrt{2}$ или $\frac{1}{3}$ денежных единиц. Для большей реалистичности следует уменьшить мощность множества рациональных чисел до:

$$Q_*^+ = \{x \mid x \cdot 100 \in N \cup \{0\} \wedge x \in Q^+\}$$

Если: данное условие не выполняется,

То: решение задачи перестает соответствовать реальности

Следствие: противоречие цели наибольшей эффективности применения человеком в повседневной жизни.

Вывод: решение задачи не отвечает заданным условиям.

3. $\sum_{j=1}^{|M_l|} M_{l,j} \leq M$ – сумма всех элементов множества $M_l \subset G: l \in \sum_{k=0}^{|G|}\binom{|G|}{k}$ не превосходит M – максимально возможной величины получаемого кэшбека. Иными словами, нельзя получить больше, чем разрешено банком. Вдобавок, данное ограничение не обязательно является единственным, здесь – в настоящей работе – рассматривается его классический вариант – с единственным ограничением, однако возможны постановки, когда, необходимо, например, обязательно включить конкретную величину в сумму. Тогда еще можно добавить h ограничений, причем важно понимать, что

$$h \in \left[0, \sum_{k=0}^{|G|}\binom{|G|}{k}\right]$$

так как количество дополнительных ограничений со знаком $\leq$, наложенных на любую комбинацию переменных, не может превышать $\sum_{k=0}^{|G|}\binom{|G|}{k}$. Однако в данный момент сконцентрируемся на решении классического варианта задачи, чтобы потом иметь возможность при необходимости ее усложнить.

Если: данное условие не выполнено,

То: решение перестает иметь какой-либо смысл, так как наша задача снизу максимально приблизиться к M.

Следствие: человек не получает пользы от решения задачи в повседневной жизни.

Вывод: решение бесполезно

4. $x^* = \arg\left(\min\left(M - \sum_{j=1}^{|M_l|} M_{l,j}\right)\right)$ – “сердце” задачи – множество таких

$$x_j^* \in M_l^* \; \forall j \in \{1 \dots |M_l^*|\}, l \in \sum_{k=0}^{|G|}\binom{|G|}{k},$$

что сумма элементов данного множества M_l^* соответствует супремуму подмножества всевозможных сумм, не превосходящих M, элементов исходного множества заданных чисел, где M – величина кэшбека.

$$x^* = M_l^* = \sup\left(\left\{\sum_{j=1}^{|M_l|} M_{l,j} : l \in \sum_{k=0}^{|G|}\binom{|G|}{k} = \sum_{k=0}^{|G|}\frac{|G|!}{(k)!\,(|G|-k)!}, \sum_{j=1}^{|M_l|} M_{l,j} < M\right\}\right)$$

Если: данное условие не выполняется

То: поставленная задача не решена

Следствие: пользы клиенту банка от решения нет

Вывод: решение бесполезно

Таким образом, получаем необходимость каждого из вышеперечисленных условий для достижения поставленной цели.

Алгоритмическая формализация

Рассмотрим задачу с алгоритмической точки зрения. То есть обозначим последовательные шаги, выполнение которых приводит к желаемому результату. Для этого разделим работу алгоритма на 3 части. Вход – ввод данных пользователем, вычисления – поиск оптимального набора, выход – вывод полученного результата.

NB! Алгоритм еще не объяснен, поэтому обозначим его как функцию от 2-х переменных: последовательности чисел и максимального кэшбека.

$$f(s,c): Q^{+,N} \times Q^{+,1} \to Q^{+,|M_l^*|}, \qquad M_l^* \subset G, l \in \sum_{k=0}^{|G|} \binom{|G|}{k}$$

$$s - sequence - \text{последовательность}$$

$$\text{c} - cashback - \text{кэшбек}$$

Подробнее:

1. $X = Q^{+,N} \times Q^{+,1}$ – первое множество – последовательность из N элементов, символизирующая затраты за период, второе множество – максимальная величина кэшбека – тоже рациональная, так как величина получаемых денежных средств не может быть представлена в виде иррационального числа. Это противоречит логике.
2. $Y = Q^{+,|M_l^*|}$ – оптимальное множество элементов, в сумме дающих наиболее близкое к заданному кэшбеку M число. $M_l^* \subset G, l \in \sum_{k=0}^{|G|}\binom{|G|}{k} : |M_l^*| \leq |G| = N$

Важно понимать, что речь идет не совсем о функции, но о сюръекции, так как одному и тому же набору выходных данных может ставиться в соответствие разный набор входных данных. Для наглядности рассмотрим пример:

$$f([1,2,4], 10) = [1,2,4]$$
$$f([1,2,4], 100) = [1,2,4]$$

Теперь рассмотрим подробнее каждый из этапов работы алгоритма:

1. Ввод:
    a) Первая строка: последовательность неотрицательных чисел с плавающей точкой – суммы затрат за период: $s_0 \in Q^{+,N}$
    b) Вторая строка: неотрицательное число с плавающей точкой – максимальная величина кэшбека: $c_0 \in Q^{+,1}$
2. Вычисления: $f(s_0, c_0) \to ans = answer = \text{ответ}$
3. Вывод данных:
    a) Первая строка: вывод последовательности выбранных сумм: ans

b) Вторая строка: вывод разницы между ожидаемым кэшбеком M и фактической величиной $\sum_{j=1}^{|ans|} ans_j$: $rem = remainder = M - \sum_{j=1}^{|ans|} ans_j$

Основной интерес представляет из себя рассмотрение и анализ функции – она же сюръекция или алгоритм. Однако прежде, чем что-то изучать это нужно придумать.

Если: говорить неформально,

То: все условие можно свести в фразе: “Как из набора чисел выбрать такие, чтобы их сумма была максимально близкой снизу к заданному числу?”

Но остается вопрос: как сделать так, чтобы для любого набора чисел, удовлетворяющих исходным условиям, алгоритм работал верно?

Вывод:

a) Нужно придумать алгоритм

b) Нужно обосновать его справедливость в общем виде

c) Нужно подсчитать вычислительную сложность алгоритма

Практическая реализация алгоритма

Для примера рассмотрим задачу, на вход которой подается набор чисел:

1. 0.04,0.03,0.06,0.09 – затраты за период – из математической формализации сразу умножим на 100;
2. 0.08 – максимальный кэшбек – аналогично умножим на 100;

Очевидно, что $f([4,3,6,9], 8) = [4,3]$, а остаток 1. Но как сказать компьютеру, какие именно шаги необходимо выполнить, чтобы получить упомянутый результат?

J [горизонталь]	0	1	2	3	4
W [вертикаль]	4	3	6	9	X
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	3	3	3
4	0	4	4	4	4
5	0	4	4	4	4
6	0	4	4	4	4
7	0	4	7	7	7
8	0	4	7	7	7

Рисунок 1 – решение вышеупомянутой задачи методом динамического программирования

Можно и так, как представлено в таблице слева, однако не совсем ясно, что откуда появилось. Поэтому разберемся со всем постепенно: для начала - переформулируем поставленную задачу. Пусть у нас есть емкость размером $M \cdot 100$ то есть максимальную величину кэшбека переводим в множество $\mathbb{N} \cup \{0\}$. В нее нам необходимо уместить максимальное количество чисел из заданного набора, так, чтобы они все поместились в емкость. Однако M была умножена на 100, значит, все остальные величины тоже необходимо умножить на 100, чтобы сохранить масштаб. Умножение производится исключительно на 100, так как ранее, в блоке Математическая формализация, фигурировало требование о вводе чисел не более, чем с 2-я знаками в мантиссе. Подобным

умножение производится технический перевод элементов множества рациональных чисел в множество натуральных чисел. Теперь наше условие выглядит следующим образом:

Вход:

1. $x_1, \dots, x_n : \forall j \in \{1 \dots n\} => x_j \in \mathbb{N} \cup \{0\}$
2. $M \in \mathbb{N}$

Вывод:

1. $x_1^*, \dots, x_m^* : \{x_j^*\}_{j=1}^{m} \subset \{x_j\}_{j=1}^{N}$
2. $M - \|x^*\|_1$

Алгоритмическая формализация сделала задачу похожей на задачу о рюкзаке. То есть можно свести задачу о кэшбеке к уже решеной, значит, не придется придумывать алгоритм заново. Действительно, рассмотрим условия задачи о рюкзаке:

Вход:

1. $\forall j \in \{1 \dots N\} => w_j \in \mathbb{N} - \text{веса предметов}$
2. $\forall j \in \{1 \dots N\} => c_j \in \mathbb{N} - \text{стоимость предметов}$
3. $W \in \mathbb{N}$ - вместимость рюкзака

Выход:

1. Максимальная стоимость предметов максимального суммарного веса не более W

NB! Существует как минимум 2 разновидности задачи о рюкзаке. В настоящей работе рассматривается базовый вариант ее постановки: объекты неделимы, каждый из предметов в последовательности может встречаться 1-н раз. Действительно: сводим задачу к уже решеной, анализируем алгоритм, определяем вычислительную сложность и понимаем, можно ли было получить решение быстрее.

Сведéние: задача о кэшбеке к задаче о рюкзаке

Трудность в том, что в условии задачи о рюкзаке фигурируют веса и стоимости, а в нашей задаче присутствуют только стоимости. Для решения данной проблемы пусть веса и стоимости тождественно равными друг другу. Так как нет необходимости заботиться о том, сколько вести каждый их элементов последовательности. Выписываем обновленную формализацию задачи.

Вход:

1. $w_j = c_j = x_j : j \in \{1 \dots N\}, w_j, c_j, x_j \in \mathbb{N} \cup \{0\}, w_j - \text{веса}, c_j - \text{стоимость}, x_j - \text{затраты}$
2. $M = W : M, W \in \mathbb{N} \cup \{0\}, W - \text{максимальная вместимость емкости}, M - \text{кэшбек}$

Вывод:

1. $x^* : M - \|x^*\|_1 \to min, \text{разница между теоретическим и фактическим кэшбеком}$

2. Если представить x^* не как множество, а как вектор, то $\|x^*\|_1 = \sum_{j=1}^{|M_l^*|} x_j^*$ по определению векторной нормы.

Теперь у нас есть оформленная задача, которую необходимо решить. Как это сделать? Сразу пробуем методы Динамического Программирования.

Анализ алгоритма: метод Динамического Программирования

Если применяется Д.П., то самое главное - определить подзадачу, она же – инвариант. Для этого используем рассуждения: пусть рюкзак уже как-то заполнен и в нем лежит *j*-ый предмет. Значит, у данного предмета уже указаны и вес w_j и его стоимость c_j. В нашем случае, и вес, и стоимость эквиваленты, однако для наиболее полной картины представления решения рассмотрим случай, якобы у нас есть оба вектора данных.

Вопрос: что будет, если данный предмет вытащить из рюкзака?

Ответ: не всегда оптимальное решение.

Утверждение: возможно неоптимальное решение заполнения рюкзака вместимости $W - w_j$.

Доказательство:

a) **Если**: заполнение не оптимально,
   То: можно взять более ценный предмет и поместить его в емкость.
   Следствие: заполнение стало оптимальным.

b) Но изначальное заполнение – до извлечения w_j – считалось оптимальным, значит, заполнение рюкзака без данного предмета тоже является оптимальным.

c) **Промежуточный вывод**: получается, что заполнение должно быть оптимальным.

d) Однако в нашем случае рассматривается задача о рюкзаке без повторений. Каждый элемент последовательности можно использовать лишь единожды. И при этом мы не знаем, содержит ли оптимальное заполнение для емкости вместимостью $W - w_j$ предмет w_j.

e) **Промежуточный вывод**: для получения оптимального решения недостаточно просто убрать w_j.

f) **Вывод**: нужно следить, какие предметы уже учтены в емкости, а какие нет.

Пусть данная матрица $D \in \mathbb{N}^{(M+1)\times(N+1)}$: дополнительная строка и столбец соответственно введены для учета емкости 0. Столбцы – это предметы доступные для заполнения емкости, а строки – это вместимость емкости. В нашем случае: вертикаль - $x_1, \dots, x_n$, а горизонталь – последовательность чисел от 0 до M. Координаты матрица интерпретируются как: $D[w, j]$ – максимальная стоимость рюкзака, вмещающего в себя предметы с номерами от 0 до j и размером w.

Словесная формализация подзадачи: у емкости есть 2 параметра: вместимости и какие предметы в нее можно упаковывать. Значит, $\forall w \in [0, W], \forall j \in [0, n]$ $D[w, j]$ максимальная - оптимальная – стоимость рюкзака вместимостью w, заполненного только предметами с номерами $[0, j]$.

Формульная формализация подзадачи:

$$D[w, j] = \max\left(D\left[w - w_j, j - 1\right] + c_j, D[w, j - 1]\right) \qquad (*)$$

Рассмотрим выражение подробнее:

1. $D[w, j]$ – текущее рассматриваемое оптимальное заполнение – максимальная стоимость - емкость вместимостью w с помощью предметов от 0 до j.
2. $D\left[w - w_j, j - 1\right]$ – оптимальное заполнение емкости “предыдущей вместимости”, то есть емкость вместимостью $w - w_j$ – без текущего предмета с использование предметов с номерами от 0 до $j - 1$.
3. $D[w, j - 1]$ – текущее заполнение емкости вместимостью w с помощью предметов с номерами только от 0 до $j - 1$.

Идея, лежащая в основе формулы такова: “Что стоит дороже: емкость с текущим предметом или без него?” Действительно, $D\left[w - w_j, j - 1\right] + c_j$ – стоимость емкости “предыдущей” вместимости, в которую упаковали j-ый предмет, а $D[w, j - 1]$ – стоимость емкости текущей вместимости без текущего предмета. Оптимальным выбором для решения является наибольшая из вышеупомянутых величин. Подзадача сформулирована. Рассматриваем псевдокод:

```
int cashPack(weight = [w(1),...,w(n)], cost = [c(1),...,c(n)],W) {
    def D = array int [W + 1,n + 1];
    for (int j = 0, j <= N, j++){
        D[0,j] = 0;
    }
    for (int w = 0, w <= W, w++){
        D[w,0] = 0;
    }
    for (int j = 1, j <= N, j++){
        for (int w = 1, w <= W, w++){
            D[w,j] = D[w,j - 1];
            if (D[w,j - 1] <= w) {
                D[w,j] = max(D[w - weight[j - 1]][j - 1] + cost[j - 1], D[w,j]);
            }
        }
    }
    return D[W,n];
}
```

Рисунок 2 – псевдокод получения оптимального – максимального по стоимости – заполнения емкости вместимостью W = M с использование предметов с номерами от 0 до N

NB! Псевдокод очень похож на полноценный код программы на C++, однако для большего удобства интерпретации принимаемые функцией аргументы и двумерный массив, инициализируемый в процессе решения, заданы в явном виде.

Рассмотрим каждый строку отдельно [номера строк на рисунке и пунктов ниже равны]:

1. Объявление и инициализация целочисленной функции cashPack, принимающей на вход 3 аргумента:
    a) Weight – целочисленный массив весов объектов размера N.
    b) Cost – целочисленный массив стоимостей объектов размера N.
    c) W – максимальная вместимость рюкзака.
2. Объявление целочисленного двумерного массива D размера $(W + 1) \times (N + 1)$ элементов.
3. Начало цикла, проходящего по каждому столбцу первой строки двумерного массива D.
4. Инициализация *j*-ого элемента 0-ой строки значением 0.
5. Конец цикла.
6. Начало цикла, проходящего первый элемент каждой строки двумерного массива D.
7. Инициализация 0-ого элемента *w*-ой строки значением 0.
8. Конец цикла.
9. Начало цикла по всем столбца.
10. Начало цикла по всем строкам.
11. Присвоение текущему способу заполнения максимальной стоимости заполнения без использования последнего доступного предмета.
12. Начало условия возможности емкости вместить текущий предмет.
13. Присвоение текущей ячейке двумерного массива - текущему способу заполнения емкости соответствующей размерности с использованием только элементов с номером не выше соответствующего - значения, полученного по формуле (*).
14. Конце условия.
15. Конец цикла по всем строкам.
16. Конце цикла по всем столбцам.
17. Возврат последнего элемента последней строки двумерного массива D.

Промежуточный вывод: приведенный алгоритм позволяет получить наиболее близкое снизу число из заданного набора не выше *M*.

Вывод: задача не является решенной, так как в ответ сюръекция должны возвращать вектор использованных элементов изначального ввода. Однако алгоритм, основанный на методе Динамического Программирования “Bottom-up” = подъем “снизу-вверх” = последовательное

усложнение простейшего варианта задачи, позволяет подсчитать очень важный показатель. Числовое решение задачи. Но теперь стоит вопрос:

Вопрос: как выбрать нужные элементы так, чтобы в сумме они давали полученное число?

Ответ: используем еще один метод Динамического Программирования под название “Top-down” = спуск “верху-вниз” = разложение полученного решения на составляющие элементы. Для понимания того, как “возвращаться назад” рассмотрим в качества примера рисунок 1. Рисунок иллюстрирует таблицу, полученную в ходе применения ранее описанного псевдокода для решения задачи о кэшбеке на примере входных данных [0.04,0.03,0.06,0.09], 0.08. Как понять, зная, что ответ к задаче 0.07, что именно 0.04 и 0.03 образуют данное число? Рассмотрим внимательнее последнюю строку. Очевидно, что как только новый предмет добавляется в емкость, ее стоимость меняется. Однако ответом к задаче является последний элемент последней строки двумерного массива D. Значит, начинаем раскручивать решения в обратную сторону именно с позиции [W, n] в D. Формализуем данную идею.

Словесная формализация: до тех пор, пока не упремся в первый элемент строки, ищем изменения оптимального заполнения емкости. А добавлением очередного предмета служит изменение стоимости емкости, то есть суммы, получаемой при оптимальном заполнении рюкзака соответствующей вместимости.

Алгоритмическая формализация: рассмотрим новый псевдокод ранее инициализированной и объявленной функции.

```
([int]) cashPack(weight = [w(1),...,w(n)], cost = [c(1),...,c(n)],W) {
    def D = array int [W + 1,n + 1];
    for (int j = 0, j <= N, j++){
        D[0,j] = 0;
    }
    for (int w = 0, w <= W, w++){
        D[w,0] = 0;
    }
    for (int j = 1, j <= N, j++){
        for (int w = 1, w <= W, w++){
            D[w,j] = D[w,j - 1];
            if (D[w,j - 1] <= w) {
                D[w,j] = max(D[w - weight[j - 1]][j - 1] + cost[j - 1], D[w,j]);
            }
        }
    }
    // return D[W,n];
    def ans = array double [?];
    def i = int W;
    def j = int n;
    while (j - 1 >= 0) {
        if (D[i,j - 1] != D[i,j]){
            ans.append(weight[j - 1] / 100);
            i -= weight[j - 1];
        }
        j--;
    }
    return ans;
}
```

Рисунок 3 – псевдокод получения оптимального набора для получения оптимального – максимального по стоимости – заполнения

Корректность строк 1-17 была ранее объяснена и показана. Теперь рассмотрим строки 18-29.

18. Объявление нецелочисленного векторного одномерного массива ans. Мы не знаем, сколько элементов было использовано в ходе формирования оптимальной суммы, значит, необходимо использовать динамическое автоматическое выделение памяти.
19. Объявление и инициализация индекса *i* рассматриваемой строки: начало с последней строки.
20. Объявление и инициализация индекса *j* рассматриваемого столбца: начало с последнего столбца.
21. Начало цикла: до тех пор, пока не будет достигнут первый элемент строки.
22. Начало условия: если текущая оптимальная стоимость емкости НЕ совпадает с оптимальной стоимостью емкости без текущего предмета
23. То добавляем вес данного предмета, деленый на 100, в нецелочисленный векторный одномерный массив ans.
24. То уменьшаем индекс вместимости *i* на вес добавленного в ans предмета.
25. Конце условия.
26. Уменьшаем индекс столбца *j* целочисленного двумерного массива D на 1.
27. Конце тела цикла.
28. Возвращаем нецелочисленный векторный одномерный массив ans с оптимальным набором предметов.

Полный псевдокод программы обоснован. Значит, поставленная задача решена. Однако остается важный момент: оценка вычислительной сложности.

Оценка вычислительной сложности

Данный алгоритм формирует матрицу размером $(W+1) \times (N+1)$ и далее производит с ней последовательные действия. Рассмотрим, сколько “стоит” каждое из действий:

1. Строки псевдокода: 3-5 – проходят по всем элементам первой строки и присваивают каждому из них значение 0. Следовательно 1-а операция на 1-у итерацию, а всего итераций $(N+1)$. Значит, всего операций произведено: $1 \cdot (N+1) = N+1 = O(N)$.
2. Строки псевдокода: 6-8 – проходят по первым элементам каждой строки и присваивают им значение 0. Следовательно 1-а операция на 1-у итерацию, а всего итераций $(W+1)$. Значит, всего операций произведено: $1 \cdot (W+1) = O(W)$.
3. Строки псевдокода: 9-16 – последовательно заполняют каждую из ячеек целочисленного двумерного массива D. В 1-ой итерации содержится присвоение и сравнение или присвоение, сравнение и присвоение. В случае выполнение условия: в 1-ой итерации 3 операции, а в случае невыполнения условия 2 операции. А всего итераций $(W+1) \times (N+1)$. Значит всего операций, в случае постоянного выполнения условия, $(W+1) \cdot (N+1) \cdot 3$, а в случае постоянного невыполнения условия, $(W+1) \cdot (N+1) \cdot 2$.

$$(W+1) \cdot (N+1) \cdot 3 = 3WN + 3W + 3N + 3 = O(WN)$$
$$(W+1) \cdot (N+1) \cdot 2 = 2WN + 2W + 2N + 2 = O(WN)$$

4. Строки псевдокода: 18-20 – представляют из себя последовательное выполнение каждой из описанных операций. Значит, общее число выполненных операций:
$$3 = O(3) = O(1)$$
5. Строки псевдокода: 21-27 – совершают обход строк и столбцов матрицы D, пока не будет достигнут первый элемент строки. В одной итерации всегда не менее 1-ого присвоения и строго 1-о сравнение, исключая условие цикла. Таким образом, в случае постоянного выполнения условия, – все рассматриваемые предметы добавляются в емкость – на одну итерацию приходится 4 операции: 3 присвоения и 1-о сравнение. В случае постоянного невыполнения условия, на одну итерацию приходится 2 операции: присвоение и сравнение. Всегда итераций $N + 1$. Значит, всего операций произведено:
$$4 \cdot (N + 1) = O(N) = 2 \cdot (N + 1)$$
6. Строки псевдокода: 28 – совершают 1-у операцию возврата оптимального набора стоимостей предметов. $1 = O(1)$

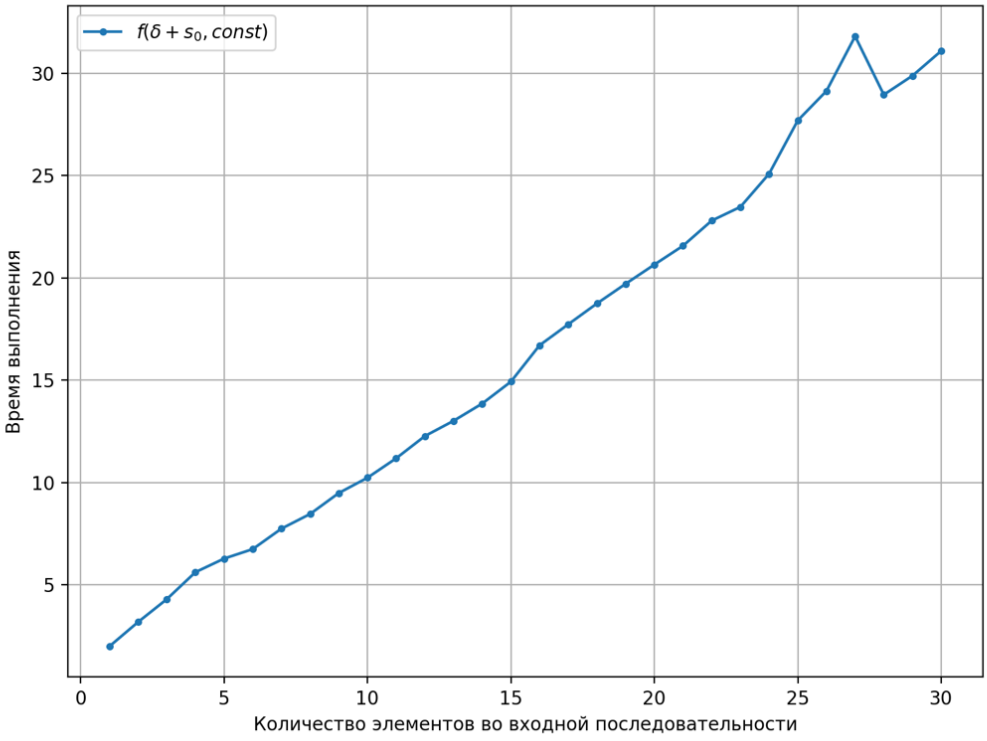
Таким образом, общее время работы алгоритма равно:

$$T\big(f(s_0 : |s_0| = N, c_0 : c_0 = M)\big) = O(N) + O(W) + O(WN) + O(1) + O(N) + O(1)$$

Функции $O(W), O(N), O(1)$ растут не быстрее, чем $O(WN)$

Значит, $T\big(f(s_0 : |s_0| = N, c_0 : c_0 = M)\big) = O(WN)$

Рассмотрим графики зависимости времени работы алгоритма от размера входа посредством: увеличения только количества исходных предметов, увеличения только максимального кэшбека, увеличения каждого из параметров. Приведенное ниже – и во всех остальных частях работы – время измерено в секундах.



На всем промежутке, исключая выбросы при 25, 26, 27, связь очень похожа на линейную. Покажем, что асимптотически – при увеличении значения по оси абсцисс – связь, при

Рисунок 4 – зависимость времени выполнения алгоритма только от количества доступных для выбора покупок – количества доступных предметов

предположении, что величина кэшбека является константой, линейна.

Утверждение: зависимость времени выполнения алгоритма только от количества доступных для выбора покупок – количества доступных предметов – линейна

Доказательство: $O(WN): W \in \mathbb{N} = O(N)$

Следствие: $O(WN): N \in \mathbb{N} = O(W)$

Графически следствие выглядит следующим образом:

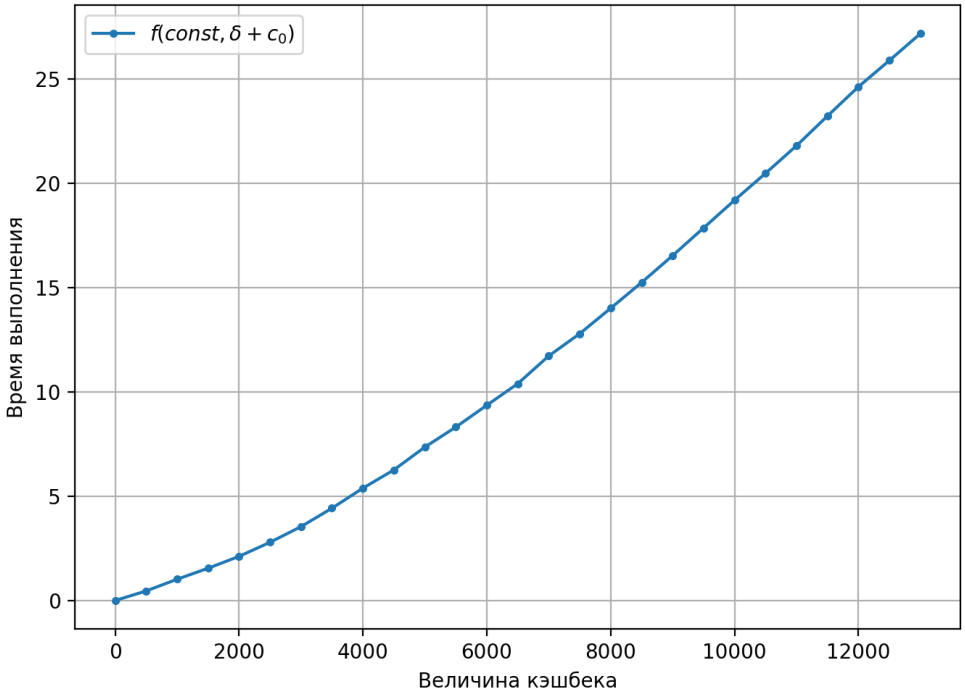


Рисунок 5 – зависимость времени выполнения алгоритма только от величины кэшбека

Действительно, несмотря на параболическую структуру на отрезке от 0 до 6’000, асимптотически – при увеличении значения по оси абсцисс – график линеен. Значит, пока что практические результаты соответствуют теоретическим выкладкам, однако вернемся к этой ситуации позже. Теперь рассматриваем общий случай: время работы алгоритмы при увеличении каждого из параметров.

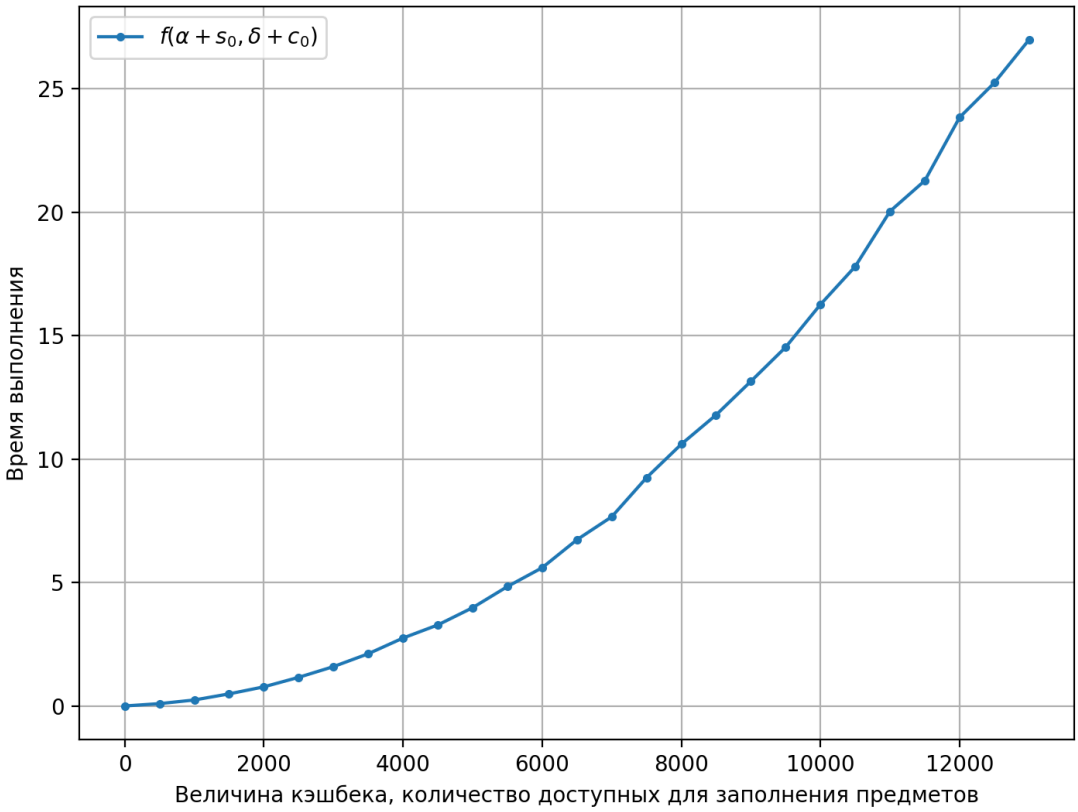


Рисунок 6 – зависимость времени выполнения алгоритма от величины кэшбека и количества предметов

Заметна связь, похожая на квадратическую. Значит, действительно, теоретические выкладки соответствуют настоящему результату. Остается два вопроса.

Вопрос 1: верно ли сделано предположение, что связь квадратическая?

Ответ: нет, неверно, так как длина входа – параметра c_0 – пропорциональна не W, но $\log(W)$. Значит, правильно оценивать размер входных данных не от N и W, а от N и $\log(W)$, где под $\log(W)$ понимается количество разрядов числа W.

NB! Основание степени логарифма не имеет значения, так как:

$$\log_b(W) = \frac{\log_c(W)}{\log_c(b)} = O(\log_c(W)) : b, c \in (0,1) \cup (1, +\infty)$$

То есть основание можно легко менять, что приведет к умножению на константу, игнорируемую при вычислении асимптотической сложности алгоритма.

Вывод: на самом деле время работы алгоритма не полиномиальное, а псевдополиномиальное и равно: $O\left(N2^{\log(W)}\right)$, что несомненно позволяет получить $O(NW)$, при условии, что основание логарифма равно 2.

Вопрос 2: откуда появилась связь параболического вида на рисунке 5?

Ответ: исходя из ответа на вопрос 1 получаем утверждение

Если: $N \in \mathbb{N}$

То: $O(NW) = O(W) \neq$ линейна связь, но $= O\left(2^{\log(W)}\right)$

Значит, это не параболическая связь, а экспоненциальная, что заметно ухудшает картину времени работы. Сравним скорость роста $g(x) = 2^{\log_c(x)}: c \in (0,1) \cup (1, +\infty)$ и $\mathrm{h}(x) = x^2$ Для этого воспользуемся теорией пределов:

$$\lim_{x \to +\infty} \frac{g(x)}{h(x)} = \lim_{x \to +\infty} \frac{2^{\log_c(x)}}{x^2} = \frac{+\infty}{+\infty} = \text{неопредленность}$$

Проверим выполнение условий правила Лопиталя[8]:

Если:

1. $\lim\limits_{x \to +\infty} x^2 = \lim\limits_{x \to +\infty} 2^{\log_c(x)} = +\infty$
2. $\frac{dh}{dx} = x \neq 0$
3. $\exists \lim\limits_{x \to +\infty} \frac{g'(x)}{h'(x)}$

То: $\lim\limits_{x \to +\infty} \frac{2^{\log_c(x)}}{x^2} = \lim\limits_{x \to +\infty} \frac{2^{\log_c(x)} \cdot \ln(2) \cdot \frac{1}{\ln(c) \cdot x}}{2x}$

Данное преобразование не очень помогло. Попробуем иначе и рассмотрим определение $O(.)$[9].

Если:

1. $g(n), h(n): \mathbb{N} \to \mathbb{R}^+$
2. $\exists a \in R^+ \setminus \{0\}: \forall n \in \mathbb{N} => g(n) \leq a \cdot h(n)$

То: $g(n) = O\big(h(n)\big)$

$$2^{\log_c(x)} \; vs \; a \cdot x^2$$

Если: $c = \sqrt{2}$

То: $2^{\log_{\sqrt{2}}(x)} vs \; a \cdot x^2 => 2^{\log_2(x^2)} \; vs \; a \cdot x^2 => x^2 \; vs \; a \cdot x^2 => 1 \leq a: x^2 \neq 0 \wedge a \in [1, +\infty)$

Значит: $\lim\limits_{x \to +\infty} \frac{g(n)}{h(n)} = \begin{cases} 0, c > \sqrt{2} \\ 1, c = \sqrt{2} \\ +\infty, c < \sqrt{2} \end{cases} => \begin{cases} 2^{\log_c(n)} = O(n^2): c \in (\sqrt{2}, +\infty) \\ n^2 = \Theta\big(2^{\log_c(n)}\big): c = \sqrt{2} \\ n^2 = O\big(2^{\log_c(n)}\big): c \in (1, \sqrt{2}) \end{cases}$

Таким образом, вышеописанное решение – в случае $N \in \mathbb{N}$ - имеет в самом плохом случае вычислительную сложность: $O\big(2^{\log_c(n)}\big)$. Сравнение производилось исключительно с n^2, так как была необходимость показать, что данная задача относится к классу NP полных задач. Значит, точного решения за полиномиальное время пока что не найдено. Таким образом: алгоритм построен, правильность показана, сложность вычислена. Остается один вопрос: можно ли лучше? Да, но с оговорками: существует способ аппроксимации, чья вычислительная сложность является полиномиальной. Главная задача подобного алгоритма, применяющего методы динамического программирования – определить точное решение с точностью $1 - \epsilon$, где ϵ – ошибка: вероятность ошибки второго рода: решение оптимальное, хотя на самом деле – это не так. Основная идея приближения решения задачи о рюкзаке за полиномиальное время заключается в снижении точность, то есть – жертвуем точностью,

чтобы сократить время: предполагаем, что некоторые предметы приблизительно равны по своей стоимости и объединяем их в группы. Далее, применяя алгоритм аппроксимации оптимального набора предметов, получаем полиномиальную вычислительную сложность от N и $\frac{1}{\epsilon}$. Однако для аппроксимации необходимо иметь точку отсчета. Для получения стартового решения используем жадный алгоритм[10]. Рассмотрим примеры решений задачи о кэшбеке с использованием метода решения задачи о рюкзаке. Далее приводятся таблицы, используемые алгоритмом для получения оптимального решения.

Пример 1

j	0	1	2	3	4	5	6
w	40	6	23	10	5	9	X
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	5	5
6	0	0	6	6	6	6	6
7	0	0	6	6	6	6	6
8	0	0	6	6	6	6	6
9	0	0	6	6	6	6	9
10	0	0	6	6	10	10	10
11	0	0	6	6	10	11	11
12	0	0	6	6	10	11	11
13	0	0	6	6	10	11	11

Данные	Единицы измерения
Вход [1]: Набор покупок: 0.4 0.06 0.23 0.10 0.05 0.09	(все в д.е.)
Вход [2]: Максимальный кэшбек: 0.13	(д.е.)
Полученный кэшбек: 0.11	(д.е.)
Остаток: 0.13 - 0.11 = 0.02	(все в д.е.)
Выход [1]: Использованные покупки: 0.05 0.06	
NB! В таблице все величины уже умноженены на 100	
$f([40,6,23,10,5,9], 13) \rightarrow [6,5]$	
NB! Введенное ранее обозначение алгоритма решения	

Рисунок 7 – 1-ый пример работы описанного алгоритма

Пример 2

j	0	1	2	3	4	5	6	7	8	9	10
w	40	6	23	10	5	9	12	54	65	23	X
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	5	5	5	5	5	5
6	0	0	6	6	6	6	6	6	6	6	6
7	0	0	6	6	6	6	6	6	6	6	6
8	0	0	6	6	6	6	6	6	6	6	6
9	0	0	6	6	6	6	9	9	9	9	9
10	0	0	6	6	10	10	10	10	10	10	10
11	0	0	6	6	10	11	11	11	11	11	11
12	0	0	6	6	10	11	11	12	12	12	12
13	0	0	6	6	10	11	11	12	12	12	12
14	0	0	6	6	10	11	14	14	14	14	14
15	0	0	6	6	10	15	15	15	15	15	15
16	0	0	6	6	16	16	16	16	16	16	16
17	0	0	6	6	16	16	16	17	17	17	17
18	0	0	6	6	16	16	16	18	18	18	18
19	0	0	6	6	16	16	19	19	19	19	19
20	0	0	6	6	16	16	20	20	20	20	20
21	0	0	6	6	16	21	21	21	21	21	21
22	0	0	6	6	16	21	21	22	22	22	22
23	0	0	6	23	23	23	23	23	23	23	23
24	0	0	6	23	23	23	24	24	24	24	24
25	0	0	6	23	23	23	25	25	25	25	25
26	0	0	6	23	23	23	25	26	26	26	26
27	0	0	6	23	23	23	25	27	27	27	27

Данные	Единицы измерения
Вход [1]: Набор покупок: 0.40 0.06 0.23 0.10 0.05 0.09 0.12 0.54 0.65 0.23	(все в д.е.)
Вход [2]: Максимальный кэшбек: 0.27	(д.е.)
Полученный кэшбек: 0.27	(д.е.)
Остаток: 0.27 - 0.27 = 0	(все в д.е.)
Выход [1]: Использованные покупки: 0.10 0.05 0.12	
NB! В таблице все величины уже умноженены на 100	

$$f([40,6,23,10,5,9,12,54,65,23], 27) \rightarrow [10,5,12]$$

NB! Введенное ранее обозначение алгоритма решения

Рисунок 8 – 2-ой пример работы описанного алгоритма

Пример 3

j	0	1	2	3	4	5	6	7	8	9	10
w	40	6	23	10	5	9	12	54	65	23	X
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0

Данные	Единицы измерения
Вход [1]: Набор покупок: 0.40 0.06 0.23 0.10 0.05 0.09 0.12 0.54 0.65 0.23	(все в д.е.)
Вход [2]: Максимальный кэшбек: 0.03	(д.е.)
Полученный кэшбек: 0.00	(д.е.)
Остаток: 0.03 - 0.0 = 0.03	(все в д.е.)
Выход [1]: Использованные покупки:	
NB! В таблице все величины уже умноженены на 100	

$$f([40,6,23,10,5,9,12,54,65,23], 3) \rightarrow []$$

NB! Введенное ранее обозначение алгоритма решения

NB! Величина кэшбека слишком мала, чтобы быть разложенной на суммы покупок

Рисунок 9 – 3-ий пример работы описанного алгоритма

Пример 4

j	0	1	2	3	4
w	10	10	6	1	X
0	0	0	0	0	0
1	0	0	0	0	1
2	0	0	0	0	1
3	0	0	0	0	1
4	0	0	0	0	1
5	0	0	0	0	1
6	0	0	0	6	6
7	0	0	0	6	7
8	0	0	0	6	7
9	0	0	0	6	7
10	0	10	10	10	10
11	0	10	10	10	11
12	0	10	10	10	11
13	0	10	10	10	11
14	0	10	10	10	11
15	0	10	10	10	11
16	0	10	10	16	16
17	0	10	10	16	17
18	0	10	10	16	17
19	0	10	10	16	17
20	0	10	20	20	20
21	0	10	20	20	21
22	0	10	20	20	21
23	0	10	20	20	21
24	0	10	20	20	21
25	0	10	20	20	21
26	0	10	20	26	26
27	0	10	20	26	27

Данные	Единицы измерения
Вход [1]: Набор покупок: 0.1 0.1 0.06 0.01	(все в д.е.)
Вход [2]: Максимальный кэшбек: 0.27	(д.е.)
Полученный кэшбек: 0.27	(д.е.)
Остаток: 0.27 - 0.27 = 0	(все в д.е.)
Выход [1]: Использованные покупки: 0.1 0.1 0.06 0.01	
NB! В таблице все величины уже умноженены на 100	
$f([10,10,6,1], 27) \rightarrow [10,10,6,1]$	
NB! Введенное ранее обозначение алгоритма решения	
NB! Использованы все доступные суммы	

Рисунок 10 – 4-ый пример работы описанного алгоритма

В итоге получаем готовый вариант работающего алгоритма, решающего задачу о кэшбеке посредством сведения ее к задаче о рюкзаке и использования методов Динамического Программирования.

Заключение и выводы

В заключение проведем анализ работы и проверим, действительно ли выполнены поставленные в самом начале цели и задачи.

Главной целью текущей работы было дать возможность каждому человеку, связанному с финансовой сферой, сберегать наибольшую сумму, обращающуюся в виде цифровой наличности, посредством предоставления алгоритма решения задачи о кэшбеке, что в результате приведет к росту величины спроса на товары различных секторов экономики со стороны рассматриваемого индивида. И как следствие к росту ВВП страны в целом, так как каждый из потребителей сможет приобретать больше товаров, необходимых ему. Значит, довольство собственными условиями жизни будет увеличиваться, что повысит производительность каждого занятого в экономике. А если это утверждение верно для конкретного индивида при нефиксированных его чертах как финансовых [где и сколько хранится средств, статистика использование данных средств, статистика приобретаемых товаров и так далее], так и личных [личные данные, место работы, занимаемая должность], то его [данное утверждение] можно обобщить на случай “для любого” индивида верно. Это приводит к росту производительности всех экономических агентов в целом.

Вывод:

1. Достигнуто более активное развитие в научной области: экономика, математика, информатика, медицина и так далее.
2. Конкурентоспособность страны увеличивается по отношению к другим участникам мировой экономики.
3. ВВП страны увеличивается.

Все вышеперечисленное достигается посредством упрощения жизни человека. Так, изложенный алгоритм решения позволяет обеспечить людей большими денежными средствами, так как само по себе решение не позволяет банку, проводящему соответствующие политики возврата средств, забирать у клиента большую сумму, чем тот могу бы получить, если бы потратил больше собственных сил и времени. Однако подобного рода затраты невозможны в среде напряженной ежедневной деятельности среднестатистического занятого в экономике, что вынуждает клиента банка мириться с отсутствием возможно жизненно необходимых средств. Чтобы избежать подобных разочарований разработан алгоритм, позволяющий за псевдополиномиальное время получить решение задачи, разбор которой вручную занял бы в разы больше сил. Однако, несмотря на удобство применения, трудность остается во времени его работы. Вторым параметром, принимаемым функцией на вход, является величина кэшбека, которая далее рассматривается как последовательность элементов

от 0 до заданной величины соответственно, что заставляет смотреть не на само значение данного числа, а на его порядок. Именно это и делает вычислительную сложность задачи псевдополиномиальной. Например, для переданных в алгоритм аргументов вида: $s_0 = [1,2,3,4], c_0 = 73'442'346'787'432'123'568$ – время выполнения будет настолько большим, что для современных компьютеров это неприлично много. Причина этого - c_0, [порядок - всего 20 цифр!] если записать в текстовый файл, то получится 21 байт, что очень мало по сравнению с невероятным значением данного числа. В заключение, нельзя не сказать, что рассмотренный алгоритм безусловно позволяет достичь поставленной цели, однако основная трудность все-таки остается неразрешенной – необходимо свести задачу о рюкзаке к классу P-полных задач оптимизации. Это позволит в разы уменьшить объем времени, используемого человеком на решение проблемы экономии, что приведет к более продуктивной работе.

Приложения

Вопрос: если Динамическое Программирование упирается в проблему порядка числа, отвечающего за величину кэшбека, то можно ли использовать другие методы, чтобы получить тот же ответ, но за меньшее время?

Ответ: внимательнее смотрим на математическую формализацию поставленной задачи

$$\begin{cases} x^* = \arg\left(\min\left(M - \sum_{j=1}^{|M_l|} M_{l,j}\right)\right) \\ \sum_{j=1}^{|M_l|} M_{l,j} \le M, l \in \sum_{k=0}^{|G|=N} \binom{|G|}{k} \\ \forall j \in \{1, \dots, N\}, \forall l \in \sum_{k=0}^{|G|=N} \binom{|G|}{k} => M_{l,j} \in Q^+ \end{cases}$$

Соображения:

1. Необходимо получить на выходе множество элементов.
2. Множество должно удовлетворять одному условию: сумма элементов данного множества не превосходит ранее заданную величину.

Замечания:

1. Используемое ограничение является линейным.
2. Абстрагируемся от предпосылки использования лишь методов динамического программирования.
3. Множество оптимальных сумм счётно.

Новая постановка:

1. Пусть $x^* = \{x_j^*\}_j^m : m = |M_l|$
2. То есть элементы множества M_l – это x_j^*

$$\begin{cases} c^T x \to max \\ c^T x \le M \\ x \in Binary^n \end{cases}$$

Анализ системы:

1. $c: c \in Q^{n,+}$ - вектор сумм покупок за период.
2. $x: x \in Binary^n$ – вектор “решений”: выбирать данную сумму или нет.
3. $c^T x \le M$ – сумма покоординатных произведений данных векторов не превосходит заданной величины кэшбека.

Идея: вектор x – вектор “решений”: использовать ли данную сумму для достижения кэшбека.

Новая формулировка: представление задачи о кэшбеке в виде задачи линейного программирования.

Пример

NB! Чтобы избежать введенного ранее умножения на 100, предположим, что на вход подаются только числа из $\mathbb{N} \cup \{0\}$.

Вход:

1. $2, 5, 7, 12, 64, 11, 9, 0$ – затраты за период. 0 – фиктивная трата, указанная исключительно для примера.
2. 33 – величина кэшбека.

Конкретный вид задачи:

1. $c = [2,5,7,12,64,11,9,0]^T$
2. $x = [x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8]^T$
3. $M = 33$
4. $n = 8$

$$\begin{cases} c^T \cdot x \to max \\ c^T \cdot x \le M \\ x \in Binary^8 \end{cases} = \begin{cases} 2x_1 + 5x_2 + 7x_3 + 12x_4 + 64x_5 + 11x_6 + 9x_7 + 0x_8 \to max \\ 2x_1 + 5x_2 + 7x_3 + 12x_4 + 64x_5 + 11x_6 + 9x_7 + 0x_8 \le 33 \\ \forall j \in \{1, \dots, 8\} => x_j \in \{0,1\} \end{cases}$$

Решение:

1. Можно применить brute force – брут форс – алгоритм полного перебора. Вычислительная сложность $O(2^n)$ и определяется двумя показателями:
    a) Количеством состояний для одной покупки: 2. Учитываем/не учитываем.
    b) Количеством покупок за период: n.
    c) Алгоритм просмотрит каждую из комбинаций покупок и найдет удовлетворяющую заданному условию.
    d) В общем случае: вычислительная сложность $O(k^n): k \in \mathbb{N}$
2. Однако $2^{\log(n)} = O(2^n)$, так как $\lim\limits_{n \to +\infty} \frac{\log(n)}{n} = 0$. Нас не интересует алгоритм, вычислительная сложность которого превосходит ранее $O\left(2^{\log(n)}\right)$
3. Применим стандартный $simplex$ метод, основанный на исследовании допустимого множества, образованного ограничениями системы. В нашем случае, всего одно ограничение, более того, $\forall j \in \{1, \dots, 8\} => x_j \in \{0,1\}$, значит, наше допустимое множество – дискретный набор точек, которых как раз не более 2^n. “Не более”, так как величина кэшбека может “отсекать” некоторые комбинации, исходя из заданного набора сумм. Например, $f([1,2,4], 1) \to [1]$, а 2 и 4 не учитывались в принципе.

Оценка вычислительной сложности

Вычислительная сложность такого алгоритма составляет - в самом плохом случае посещена каждая точка допустимого множества - $O(2^n)$: подробнее по ссылкам 11-12. На первый взгляд, не лучше, чем обычный brute force, однако исходя из логики $simplex$ алгоритма очевидно, что в процессе решения точки, лежащие внутри допустимого множества, не будут посещены, что сокращает вычислительную сложность. Интерес же представляют только “вершины” допустимого множества – в нашем случае, все граничные точки пространства возможных вариантов решения. Именно “граничные точки”, а не “вершины”, так как допустимое множество дискретно. Важно отметить, что для слегка возмущенных входных данных $simplex$ алгоритм находит решение за полиномиальное время[13]. Более того Дэниэлом Спилменом и Шан-Хуа Тенгом была представлена статья, описывающая модифицированный $simplex$ алгоритма, работающий за полиномиальное время[14]. Таким образом, вывод: для решения задачи необходимо затратить полиномиальное время. Следовательно, если рассматривать задачу о кэшбеке, то ее псевдополиномиальное время при использовании методов динамического программирования переходит в полиномиальное при использовании методов линейного программирования. Рассмотрим код соответствующей программы.

Код нового алгоритма

Идея заключается в использовании дополнительного модуля Pyomo[15], предоставленного Центром компьютерных исследований. Данная библиотека предназначена для решения разнообразных задач оптимизации: линейного программирования, нелинейного программирования, целочисленного программирования.

```
import pyomo.environ as pyo
class CashBackMachine():
    def __init__(self,c):
        self.model = pyo.ConcreteModel()
        self.model.c = pyo.Set(initialize= c)
        self.model.x = pyo.Var(range(1,len(c) + 1), domain= pyo.Binary)
        self.model.obj = pyo.Objective(expr= self.objective_func(), sense= pyo.maximize)

    def objective_func(self):
        return pyo.summation(self.model.c,self.model.x)

    def constraint(self,val):
        return pyo.summation(self.model.c,self.model.x) <= val

    def approximate(self,val):
        self.model.constr = pyo.Constraint(expr= self.constraint(val))
        opt = pyo.SolverFactory("cplex_direct")
        opt.solve(self.model)

        return (self.model,val)
```

Рисунок 11 – код алгоритма, решающего задачу о кэшбеке методами линейного программирования

Рассматриваем построчно:

1. Строки кода: 3-7 инициализация задачи с использованием заданной последовательности сумм. Без фиксации величины кэшбека.
2. Строки кода: 9-10, 12-13 – инициализация целевой функции и ограничения соответственно.
3. Строки кода: 15-18 – решения задачи оптимизации с использованием модуля “cplex-direct”, предоставленного компанией IBM.
4. Строки кода: 20 – фиксация кэшбека и возврат результата: полученной модели и заданного кэшбека.

Выход:

$$x^* = [0,1,1,1,0,0,1,0] => c^T x^* = 5 + 7 + 12 + 9 = 33 => 33 - 33 = 0 \text{ остаток}$$

Замечание: если в случае Динамического Программирования было важно использовать только натуральные числа или 0, то в случае методов Линейного Программирования можно использовать элементы множества рациональных чисел, так как на основании последовательности, поданной на вход, например, не строится двумерный массив.

Далее к работе прилагаются графики зависимости времени работы алгоритма от изменения количества поданных на вход сумм, а также величины кэшбека.

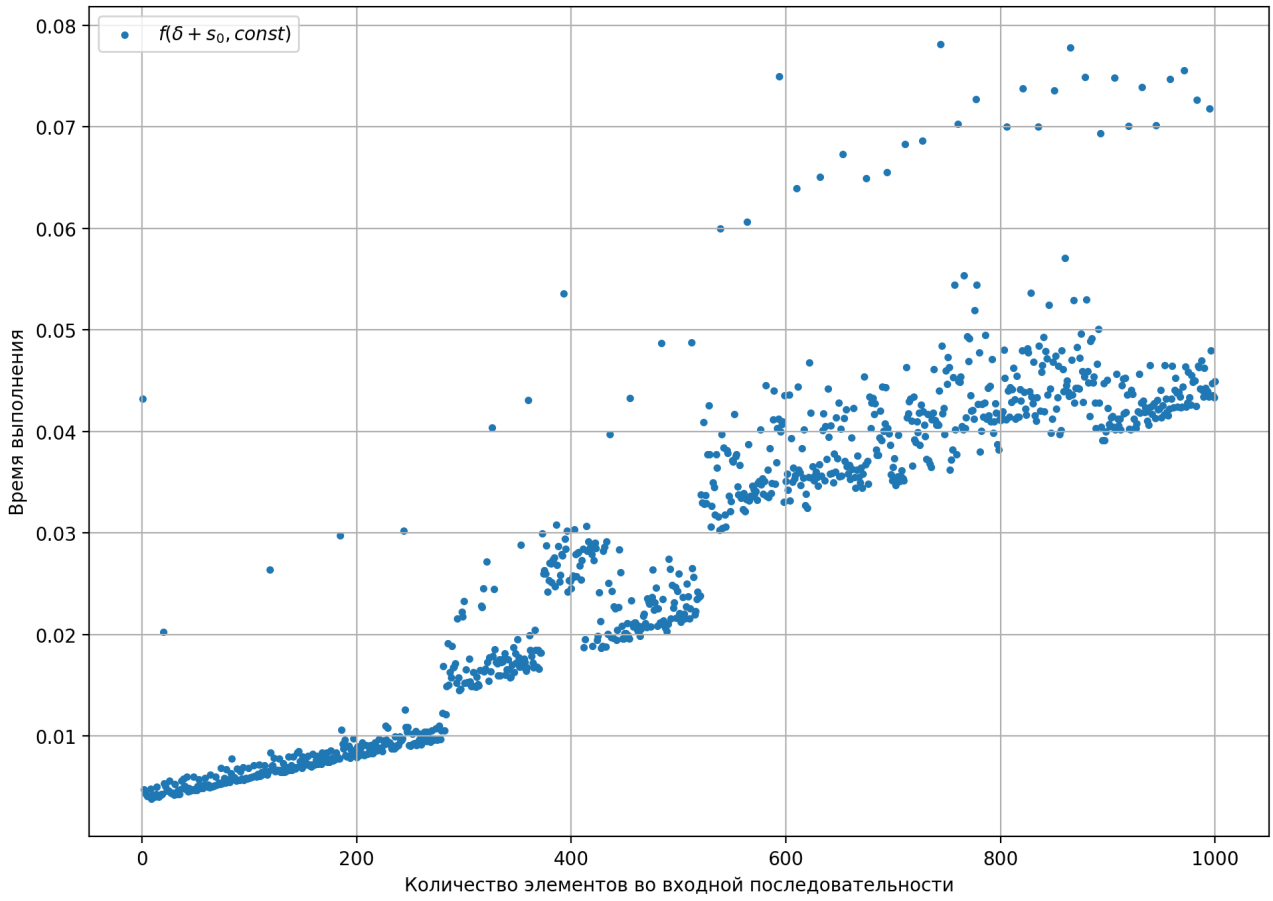


Рисунок 12 – зависимость времени работы алгоритма Линейного Программирования от количества элементов во входной последовательности сумм

На первых парах зависимость похожа на линейную, однако, рассматривая график далее, заметны скачкообразные изменения времени работы, более того, начиная с 300 элементов во входной последовательности, появляется “нестабильность” во времени работы. Это наводит на мысль о специфичности алгоритма оптимизации “cplex-direct”. Для большей уверенности в линейности связи проводим регрессионный анализ данной зависимости, используя модель python statsmodels. Применяя метод наименьших квадратов с поправкой на гетероскедастичность, получаем следующий вывод:

Показатель	**Модель 1**
Константа	0.00*** (0.00)
Количество элементов во входной последовательности	0.00*** (0.00)
Число наблюдений	1’000
$\boldsymbol{R^2}$	0.78
P(F-statistics)	0.00
NB! В скобках под коэффициентами представлено стандартное отклонение *** = значимо при 1%-ом уровне значимости.	

Уравнение значимо при 1%-ом уровне. Однако наблюдается проблема: проводя анализ исходных данных, замечаем, что среднее значение для времени выполнения программы 0.03 секунды, что достаточно близко к 0. Значит, вклад дополнительной единицы входной последовательности во время работы программы столь незначителен, что оно колеблется около 0. Но, несмотря на столь малые изменения, взглянем на график:

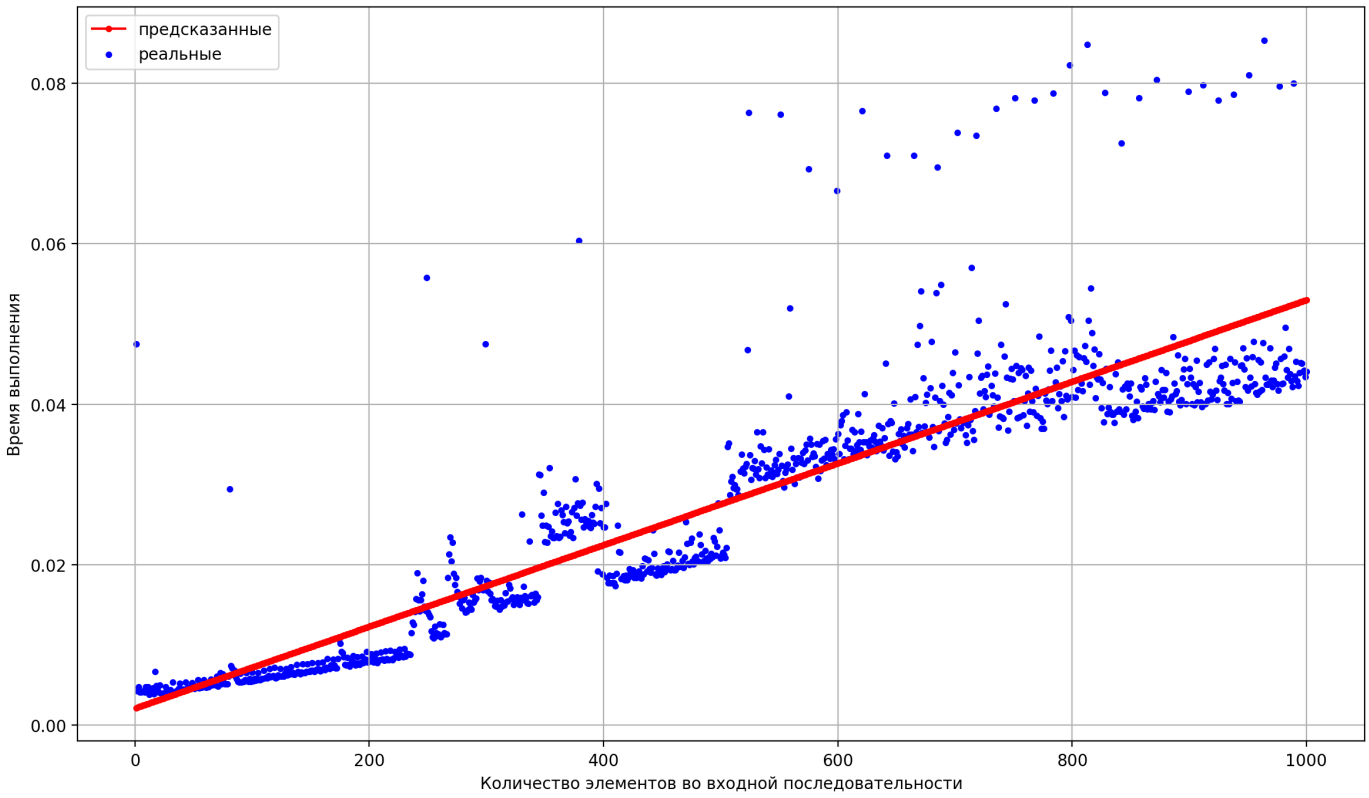


Рисунок 13 – зависимость времени работы алгоритма Линейного Программирования [и его аппроксимация] от количества элементов во входной последовательности сумм

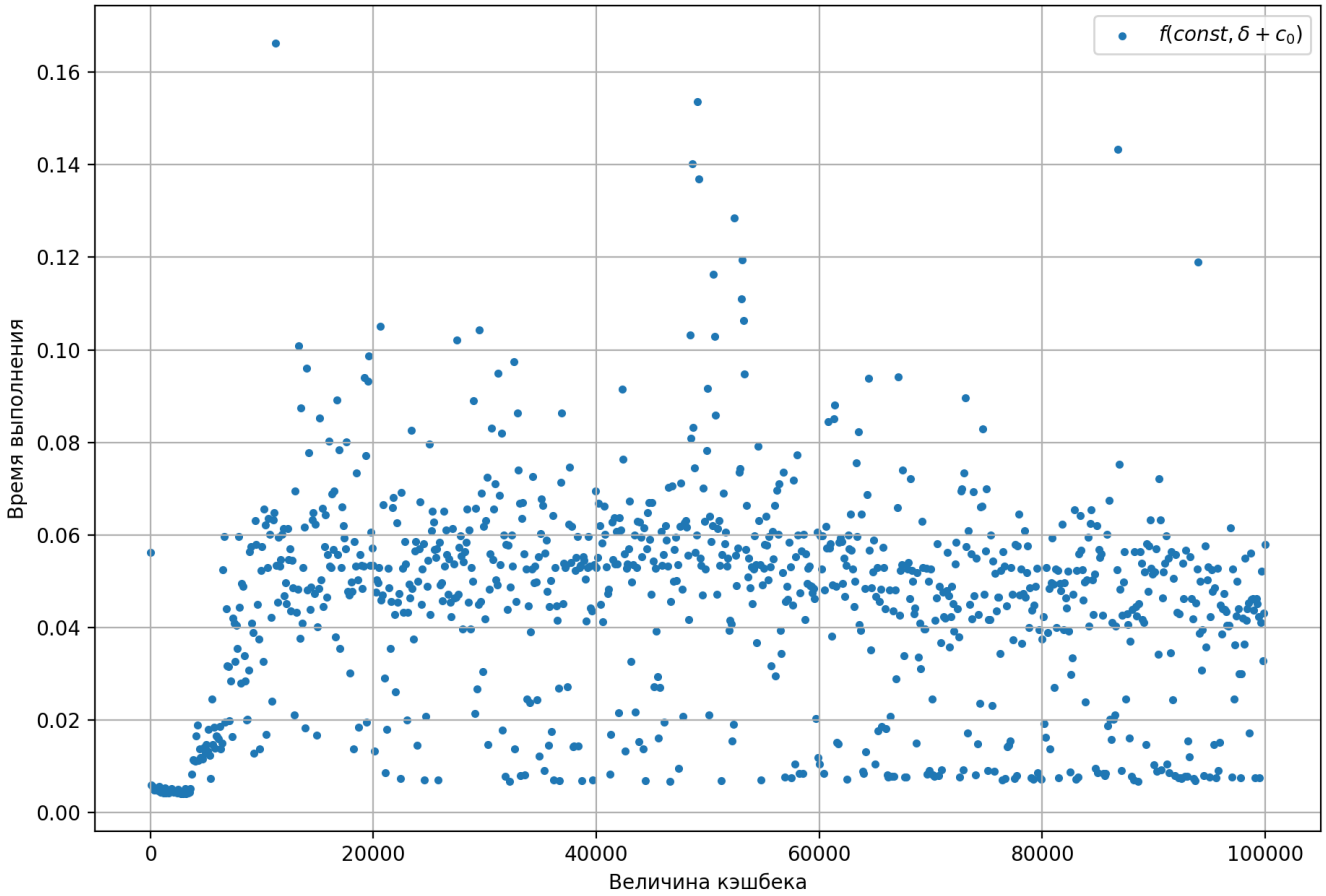
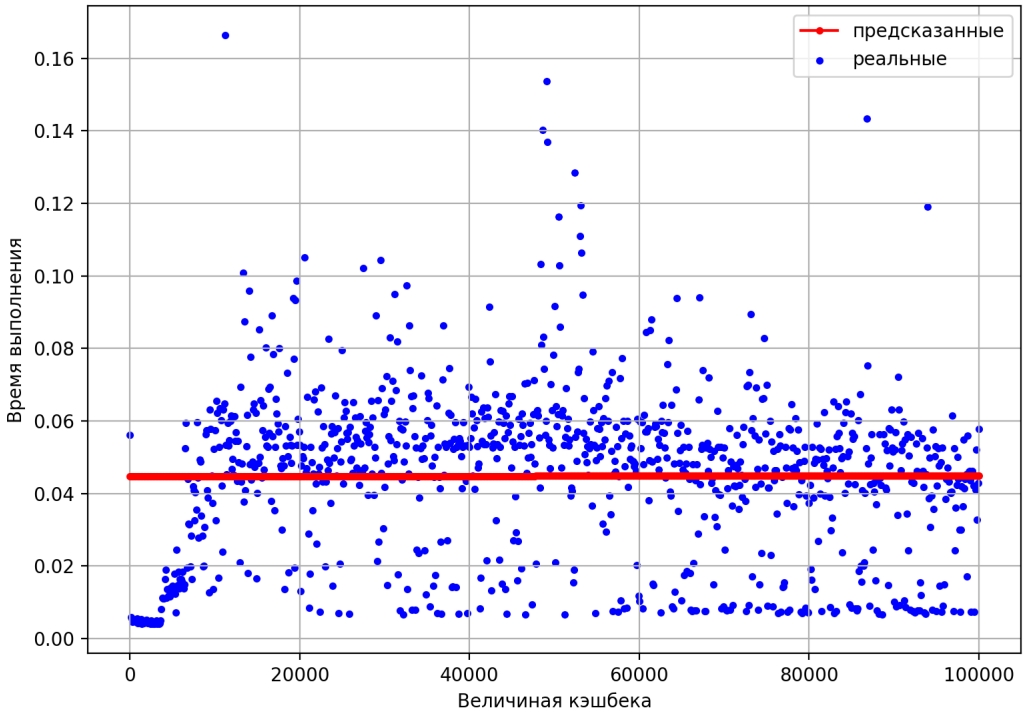


Рисунок 14 – зависимость времени работы алгоритма Линейного Программирования от величины кэшбека

На первый взгляд, связь тут не прослеживается, однако, проводя визуальный анализ, большая часть показателей колеблется в районе от 0.04 до 0.06, что позволяет говорить о незначительности изменений кэшбека. То есть изменение кэшбека в среднем при прочих равных не влияет на скорость работы алгоритма. Проводим соответствующий тест для линейной модели парной регрессии.

Показатель	**Модель 1**
Константа	0.04*** (0.00)
Величина кэшбека	0.00 (0.00)
Число наблюдений	1’000
$\boldsymbol{R^2}$	0.00
P(F-statistics)	0.94
NB! В скобках под коэффициентами представлено стандартное отклонение *** = значимо при 1%-ом уровне значимости.	

Получаем, что коэффициент при величине кэшбека не значим, то есть равен 0. Следовательно гипотеза о постоянном времени работы алгоритма при изменении величины кэшбека не отвергается. Однако нельзя полностью и всецело полагаться на данный тест, так как само уравнение незначимо, более того $\boldsymbol{R^2}$ равен около нулевому значению.



Вывод:

1. Для оценки изменения времени работы алгоритма в зависимости от величины кэшбека необходимо применять более тонкие методы.
2. Исходя из визуального анализа, на время работы алгоритма влияет только количество совершенных за период покупок.

Рисунок 14 – зависимость времени работы алгоритма Линейного Программирования [и его аппроксимация] от величины кэшбека

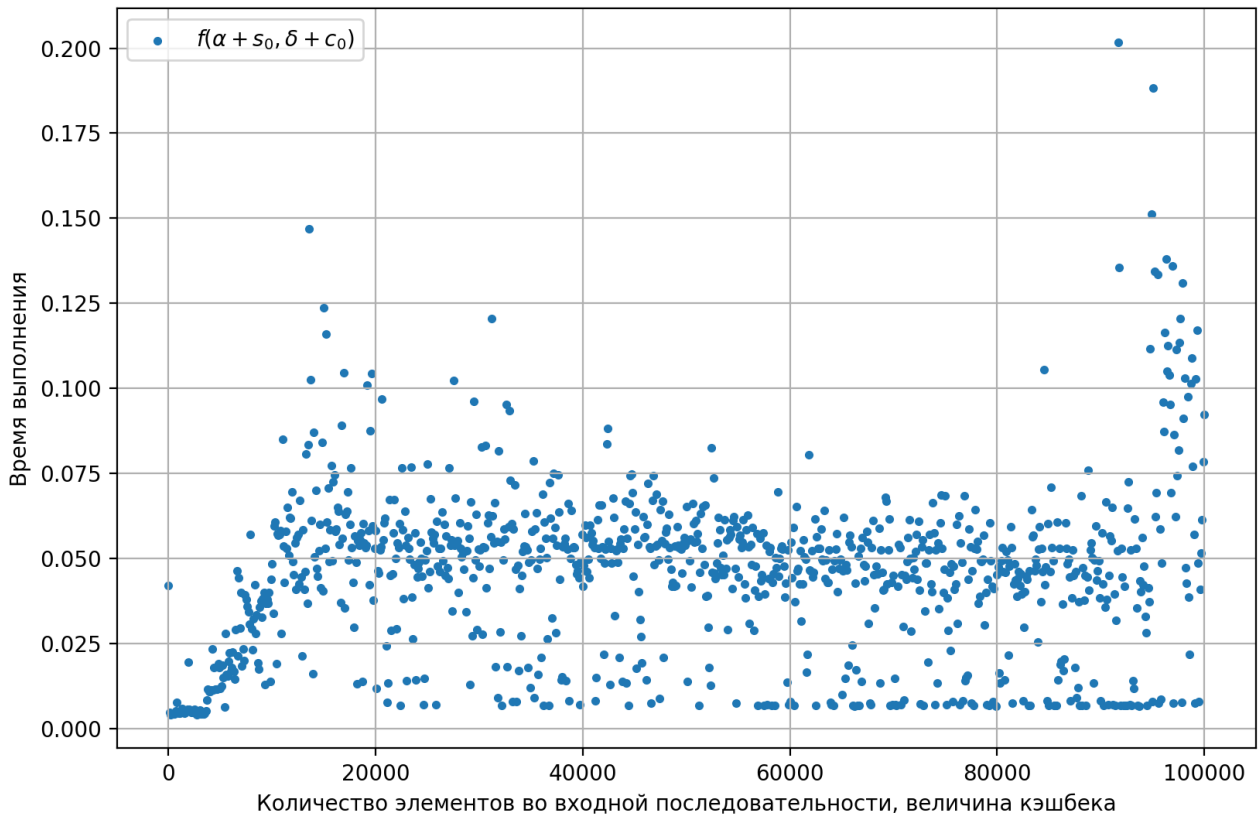


Рисунок 15 – зависимость времени работы алгоритма Линейного Программирования от величины кэшбека и количества элементов во входной последовательности

Аналогично в случае увеличения каждого из принимаемых на вход параметров: наблюдаются незначительные колебания в области 0.05 секунд. Таким образом, величина кэшбека, опираясь на визуальный анализ, в среднем при прочих равных не влияет на время работы алгоритма, что делает данный подход к решению задачи о кэшбеке более эффективным в плане вычислительной сложности, чем подход с применением методов динамического программирования.

Список литературы

1. Федеральный закон от 02.12.1990 N 395-1 (ред. от 30.12.2021) "О банках и банковской деятельности" [Электронный ресурс]. URL: https://www.consultant.ru/document/cons_doc_LAW_5842/6833df0e9ef08568539f50f01a3a53c29505430e/
2. Финансовая грамотность населения с Ставропольского края [Электронный ресурс] URL: https://fingram26.ru/articles/banki-i-bankovskie-produkty/6106/
3. Ксенофонт: Домострой. – Наука: 2004. — 122 с.
4. Платон: Государство. – Азбука: 2021. – 480 с.
5. Алгоритмы: построение и анализ, 3-е изд. : Пер. с англ. – М. : ООО “И.Д. Вильямс”, 2013 – 1328 с. : ил. – Парал. тит. англ.
6. Совершенный алгоритм. Основы. — СПб.: Питер, 2019. — 256 с.: ил. — (Серия «Библиотека программиста»).
7. Построение и анализ алгоритмов обработки данных: учеб.-метод. пособие / И. А. Селиванова, В. А. Блинов. — Екатеринбург : Изд-во Урал. ун-та, 2015. — 108 с.
8. Анализ бесконечно малых / Г.Ф. де Лопиталь; Пер. Под ред. А. П. Юшкевича – Технико-Теоретическое издательство.: Ленинград, 1935. – 337 с.
9. Алгоритмы / С. Дасгупта, Х. Пападимитриу, У. Вазирани; Пер. с англ. под ред. А. Шеня. – М.: МЦНМО, 2014. – 320 с.
10. Kellerer H., Pferschy U., Pisinger D. Knapsack Problems— Springer Science+Business Media, 2004. — 548 p.
11. Klee, V., Minty, G.J., 1972, How Good is the Simplex Algorithm? In O.Shisha editor, Inequalities III, Academic press, New York, 159-175.
12. Klee, V., Minty, G.J., 1972, How Good is the Simplex Algorithm? In Shisha, Oved. editor, Inequalities III, Proceedings of the Third Symposium on Inequalities held. Univer- sity of California.
13. Nocedal, J. and Wright, S. J. Numerical Optimization. New York: Springer-Verlag, 1999.
14. Daniel A. Spielman, Shang-Hua Teng Smoothed Analysis of Algorithms: Why the Simplex Algorithm Usually Takes Polynomial Time: Proceedings of the 33rd Annual ACM Symposium on Theory of Computing, pp. 296-305, 2001
15. Pyomo/about Pyomo [Электронный ресурс] URL: http://www.pyomo.org/about