


Дивергент, глава 3: За стеной  
Они покинут Чикаго, но найдут ли они ответы?




Half Brite English

Уникальная методика обучения. 43 года опыта в 28 странах мира.

Тест

# Задача об одномерной оптимальной упаковке



 **Helgus ~ мастер ~ Класс:** Это незавершённая статья по **эвентологии** и её применениям

**Задача об одномерной оптимальной упаковке**, или **задача о рюкзаке**, формулируется так: пусть имеется рюкзак заданной грузоподъемности; также имеется некоторое множество предметов различного веса и различной стоимости (ценности); требуется упаковать рюкзак так, чтобы он закрывался и сумма стоимостей упакованных предметов была бы максимальной.

Существует множество разновидностей этой задачи, широко используемых в практике: оптимальное заполнение контейнеров; загрузка грузовиков с ограничением по весу; создание резервных копий на съёмных носителях; выбор оптимального управления в различных экономико-финансовых операциях.

Содержание [\[развернуть\]](#)

## Постановка задачи [Править](#)

### Математическая постановка задачи [Править](#)

Пусть задано конечное множество предметов  $Q = \{q_1, q_2, \dots, q_n\}$ , для каждого  $q_i \in Q$  известна ценность (стоимость)  $c_i$  и определен вес  $a_i$ . Имеется рюкзак объема  $B$ . Требуется упаковать рюкзак так, чтобы общая ценность упакованных предметов была наибольшей и их общий вес не превосходил  $B$ . Традиционно полагают, что  $c_i, a_i, B$  - целые неотрицательные числа.

Введем двоичные переменные  $x_1, x_2, \dots, x_n$ :

- $x_i = 1$ , если предмет  $q_i \in Q$  выбран для упаковки,
- $x_i = 0$  в противном случае.

Тогда задача о рюкзаке сводится к следующей задаче линейного целочисленного программирования с булевыми переменными: найти такие значения переменных  $x_1, x_2, \dots, x_n$ , при которых достигается максимум суммы

$$W = \sum_{i=1}^n c_i x_i, \tag{1.1}$$

и выполняется ограничение

$$\sum_{i=1}^n a_i x_i \leq B, \tag{1.2}$$

Если имеется только одно ограничение вида **(1.2)**, то задачу о рюкзаке называют *одномерной*, в противном случае - *многомерной*.

### Сложность задачи [Править](#)

Далее, чтобы определить сложность задачи о рюкзаке, рассмотрим существующие классы сложности алгоритмов и задач.

Алгоритм называется *полиномиальным*, если время его работы оценивается некоторым полиномом от размерности задачи  $n$ . Полиномиальные алгоритмы считаются «хорошими», практически эффективными. Алгоритм называется *экспоненциальным*, если его выполнение требует количество операций, увеличивающихся с ростом  $n$  экспоненциально. Такие алгоритмы высокостратные, поэтому их считают *плохими*.

Под *сложностью задачи* принято понимать минимальную из сложностей алгоритмов, решающих эту задачу.

При разработке алгоритмов можно наблюдать, что для некоторых задач удается построить алгоритмы полиномиальной сложности. Такие задачи называют *полиномиально разрешимыми*. Полиномиально разрешимые задачи можно успешно решать на компьютере даже в тех случаях, когда они имеют большую размерность. Для других задач не удается найти полиномиальный алгоритм, поэтому они называются *трудноразрешимыми*.

Трудноразрешаемая задача характеризуется, как правило, конечным по мощности экспоненциальным множеством вариантов, среди которых нужно найти решение, и поэтому она может быть решена алгоритмом полного перебора. Такой, например, является задача о рюкзаке. Ясно, что решение данной задачи надо искать среди  $2^n$  двоичных векторов длины  $n$ , где  $n$  — число предметов. Перебрав это экспоненциальное множество векторов, задача **(1.1)**-**(1.2)** будет решена. Очевидно, что переборный алгоритм имеет экспоненциальную сложность и может хорошо работать только для небольших размеров задачи. С ростом размера задачи число вариантов быстро растет, и задача становится практически неразрешимой методом перебора.

Классическая теория алгоритмов разделяет задачи по сложности. При этом классифицируются лишь *распознавательные задачи* — задачи, имеющие распознавательную форму. В распознавательной форме суть задачи сводится к нахождению некоторого свойства, а ее решение — один из двух ответов: «да» или «нет».

Множество всех распознавательных задач, для которых существует полиномиальный разрешающий алгоритм, образует класс **P**. При этом распознавательные трудноразрешимые задачи не принадлежат классу **P**. Распознавательные задачи, которые можно решить за полиномиальное время на недетерминированной машине Тьюринга, входят в класс **NP**. Для некоторых задач из этого класса обнаружено удивительное свойство. Оказалось, что некоторые из них универсальны в том смысле, что построение полиномиального алгоритма для любой такой задачи влечет за собой возможность построения такого же алгоритма для всех остальных задач класса **NP**. Такие задачи называются **NP**-полными.

К настоящему времени установлена ***NP***-полнота большого числа задач, в том числе и задачи о рюкзаке. В распознавательной форме задача о рюкзаке формулируется так: имеется конечное множество предметов  $Q = \{q_1, q_2, \dots, q_n\}$ , для каждого  $q_i \in Q$  задана ценность (стоимость)  $c_i$  и определен вес  $a_i$ , а также объем рюкзака  $B$  и стоимость  $C$ . Считается, что  $a_i, c_i, B, C$  — целые неотрицательные числа. Существует ли такое подмножество  $Q' \subseteq Q$  что

$$\sum_{i \in Q'} a_i \leq B, \tag{1.3}$$

$$\sum_{i \in Q'} c_i \leq C. \tag{1.4}$$

Обзор известных методов решения задачи [Править](#)

Часто возникающие на практике ***NP***-полные задачи настолько важны, что отказаться от их решений невозможно. Конечно, нет надежд построить для них полиномиальный алгоритм. Однако это еще не означает, что с данной задачей вообще ничего нельзя сделать. Во-первых, может оказаться, что некоторый экспоненциальный алгоритм, например, переборный, работает приемлемое время на реальных данных. Во-вторых, можно попытаться найти полиномиальный алгоритм, дающий не оптимальное решение задачи, а близкое к нему, т.е. приближенное решение. Найти приближенное решение может быть вполне достаточным для практического применения.

К приближенным методам для задачи о рюкзаке относят:

- генетические алгоритмы;
- алгоритмы муравьиной колонии;
- жадные алгоритмы и др.

Для этих алгоритмов характерна, как правило, полиномиальная сложность. Цена этому — приближенное решение.

*Жадный алгоритм* для задачи о рюкзаке состоит в следующем:

- вначале множество предметов  $Q$  упорядочивается по убыванию «удельной ценности» (или цены единицы веса) предметов, т. е. так, чтобы 
$$\frac{c_1}{a_1} \geq \frac{c_2}{a_2} \geq \dots \geq \frac{c_n}{a_n};$$
- затем, начиная с пустого множества, к приближенному решению  $Q'$  (сначала оно пустое) последовательно прибавляются предметы из упорядоченного множества  $Q$ ;
- при каждом очередном добавлении предмета в рюкзак выполняется проверка, не превосходит ли его вес величины оставшегося запаса объема рюкзака;
- окончание просмотра всех предметов завершает процесс построения приближенного решения задачи о рюкзаке.

Найти точное решение задачи о рюкзаке можно одним из трёх основных методов дискретного программирования:

- отсечения;
- ветвей и границ;
- динамического программирования;

Идея метода отсечения состоит в снятии условия целочисленности и поиске оптимального решения двойственным симплекс-методом. Ввод дополнительного ограничения позволяет получить целочисленное оптимальное решение. Примером реализации этой идеи может служить алгоритм Гомори. Основной недостаток метода отсечения — плохая сходимость к целочисленному решению.

Метод ветвей и границ сводится к построению дерева возможных вариантов, определению оценки границы решения для каждой вершины дерева, отсечению бесперспективных вершин. Эффективность метода ветвей и границ зависит от удачного определения оценки границы решения. Плохая оценка может может приводить к перебору почти всех вариантов решения.

Метод динамического программирования базируется на принципе оптимальности Беллмана. Данный метод получил своё развитие в методе последовательного анализа вариантов. Метод динамического программирования при некоторых исходных данных способен существенно сократить полный перебор.

Все указанные точные методы решения задачи о рюкзаке в худшем случае (при любых исходных данных, включая самые неблагоприятные) имеют экспоненциальную сложность. Этот факт предопределяет класс сложности задачи о рюкзаке.

Оптимальная упаковка рюкзака как задача динамического программирования [Править](#)

Суть метода динамического программирования [Править](#)

В основу метода динамического программирования положен *принцип оптимальности*, сформулированный в 1957 г. американским математиком Ричардом Беллманом: «Оптимальное поведение обладает тем свойством, что таковы бы ни были первоначальные состояние и решение в начальный момент времени, последующие решения должны составлять оптимальное поведение относительно состояния, получающегося в результате первого решения».

Физическая сущность принципа оптимальности заключается в том, что ошибка выбора решения в данный момент не может быть исправлена в будущем.

Рассматривается следующая общая задача. Имеется некоторая физическая система, в которой происходит какой-то процесс, состоящий из  $n$  шагов. Эффективность процесса характеризуется некоторым показателем  $W$ , который называют *выигрышем*. Пусть общий выигрыш  $W$  за все  $n$  шагов процесса складывается из выигрышей на отдельных шагах

$$W = \sum_{i=1}^n w_i, \tag{2.1}$$

где  $w_i$  — выигрыш на  $i$ -м шаге. Если  $W$  обладает таким свойством, то его называют *аддитивным критерием*.

Процесс, о котором идет речь, представляет собой управляемый процесс, т.е. имеется возможность выбирать какие-то параметры, влияющие на его ход и исход, причем на каждом шаге выбирается какое-то решение, от которого зависит выигрыш на данном шаге. Это решение называется *шаговым управлением*. Совокупность всех шаговых управлений представляет собой управление процессом в целом. Обозначим его буквой  $U$ , а шаговые управления — буквами  $u_1, u_2, \dots, u_n$ . Тогда

$$U = (u_1, u_2, \dots, u_n).$$

Шаговые управления  $u_1, u_2, \dots, u_n$  в общем случае не числа, а, как правило, векторы, функции и т.п.

В модели динамического программирования процесс на каждом шаге находится в одном из состояний  $S$  множества состояний  $S$ . Считается, что всякому состоянию сопоставлены некоторые шаговые управления. Эти управления таковы, что управление, выбранное в данном состоянии при любой предыстории процесса, определяет полностью следующее состояние процесса. Обычно выделены два особых состояния:  $s_0$  — начальное и  $s_w$  — конечное.

Итак, пусть каждому состоянию  $s_i \in S$  поставлено множество допустимых шаговых управлений  $U_{s_i}$  и каждому шаговому управлению  $u \in U_{s_i}$  соответствует  $s_j = \sigma(s_i, u) \in S$  — состояние, в которое процесс попадает из  $s_i$  в результате использования шагового управления  $u$ . Пусть процесс находится в начальном состоянии  $s_0$ . Выбор  $u_1 \in U_{s_0}$  переводит процесс в состояние  $s_1 = \sigma(s_0, u_1)$ . выбор  $u_2 \in U_{s_1}$  — в состояние  $s_2 = \sigma(s_1, u_2)$  и т.д. В результате получается траектория процесса, которая состоит из последовательности пар

$$(s_0, u_1), (s_1, u_2), \dots, (s_w, u_{w+1})$$

и заканчивается конечным состоянием. Для единообразия можно считать, что  $U_{s_w}$  включает только одно состояние  $u_{w+1}$ , оставляющее процесс в том же конечном состоянии. Следует отметить, что множества допустимых состояний и управлений

$$S = \{s_0, s_1, \dots, s_w\}, \tag{2.2}$$

$$\sum_{s_i \in S} U_{s_i}.$$

(2.3)

конечны и  $U_s$  для различных  $s$  не пересекаются.

В общем виде задача динамического программирования формулируется следующим образом: найти такую траекторию процесса, при которой выигрыш **(2.1)** будет максимальным.

То управление, при котором достигается максимальный выигрыш, называется *оптимальным управлением*. Оно состоит из совокупности шаговых управлений

$$U^* = (u_1^*, u_2^*, \dots, u_n^*).$$

(2.4)

Тот максимальный выигрыш, который достигается при этом управлении обозначим  $W_{max}$ :

$$W_{max} = \max_U \{W(u)\}.$$

(2.5)

Рассмотрим на примере задачи о рюкзаке, что понимается под шагом, состоянием, управлением и выигрышем.

Загрузку рюкзака можно представить себе как процесс, состоящий из  $n$  шагов. На каждом шаге требуется ответить на вопрос: взять данный предмет в рюкзак, или нет? Таким образом, шаг процесса — присваивание переменной  $x_i$  значения 1 или 0.

Теперь определим состояния. Очевидно, что текущее состояние процесса характеризует остаточная грузоподъёмность рюкзака — вес, который остался в нашем распоряжении до конца (до полной укладки рюкзака). Следовательно, под состоянием перед  $i$ -м шагом понимается величина

$$s_{i-1} = b - \sum_{k=1}^{i-1} a_k x_k, \quad i = 2, \dots, n, \tag{2.6}$$

при этом  $s_0$  является начальным состоянием, которому соответствует величина  $b$  — исходная грузоподъёмность рюкзака.

Управление на  $i$ -м шаге означает присваивание двоичной переменной  $x_i$  значения 0 или 1. Значит, на каждом шаге имеем всего два управления. Причем допустимость управления  $u_i$ , устанавливающего  $x_i = 1$ , определяется условием

$$s_i = \sigma(s_{i-1}, u_i) = s_{i-1} - a_i x_i = b - \sum_{k=1}^i a_k x_k \geq 0. \tag{2.7}$$

Далее везде вместо переменных  $x_1, x_2, \dots, x_n$  будем использовать соответствующие управления  $u_1, u_2, \dots, u_n$ . Тогда формулы **(2.6)**, **(2.7)** примут следующий вид:

$$s_{i-1} = b - \sum_{k=1}^{i-1} a_k u_k, \quad i = 2, \dots, n, \tag{2.8}$$

$$s_i = \sigma(s_{i-1}, u_i) = s_{i-1} - a_i u_i = b - \sum_{k=1}^i a_k u_k \geq 0. \tag{2.9}$$

Шаговый выигрыш можно определить как  $w_i = c_i u_i$ . Поэтому

$$W = \sum_{i=1}^n w_i = \sum_{i=1}^n c_i u_i. \tag{2.10}$$

Требуется найти оптимальное управление  $U^* = (u_1^*, u_2^*, \dots, u_n^*)$  при котором величина выигрыша **(2.10)** обращается в максимум.

Уравнение Беллмана для задачи о рюкзаке [Править](#)

Пусть к началу  $n$ -шага остаточная грузоподъёмность равна  $s$ . Оптимальное управление определяется следующим образом:

- если  $s - a_n \geq 0$ , то последний предмет можно положить в рюкзак, что соответствует оптимальному управлению
- $$U_n(s) = u_n = 1;$$
- иначе  $U_n(s) = u_n = 0$ .

Ясно, что оптимальный условный выигрыш на  $n$ -ом шаге составит

$$W_n(s) = c_n u_n.$$

Рассмотрим  $(n - 1)$ -й шаг. Предположим, что остаточная грузоподъёмность равна  $s$ . Если на этом шаге выбрать управление  $u$ , то на начало последнего шага останется вес  $s - a_{n-1} u$ . Тогда выигрыш двух последних шагах будет равен

$$c_{n-1} u + W_n(s - a_{n-1} u).$$

Нужно найти такое  $u$ , при котором этот выигрыш максимален

$$W_{n-1}(s) = \max\{c_{n-1} u + W_n(s - a_{n-1} u)\},$$

где максимум берется по всем допустимым управлениям — управлениям, для которых верно ограничение **(2.9)**. Напомним, что ***u*** может принимать лишь два значения: 0 или 1.

Рассуждая далее аналогичным образом, приходим к рекуррентному уравнению

$$W_i(s) = \max\{c_i u + W_{i+1}(s - a_i u)\},$$

(2.11)

которое позволяет для любого *i*-го шага вычислить условный оптимальный выигрыш и найти соответствующее ему условное оптимальное управление *U<sub>i</sub> (s) = u\**. Здесь ***u\**** — значение, при котором достигается максимум в **(2.11)**.

К достоинствам метода динамического программирования следует отнести

- сравнительную простоту расчетов, удобство их алгоритмизации;
  - независимость от вида исходных данных (целевая функция и функции ограничений могут быть линейными и нелинейными, целочисленными и нецелочисленными, заданными аналитически и таблично).

Недостатки метода

- большой объём промежуточной информации;
  - большой объём вычислительной работы.

При увеличении числа ограничений типа **(2.9)** множество состояний процесса может быть слишком велико для проведения практических расчетов даже с использованием компьютера. Именно этим обстоятельством в первую очередь определяется сфера применения динамического программирования.

Метод динамического программирования относят к рекуррентным методам, поскольку он позволяет построить оптимальное решение с помощью рекуррентного уравнения Беллмана **(2.11)**. Таким образом можно говорить о рекурсивном характере метода. Метод динамического программирования допускает две реализации: табличную и рекурсивную. Традиционно этот метод излагается с использованием таблиц. Поэтому его также называют табличной техникой или программированием на основе массивов.

Реализации метода динамического программирования для задачи о рюкзаке

Править

Табличная реализация и её сложность

Править

Идея постепенной, пошаговой оптимизации лежит в основе табличной реализации метода динамического программирования. Принцип динамического программирования отнюдь не предполагает, что каждый шаг оптимизируется отдельно, независимо от других. Напротив, шаговое управление должно выбираться дальновидно, с учетом всех его последствий в будущем.

Общий принцип, лежащий в основе решения задач динамического программирования: каково бы ни было состояние процесса перед очередным шагом, надо выбирать управление на этом шаге так, чтобы выигрыш на данном шаге плюс оптимальный выигрыш на всех последующих шагах был максимальным. Это более конкретная перефразировка принципа оптимальности Беллмана.

Однако из этого принципа есть исключение. Среди всех шагов есть один, который можно планировать без оглядки на будущее. Это последний шаг.

Поэтому процедура решения задачи динамического программирования обычно разворачивается от конца к началу. Вначале планируется последний ***n***-ый шаг. При этом делаются разные предположения о том, чем закончился предпоследний **(n — 1)**-шаг. Для каждого из этих предположений следует найти условное оптимальное управление на ***n***-ый шаг. «Условное» потому, что оно выбирается исходя из условия, что предпоследний шаг закончился так-то и так-то. Далее, «пятясь назад» оптимизируется управление на **(n — 1)**-м шаге и т.д., пока не будет достигнут первый шаг.

Предположим, что все условные оптимальные управления и условные оптимальные выигрыши найдены для всего «хвоста» процесса. Далее выполняется построение не условно оптимального, а просто оптимального управления ***U\*** = (u<sub>1</sub><sup>\*</sup>, u<sub>2</sub><sup>\*</sup>, . . . , u<sub>n</sub><sup>\*</sup>)* и вычисление не условно оптимального, а просто оптимального выигрыша ***W<sub>max</sub>***.

Таким образом, при построении оптимального решения многошаговый процесс «проходится» дважды:

- сначала — от конца к началу, в результате чего находятся условные оптимальные управления и условные оптимальные выигрыши на оставшийся «хвост» процесса;
  - после — от начала к концу, когда выписываются уже готовые рекомендации и находится оптимальное управление.

Из этих двух этапов оптимизации наиболее важным и трудоемким является первый этап. Второй практически не требует дополнительных вычислений.

Запишем принципиальную структуру обоих этапов оптимизации с помощью общих формул. Введем некоторые обозначения. Пусть

- ***W<sub>i</sub>(s)***— условный оптимальный выигрыш, получаемый на всех последующих шагах, начиная с *i*-го шага и до конца. Считаем, что процесс в начале *i*-го шага находится в состоянии ***S***;
  - ***U<sub>i</sub>(s)***— условное оптимальное управление на *i*-м шаге, которое, совместно с оптимальным управлением на всех последующих шагах, обращает выигрыш «хвоста» в максимум.

Определим условный оптимальный выигрыш ***W<sub>i</sub>(s)*** и условное оптимальное управление ***U<sub>i</sub>(s)*** для всех шагов ***i = 1, 2, . . . , n***.

Рассмотрим *i*-й шаг процесса. Предположим, что в результате **(i — 1)** предыдущих шагов процесс оказался в состоянии ***s*** и на *i*-м шаге выбрано шаговое управление ***u<sub>i</sub>***. Получаемый на этом шаге выигрыш ***w<sub>i</sub>*** зависит как от состояния ***s***, так и от примененного управления ***u<sub>i</sub>***, т.е.

$$w_i = w_i(s, u_i).$$

(3.1)

Кроме того, имеется некоторый выигрыш на всех оставшихся шагах. Согласно принципу оптимальности, будем считать, что он максимален. Чтобы найти этот выигрыш, необходимо знать следующее состояние — состояние перед **(i + 1)**-м шагом. Это новое состояние ***s'*** определяется так:

$$s' = \sigma(s, u_i).$$

(3.2)

Найдем теперь выигрыш, который получается на всех шагах, начиная с *i*-го, если на *i*-м шаге применено любое (не обязательно оптимальное) управление ***u<sub>i</sub>***, а на всех последующих — оптимальное управление. Этот выигрыш обозначим через ***W<sub>i</sub><sup>\*</sup>(s, u<sub>i</sub>)***. Тогда можно записать

$$W_i^*(s, u_i) = w_i(s, u_i) + W_{i+1} s'.$$

Или, учитывая (3.2),

$$W_i^*(s, u_i) = w_i(s, u_i) + W_{i+1}(\sigma(s, u_i)). \tag{3.3}$$

Далее, следуя принципу оптимального управления, надо выбрать такое управление  $u_i = U_i$ , при котором величина (3.3) максимальна и достигает значения

$$W_i(s) = \max_{u_i \in U_s} \{w_i(s, u_i) + W_{i+1}(\sigma(s, u_i))\}. \tag{3.4}$$

Управление, в котором достигается максимум в (4.4), и есть условное оптимальное управление  $U_i(s)$  на  $i$ -м шаге, а сама величина  $W_i(s)$  — условный оптимальный выигрыш на всех шагах, начиная с  $i$ -го и до конца. В уравнении (3.4) функции  $w_i(s, u_i)$  и  $\sigma(s, u_i)$  известны. Неизвестными остаются функции  $W_i(s)$  и  $W_{i+1}(s)$ .

Формула (3.4) называется основным рекуррентным уравнением динамического программирования (или *уравнением Беллмана*). Она позволяет определить функцию  $W_i(s)$  через следующую за ней по порядку функцию  $W_{i+1}(s)$ .

Условный оптимальный выигрыш на последнем шаге  $W_n(s)$  можно найти так:

$$W_n(s) = \max_{u_i \in U_{s_w}} \{w_i(s, u_i)\}. \tag{3.5}$$

Управление  $U_n(s)$ , при котором достигается максимум выигрыша (3.5), является условным оптимальным управлением на последнем шаге.

С помощью формулы (3.5) и основного рекуррентного уравнения (3.4) можно, одно за другим, построить всю цепочку условных оптимальных управлений и условных оптимальных выигрышей.

Оптимальное управление  $U^* = (u_1^*, u_2^*, \dots, u_n^*)$  можно создать следующим образом.

- Выполним подстановку начального состояния  $s_0$  в формулу для условного оптимального выигрыша  $W_1(s)$  и условного оптимального управления  $U_1(s_0)$ . В результате получим

$$W_{max} = W_1(s_0), u_1^* = U_1(s_0).$$

- Далее определим следующее состояние процесса, в которое оно попадает под управлением  $u_1^*$

$$s^* = \sigma(s_0, u_1^*).$$

- Исходя из состояния  $s_1^*$ , находим  $u_2^* = U_2(s_1^*)$ .  $s_1^* = \sigma(s_1^*, u_2^*)$  и т.д. Таким образом, получаем цепочку

$$s_0 \rightarrow u_1^* \rightarrow s_1^* \rightarrow u_2^* \rightarrow \dots \rightarrow s_{n-1}^* \rightarrow u_n^* \rightarrow s_n^* = s_w. \tag{3.6}$$

На этом процесс оптимизации заканчивается выигрышем  $W_{max} = W_1(s_0)$ . Из описания процедуры оптимизации следует, что для метода динамического программирования целочисленность исходных данных решаемой задачи не является обязательной. Главное, чтобы число шагов, состояний и управлений было конечно.

Рассмотрим итерационную реализацию метода динамического программирования на конкретном числовом примере. Результаты расчетов будем вносить в табл. 3.1

Пример 3.1.

Имеется 3 предмета и рюкзак, способный выдержать 50 кг. Первый предмет весит 10 кг и стоит 60 у. е. Второй предмет весит 20 кг и имеет стоимость 100 у. е. Третий предмет весит 30 кг и стоит 120 у. е. Требуется оптимально заполнить рюкзак предметами, не превысив его веса.

Таблица 3.1

s	i=3		i=2		i=1	
	$U_3(s)$	$W_3(s)$	$U_2(s)$	$W_2(s)$	$U_1(s)$	$W_1(s)$
0	0	0	0	0	0	0
10	0	0	0	0	1	60
20	0	0	1	100	0	100
30	1	120	0	120	1	160
40	1	120	0	120	1	180
50	1	120	1	220	0	220

Поскольку грузоподъёмность рюкзака и веса предметов кратны 10, в табл. 3.1. всего лишь шесть строк, соответствующих возможным значениям остаточной грузоподъёмности рюкзака. Здесь  $i$  — номер предмета для упаковки. Согласно алгоритму начинаем заполнять таблицу с последнего предмета (т. е. идем в обратном порядке). Последний предмет весит 30 кг. Следовательно, он будет помещен в рюкзак, если  $s \geq 30$ . Тогда  $U_3(s) = 1$  и  $W_3(s) = c_3 = 120$ .

Далее, чтобы вычислить значения  $U_i(s)$  и  $W_i(s)$  для  $i = 1, \dots, n - 1$ , требуется строить для каждой пары вспомогательные таблицы.

Табл. 3.2 демонстрирует вычисление величин  $U_2(s)$ ,  $W_2(s)$  для  $s = 30$  по формуле (2.11). Соответственно ищутся величины  $U_1(s)$ ,  $W_1(s)$  для  $s \geq 10$  по той же формуле.

Таблица 3.2

$u$	$30 - a_2u$	$c_2u$	$W_3(30 - a_2u)$	$c_2u + W_3(30 - a_2u)$
0	30	0	120	120
1	10	100	0	100

Итак, метод динамического программирования дает следующее оптимальное управление процессом укладки рюкзака

$$U^* = (u_1^*, u_2^*, u_3^*) = (0, 1, 1)$$

с оптимальным выигрышем  $W_{max} = 220$  у. е.

Оценим сложность решения задачи о рюкзаке методом динамического программирования. Наиболее трудоёмким является первый этап процедуры поиска оптимального решения. Этот этап сводится к построению основной таблицы (табл. 3.1). Чтобы найти значения для первого шага, требуется  $2B$  ресурсов. Для вычисления любой пары

элементов  $U_i(s), W_i(s)$  где  $i = 1, 2, \dots, n - 1$ . основной таблицы приходится формировать вспомогательную таблицу типа 4.2. Построение основной таблицы для общего случая задачи о рюкзаке будет зависеть от множества состояний рюкзака, а также от количества управлений и количества предметов. То есть асимптотическая функция сложности принимает вид:

$$t(n) = O(|S| \cdot n \cdot |k|), \tag{3.7}$$

где  $|k|$  — мощность множества управлений, в нашей задаче оно состоит из 0 и 1, а  $|S|$  — мощность множества состояний **(2.2)**(наихудший случай для  $|S|$  в задаче о рюкзаке это число, равное размерности рюкзака, то есть  $B$ ). Значит, для формирования вспомогательных таблиц  $t(n) = 8B(n - 1)$ . Таким образом, приходится вычислять  $2B + 8B(n - 1)$  элементов, что составляет  $O(Bn)$ .

Рекурсивная реализация и её сложность [Править](#)

Рекурсивность метода динамического программирования позволяет реализовать его в виде рекурсивной процедуры. Эта рекурсивная процедура определяется рекуррентным соотношением, которое следует из основного функционального уравнения Беллмана **(2.11)**:

$$\begin{aligned} W_1(s) &= c_1, \quad i = 1, \\ W_i(s) &= \max u\{c_i u + W_{i-1}(s - a_i u)\}, \quad i > 1. \end{aligned} \tag{3.8}$$

На языке Delphi схема рекурсивной процедуры имеет вид:

```
type

mass = [0..n-1] of integer;    // тип массив, n-количество
                                // предметов

const

k=2;    // Число возможных состояний: 0 или 1
B=50;    // высота рюкзака

var

    optium_var,    // максимальная стоимость, результат работы
                  // функции
    Cost,          // вспомогательная переменная для вычисления
                  // стоимости на i-ом шаге
    Max,           // вспомогательная переменная для вычисления
                  // максимального значения стоимости
    : integer;

    i: byte;    // счетчик

    A, C: mass;    // массивы стоимости и высоты предметов

function Optium(B,x); // Поиск оптимального решения BEGIN

    if x=1 then begin
        if A[1] < B then optium_var:=C[1]
        else optium_var:=0;
    end
    else begin
        Max:=0;
        for i:=0 to k-1 do begin
            Cost:=i*C[x]+Optium((B-A[x])*i, x-1);
            if Cost > Max then begin
                Max:=Cost;
                Opt_x:=i;
            end;
        end;
        Optium_var:=Max;
        Print(Optium_var);
    end;

END;
```

Из текста процедуры видно, что если укладывается один предмет, то первая ветвь процедуры нерекурсивная. Трудоёмкость нерекурсивной ветви не зависит от  $n$ , поэтому для неё  $t(n) = c_0 > 0$  - константа.

Вторая ветвь процедуры является рекурсивной, описывающей процесс упаковки  $n > 1$  предметов. Согласно второму соотношению из **(3.8)**, в рекурсивной ветви задачи размерности  $n$  сводятся к двум задачам размерности  $n - 1$ . Временные затраты на определение максимума составляют некоторое постоянное число  $c_1 > 0$  единиц времени. Таким образом, трудоёмкость рекурсивной ветви составляет  $t(n) = 2t(n - 1) + c_1$ .

В общем случае временная сложность рекурсивной процедуры задается функцией, удовлетворяющей рекуррентному соотношению

$$\begin{aligned} t(n) &= c_0, \quad n = 1 \\ t(n) &= 2t(n - 1) + c_1, \quad n > 1. \end{aligned} \tag{3.9}$$

Соотношение **(3.9)** - линейное неоднородное рекуррентное соотношение с постоянными коэффициентами. Его можно решить различными методами.

Самый простой из них - вторая основная теорема о рекуррентных соотношениях, дающая решение уравнений, характерных для рекурсии, организованной путём аддитивного уменьшения размерности на некоторую константу. Эта теорема даёт следующее решение для **(3.9)**:



$$t(n) = 2^n (c_0 + c_1) - c_1.$$

(3.10)

Из (3.10) следует что временная сложность рекурсивной реализации метода динамического программирования для задачи о рюкзаке является экспоненциальной, т. к.  $t(n) = O(2^n)$ . это вполне соответствует классу сложности рассматриваемой задачи.

Сравнение алгоритмов по сложности

Править

Рассмотрим вычислительную сложность полного перебора. Исходя из того, что полный перебор составляет  $2^n$  вариантов, каждый из которых предполагает обработку из  $n$  предметов (проверка условия и вычисление целевой функции), то для него  $t(n) = O(n2^n)$ .

Сравним по сложности точные алгоритмы, которые были рассмотрены в работе. Как было найдено, сложность табличной реализации метода динамического программирования равна  $O(Bn)$ . Получается, что временная и ёмкостная сложности процедуры решения задачи о рюкзаке линейны по числу предметов  $n$  и росту значения  $B$ . Но, оказывается, что временная сложность порядка  $O(Bn)$  для рассматриваемой задачи — это иллюзия линейной сложности относительно  $n$ . Для задачи о рюкзаке единственным параметром, характеризующим её размерность является число предметов  $n$ , а значение  $B$  — конкретное значение одного из исходных данных задачи. Именно число предметов определяет длину входа

$$c_1, c_2, \dots, c_n, a_1, a_2, \dots, a_n, B$$

всякого алгоритма решения задачи о рюкзаке. По значению  $B$  (вместе с  $a_i$ ) неравенство (1.2) лишь ограничивает множество допустимых решений задачи. В худшем случае значения  $a_1, a_2, \dots, a_n, B$  могут быть такими, что в рюкзак можно уложить  $n - 1$  предметов. Но тогда оптимальное решение приходится выбирать почти из  $2^n$  возможных решений. Таким образом, для табличной реализации число состояний при больших значениях  $n$  асимптотически пропорционально  $2^n$ , т. е.  $|S| = O(2^n)$ . Применяя для вычисления сложности табличного алгоритма формулу (3.7) получаем  $t(n) = O(n|S|) = O(n2^n)$ . Следовательно, в худшем случае сложность табличной реализации сопоставима со сложностью полного перебора. Поскольку худший случай дает пессимистическую оценку сложности алгоритма, то на многих исходных данных значение грузоподъёмности рюкзака может сильно ограничивать область поиска оптимального решения.

Для рекурсивной реализации временная сложность равна  $t(n) = O(2^n)$ . Получается, что наиболее приемлимой по сложности является именно рекурсивная реализация метода динамического программирования. Известно, что рекурсивные алгоритмы, порождаемые методом динамического программирования, в большинстве своем приводят к неполиномиальной сложности вычислений даже для полиномиально разрешимых задач (например, для задачи о кратчайшем пути в ориентированном графе). Возникает вопрос: почему для **NP**-трудной задачи о рюкзаке рекурсия оказывается более быстрой, чем итерация? Объяснить это можно следующим образом. Дело в том, что в табличной реализации метода динамического программирования используется принцип инвариантного погружения. В основе этого принципа лежит замена исходной задачи на более общую, в результате решения которой находится решение исходной задачи. Например, в задаче о рюкзаке оптимальное решение находится не только для начального состояния  $s_0 = B$ , но и для всякого другого состояния из  $S$ .

Рекурсивная реализация находит оптимальное решение только для заданного объёма рюкзака. Если попробовать приравнять задачи двух реализаций метода динамического программирования, то оценка рекурсивной реализации станет  $t(n) = O(2^n) \cdot O(2^n) = O(2^{2n})$ , это значительно хуже чем  $t(n) = O(n2^n)$ .

В некоторых частных случаях решение рассматриваемой задачи о рюкзаке вообще может иметь полиномиальную асимптотическую сложность. Рассмотрим один из них, когда веса всех предметов равны, а различной является лишь стоимость. Тогда решение задачи сводится к упорядочению всех предметов по стоимости (от большей к меньшей). В этом случае оценка временной сложности алгоритма сопоставима со сложностью алгоритма сортировки. Если применить самый простой метод сортировки — метод «пузырька», то  $t(n) = O(n^2)$

Заключение

Править

В настоящей статье были исследованы с точки зрения временной сложности три алгоритма нахождения точного решения задачи о рюкзаке:

- алгоритм полного перебора;
- табличная реализация метода динамического программирования;
- рекурсивный алгоритм, непосредственно реализующий уравнение Беллмана.

Найденные асимптотические оценки этих алгоритмов показали, что

- в худшем случае самым приемлимым является рекурсивная реализация;
- табличный алгоритм позволяет получить оптимальное решение не только для заданного объёма рюкзака  $B$ , но и для всех промежуточных дискретных значений этого объёма;
- если рекурсивный алгоритм применить к такой же (более общей задаче), то он начинает работать хуже полного перебора.

Полученные результаты позволяют сформулировать следующие практические рекомендации: рекурсивные алгоритмы, реализованные методом динамического программирования, для **NP**-трудных задач большой размерности допустимы; для полиномиально разрешимых задач они могут приводить к неоправданно большим вычислительным затратам и поэтому их лучше не использовать.

Литература

Править

- Беллман, Р. Прикладные задачи динамического программирования / Р. Беллман, С. Дрейфус; перевод с англ. Н. М. Митрофановой. — М.: Наука, 1965. — 460 с.
- Быкова, В. В. Дискретная математика с использованием ЭВМ / В. В. Быкова. — Красноярск: РИО КрасГУ, 2006. — 200 с.
- Вирт, Н. Алгоритмы и структуры данных / Н. Вирт. — М.: Мир, 1989. — 360 с.
- Грин, Д. Математические методы анализа алгоритмов / Д. Грин, Д.-Кнут. — М.:Мир, 1987. — 120 с.
- Кормен, Т. Алгоритмы. Построение и анализ: [пер. с англ.] / Томас Кормен, Чарльз Лейзерсон , Рональд Ривест, Клиффорд Штайн. — 2-е изд. — М.: Вильямс, 2005. — 1296 с.
- Пападимитриу, Х. Комбинаторная оптимизация. Алгоритмы и сложность. / Х. Пападимитриу, К. Стайглиц. — М.: Мир, 1985. — 512 с.

Ресурсы в интернете

Править

- [Поиск оптимального управления методом динамического программирования.](#)
- [Dynamic Programming.](#)
- [Dynamic Programming — Integer Knapsack.](#)
- [Knapsack Problem.](#)

Категории: [Незавершённые статьи по эвентологии](#) | [Добавить категорию](#)

You May Like

Sponsored Links by Taboola

A New MMORPG You Won't Get Bored With : Click here to try!

Видеоигры

Кино

Телевидение

Исследуйте Викия

Поиск внутри: Наука...

Войти

Создать вики

Т

Мы используем cookies, чтобы сделать Вашу работу с Викия простой и удобной. Поэтому, посещая Викия, вы соглашаетесь на использование cookies. [Подробнее о cookies](#)


Fandom

**Yoga, Showers, Nap Rooms: Why Flying Has Never Been So Good**  
Fortune Skyteam - Russia

**The Top 10 Kisses of the DC Universe**  
Fandom

**A New MMORPG You Won't Get Bored With. Click Here To Try!**  
Stormfall - Online Game

**Five Things Episode VIII Needs to Explain**  
Fandom



Facebook

Официальный Сайт

Самая популярная социальная сеть Создайте профиль бесплатно!

УВЛЕЧЕНИЯ

[О Викия](#)[Справка](#)[Центральная Вики](#)[Карьера](#)[Реклама](#)[Связаться с Викия](#)[Условия использования](#)[Конфиденциальность](#)

[Глобальная карта сайта](#)[Содержимое доступно в соответствии с CC-BY-SA.](#)

http://ru.science.wikia.com/wiki/%D0%97%D0%B0%D0%B4%D0%B0%D1%87%D0%B0\_%D0%BE%D0%B1\_%D0%BE%D0%B4%D0%BD%D0%BE... 8/8