1 Введение в структуры данных

В этой главе

- ✓ Почему стоит изучать структуры данных и алгоритмы.
- ✓ Разница между алгоритмами и структурами данных.
- ✓ Построение абстрактной задачи на основе конкретной.
- ✓ Переход от задач к решениям.

Итак, вы решили изучить алгоритмы и структуры данных. И это правильно!

Если вы все еще размышляете, есть ли такая необходимость, я надеюсь, что вводная глава развеет все сомнения и пробудит интерес к такой замечательной теме.

Зачем изучать алгоритмы? Короткий ответ: чтобы попытаться стать лучшим разработчиком программного обеспечения. Можно сказать, что знание структур данных и алгоритмов будет весьма полезным добавлением в ваш чемоданчик с инструментами.

Вы когда-нибудь слышали о молотке Маслоу, также известном как золотой молоток или закон инструмента? Это основанная на наблюдениях гипотеза, что люди, которым известен только один способ сделать что-либо, обычно применяют его во всевозможных ситуациях.

Если в чемоданчике с инструментами есть только молоток, возникнет соблазн относиться ко всему как к гвоздю. Если вы знаете только, как отсортировать список,

будет появляться искушение сортировать список задач каждый раз, когда в него добавляется новая задача или когда приходится выбирать, какую задачу взять в дальнейшую работу. Однако так вы не сумеете использовать понимание контекста, чтобы найти более эффективное решение.

Цель этой книги — дать вам множество инструментов, которые можно использовать при решении задач. Мы будем опираться на алгоритмы, обычно представленные в базовых курсах информатики в университетах (курс 101 по принятой в США кодировке), но я познакомлю вас и с более сложным материалом.

Прочитав книгу, вы должны научиться распознавать ситуации, в которых можно улучшить производительность кода, используя определенную структуру данных и/или алгоритм.

Конечно же, не стоит ставить перед собой цель запомнить наизусть все детали всех обсуждаемых структур данных. Скорее я постараюсь показать, как вникать в суть задач и где найти идеи алгоритмов, которые могут помочь в их решении. Книга также может послужить справочником, своего рода сборником рецептов, содержащим некоторые типовые сценарии и наиболее подходящие для этих сценариев структуры. В дальнейшем будет полезно классифицировать задачи по соответствию таким типовым сценариям.

Имейте в виду, что некоторые темы довольно сложны, и, когда мы углубимся в детали, может потребоваться прочесть один и тот же фрагмент текста несколько раз, чтобы все понять. Структура книги позволяет продемонстрировать несколько уровней углубленного анализа, при этом наиболее продвинутые разделы, как правило, сгруппированы в конце каждой главы, так что, если вы захотите получить только базовое понимание темы, сможете не углубляться в теорию.

1.1. СТРУКТУРЫ ДАННЫХ

Чтобы начать наше путешествие, сначала нужно договориться об общем языке для описания и оценки алгоритмов.

Способ описания алгоритмов довольно стандартен: алгоритмы описываются через принимаемые ими входные данные и возвращаемые ими выходные данные. Детали алгоритмов могут быть отражены с помощью псевдокода (без учета особенностей языков программирования) или фактического кода.

То же верно и для структур данных (СД), но для них этого недостаточно. Описывая структуру данных, необходимо также указывать и действия, которые можно с ней выполнить. Обычно каждое действие описывается как алгоритм со входом и выходом, но в дополнение к этому необходимо упомянуть и объяснить *побочные эффекты* для СД — изменения, которые действие может вызвать в самой структуре данных.

Чтобы осознать, что все это означает, нужно сначала правильно определить понятие «структура данных».

1.1.1. Определение структуры данных

Структура данных — это специальное решение для организации данных, которое предоставляет место для размещения элементов и возможность для их сохранения и извлечения[1].

Самый простой пример структуры данных — это массив. Так, массив символов обеспечивает хранилище для конечного числа символов и методы для извлечения каждого символа в зависимости от его позиции в массиве. На рис. 1.1 показано, как хранится `array = ['C', 'A', 'R']`: массив символов, в котором символы `C`, `A` и `R` размещаются таким образом, чтобы, например, элемент `array[1]` соответствовал значению `A`.

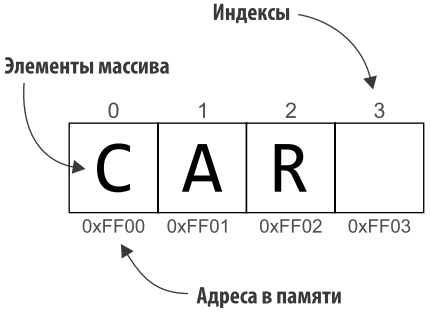

Рис. 1.1. Внутреннее представление массива (упрощенно). Каждому элементу массива на картинке соответствует байт памяти[2], адрес которого указан под ним. Над ним отображается индекс каждого элемента. Массив хранится как непрерывный блок памяти, и адрес каждого элемента может быть получен путем добавления его индекса в массиве к смещению первого элемента. Например, четвертый символ массива (array[3], на рисунке пустой) имеет адрес 0xFF00 + 3 = 0xFF03

Структуры данных могут быть абстрактными или конкретными. Между собой они соотносятся так, как указано в перечне ниже.

- *Абстрактный тип данных (АТД)* определяет операции, которые могут выполняться с некоторыми данными, и вычислительную сложность этих операций. Никаких подробностей о том, как хранятся данные или как используется физическая память, не приводится.
- *Структура данных (СД)* — это конкретная реализация спецификации, предоставляемой АТД.

---

[1] А именно, хотя бы один метод для добавления нового элемента в структуру и один метод либо для извлечения указанного элемента, либо для выполнения запроса к структуре данных.

[2] В современных архитектурах/языках элемент массива может соответствовать слову, а не байту, но для простоты предположим, что массив символов хранится как массив байтов.

ЧТО ТАКОЕ АТД

АТД можно представить как шаблон, задающий спецификации, а СД — перевод этих спецификаций в реальный код.

АТД определяется с точки зрения того, кто его использует. Пользователь описывает поведение АТД в терминах его возможных значений, возможных операций над ним, а также результатов и побочных эффектов этих операций.

Более формальное определение описывает АТД через набор типов, определенный тип из этого набора типов, набор функций и набор аксиом.

В свою очередь, структура данных, являющаяся конкретным представлением данных, описывается с точки зрения разработчика, а не пользователя.

Вернемся к примеру с массивом. Возможный АТД для статического массива — это, например, контейнер, который, во-первых, может хранить фиксированное количество элементов, каждый из которых ассоциирован с индексом (позицией элемента в массиве), и, во-вторых, предоставляет возможность обращаться к любому элементу, используя его позицию (произвольный доступ).

Обращаю ваше внимание на то, что при его реализации необходимо учесть:

- будет ли размер массива зафиксирован при создании, или его можно изменить;
- будет ли память для массива выделяться статически или динамически;
- будет ли массив содержать элементы одного типа или любого типа;
- будет ли он реализован как просто непрерывная область памяти или как объект;
- какие у него будут атрибуты?

Даже для такой базовой структуры данных, как массивы, разные языки решают перечисленные выше вопросы по-разному. Но все они обеспечивают, чтобы их версия массивов соответствовала описанному ранее АТД массива.

Еще одним хорошим примером, помогающим понять разницу между АТД и СД, может быть стек. Стеки описаны в приложениях В и Г, но я предполагаю, что вы слышали о них и раньше.

Возможное определение АТД стека таково: контейнер, который может хранить неопределенное количество элементов и удалять элементы по одному, начиная с самого последнего, в соответствии с порядком, обратным вставке.

В альтернативном его определении можно привести разбиение действий, которые могут быть выполнены с контейнером. Итак, стек — это контейнер, который поддерживает два основных метода:

- вставку элемента;
- удаление элемента: если стек не пуст, добавленный последним элемент будет возвращен и удален.

Это все еще высокоуровневое описание, но оно более четкое и модульное, чем предыдущее.

Оба определения достаточно абстрактны, чтобы их можно было легко обобщить, и это позволяет реализовать стек в широком диапазоне языков программирования, парадигм и систем[1].

Однако в какой-то момент придется перейти к конкретной реализации и обсудить детали, например те, которые приведены в следующем далеко не полном перечне.

- Где хранить элементы:
  - в массиве;
  - в связном списке;
  - в B-дереве на диске?
- Как мы будем отслеживать порядок вставки? (Это тесно связано с предыдущим вопросом.)
- Будет ли известен и фиксирован заранее максимальный размер стека?
- Может ли стек содержать элементы любого типа, или они все должны быть одного типа?
- Что произойдет, если вызвать удаление из пустого стека? (Например, это может быть возврат `null` или выдача ошибки?)

Можно продолжать задавать подобные вопросы, но, надеюсь, идея понятна.

1.1.2. Описание структуры данных

Наиболее важно при определении АТД перечислить набор операций, которые он разрешает. Это эквивалентно определению API[2], то есть контракта с клиентами.

Каждый раз, когда вам нужно описать структуру данных, можно выполнить несколько простых шагов, чтобы предоставить исчерпывающую и однозначную спецификацию.

1. Сначала указать ее API, уделяя особое внимание вводам и выводам методов.
2. Описать ее поведение на высоком уровне.
3. Подробно описать поведение ее конкретной реализации.
4. Проанализировать эффективность ее методов.

Именно такой рабочий процесс мы будем использовать для структур данных, представленных в книге, после того как опишем конкретный сценарий практического применения каждой структуры.

---

[1] В принципе, это не обязательно должно иметь отношение к информатике. Например, вы можете описать в качестве системы стопку папок, которые необходимо изучить, или — распространенный пример на уроках информатики — стопку посуды, которую нужно вымыть.

[2] Application programming interface (программный интерфейс приложения)

Во второй главе вместе с описанием первой структуры данных я более подробно объясню соглашения, которые используются для описания API.

1.1.3. Алгоритмы и структуры данных: так есть ли разница?

Это все же не совсем одно и то же; технически они не эквивалентны. Тем не менее иногда можно использовать эти два термина как синонимы и для краткости применять термин *«структура данных»* для обозначения как СД, так и всех соответствующих методов.

Есть много способов обозначить разницу между этими двумя терминами, но мне особенно нравится такая метафора: структуры данных подобны *существительным*, а алгоритмы — *глаголам*.

На мой взгляд, такое сравнение не только намекает на разницу в поведении структур и алгоритмов, но и неявно показывает взаимную зависимость между ними. Например, в английском языке, чтобы построить осмысленную фразу, нужны как существительные, так и глаголы: подлежащее (или дополнение) и выполненное им (или над ним) действие.

Структуры данных и алгоритмы взаимосвязаны; они нужны друг другу:

- структуры данных — это основа, способ организации области памяти для представления данных;
- алгоритмы — это процедуры, последовательность инструкций, направленных на преобразование данных.

Структуры данных были бы просто битами, хранящимися на микросхеме памяти, если бы не было алгоритмов их преобразования; и наоборот, без структур данных, с которыми можно было бы работать, большинство алгоритмов просто не существовало бы.

Более того, каждая структура данных сама неявно определяет алгоритмы, которые могут быть для нее выполнены, например методы для добавления, извлечения и удаления элементов в структуру данных и из нее.

Некоторые структуры данных фактически определены именно для того, чтобы один или несколько алгоритмов могли эффективно работать с ними. Вспомните о хеш-таблицах и поиске по ключу[1].

Итак, использование терминов «алгоритм» и «структура данных» как синонимов допустимо только потому, что в конкретном контексте одно подразумевает другое. Например, когда описывается СД, чтобы это описание было значимым и точным, обязательно нужно описать ее методы (то есть алгоритмы).

---

[1] Больше информации по этой теме вы найдете в приложении В.

1.2. ЦЕЛЕПОЛАГАНИЕ: ВАШИ ОЖИДАНИЯ ОТ ПРОЧТЕНИЯ ЭТОЙ КНИГИ

У вас может возникнуть такой вопрос: «Придется ли мне когда-нибудь писать собственные структуры данных?»

Весьма вероятно, что в большинстве случаев у вас будет возможность не создавать структуры данных с нуля. Сегодня нетрудно найти библиотеки, реализующие наиболее распространенные структуры данных на большинстве языков программирования, и обычно эти библиотеки написаны экспертами, которые знают, как оптимизировать производительность или позаботиться о безопасности.

По сути, основная цель книги — познакомить вас с обширным набором инструментов и научить распознавать возможности их использования для улучшения кода. Понимание внутреннего устройства этих инструментов, по крайней мере на высоком уровне, является важной частью процесса обучения. Тем не менее существуют определенные ситуации, в которых может потребоваться окунуться в код; например, если используется совершенно новый язык программирования, для которого не так много библиотек, или если нужно подстроить структуру данных под решение особого случая.

В конце концов, соберетесь ли вы писать собственную реализацию структур данных, зависит от многих факторов.

Например, от того, насколько продвинута структура данных, которая вам нужна, и насколько широко распространен язык, который вы используете.

Проиллюстрирую это на примере кластеризации.

Если вы работаете с популярным языком, например Java или Python, очень вероятно, что вы найдете множество надежных библиотек для метода *k*-средних, одного из простейших алгоритмов кластеризации.

Если же используется нишевой язык (предположим, вы экспериментируете с одним из недавно созданных, например с *Nim* или *Rust*), тогда может оказаться сложно найти библиотеку с открытым исходным кодом, реализованную командой, которая тщательно протестировала код и будет поддерживать библиотеку в дальнейшем.

Аналогично, если нужен продвинутый алгоритм кластеризации, такой как *DeLiClu*, будет непросто найти его реализации даже для Java или Python, такие, которым можно доверять в достаточной степени, чтобы запустить их в работу в производственной среде как часть вашего приложения.

Другая ситуация, в которой будет важно понимать внутреннее устройство этих алгоритмов, — возникновение необходимости подстройки одного из них. Например, может понадобиться оптимизировать его для среды реального времени, или нужно будет добавить в него какую-либо конкретную особенность (скажем, настроить его для одновременной работы и обеспечения безопасности потоков), или даже потребуется немного отличное от стандартного поведение.

Но даже если вы сосредоточитесь только на уяснении, когда и как использовать структуры данных, — а с этого начинается описание каждой из них, — уже одно это кардинально повлияет на ваше развитие как программиста, позволит поднять навыки кодирования на новый уровень. Воспользуюсь примером, чтобы показать важность алгоритмов в реальном мире, и заодно продемонстрирую способ описания алгоритмов, принятый в книге.

1.3. СОБИРАЕМ РЮКЗАК: СТРУКТУРЫ ДАННЫХ В РЕАЛЬНОМ МИРЕ

Поздравляем, вас выбрали для заселения первой марсианской колонии! В продуктовых магазинах на Марсе ситуация по-прежнему так себе... да и найти их трудновато. Так что готовьтесь, вам придется выращивать себе еду. Между тем на первые несколько месяцев вы будете обеспечены товарами для поддержания жизнедеятельности, их доставят на том же корабле, что и вас.

1.3.1. Абстрагирование задачи

Проблема в том, что ваши ящики не могут весить более 1000 кг, и это жесткое ограничение.

Задача усложняется тем, что вы можете выбирать только из ограниченного перечня продуктов, уже упакованных в коробки:

- картофель, 800 кг;
- рис, 300 кг;
- пшеничная мука, 400 кг;
- арахисовое масло, 20 кг;
- консервированные помидоры, 300 кг;
- фасоль, 300 кг;
- клубничное варенье, 50 кг.

Воду вы получите бесплатно, так что об этом беспокоиться не приходится. Что касается каждого продукта, то можно либо взять целую упаковку, либо не брать его вообще. Вы, конечно, хотели бы иметь в запасе какое-то разнообразие, а не тонну картошки (как было в «Марсианине»).

В то же время критически важно поддерживать собственное хорошее самочувствие и энергичность на протяжении всего пребывания на Марсе, поэтому главным фактором при выборе продуктов, которые отправятся с вами в путешествие, будет питательная ценность. Положим, ее хорошим индикатором будет общее число калорий. Рассмотрим табл. 1.1, которая показывает список доступных товаров с учетом новых данных.

Таблица 1.1. Перечень доступных товаров с указанием их веса и калорийности

Продукт	Вес, кг	Общее число калорий
Картофель	800	1 501 600
Пшеничная мука	400	1 444 000
Рис	300	1 122 000
Фасоль (консервированная)	300	690 000
Помидоры (консервированные)	300	237 000
Клубничное варенье	50	130 000
Арахисовое масло	20	117 800

Поскольку фактическое содержимое не имеет значения для принятия решения (несмотря на ваши понятные протесты, центр управления полетом очень тверд в этом вопросе), единственное, что принимается в рассмотрение, — это вес продуктов и общее число калорий в каждой из упаковок.

Следовательно, нашу задачу можно сформулировать в абстрактном виде так: «Выбрать любое количество предметов из набора, *не имея возможности взять часть какого-либо предмета*, при этом их общий вес не должен превысить 1000 кг и общее число калорий должно быть максимальным».

1.3.2. Поиск решений

Теперь, когда задача сформулирована, можно приступить к поиску решений. У вас может возникнуть соблазн начать упаковывать свой ящик с той коробки, в которой содержится наибольшее число калорий. Это будет ящик с картофелем весом 800 кг.

Но если вы это сделаете, ни рис, ни пшеничная мука не поместятся в ящик, а их общее число калорий намного превосходит любую другую комбинацию, которую можно создать в рамках оставшихся 200 кг. Лучшее, что можно получить при этой стратегии, — 1 749 400 калорий при выборе картофеля, клубничного джема и арахисового масла.

Итак, то, что выглядело бы наиболее естественным подходом, — *жадный*[1] алгоритм, который на каждом шаге выбирает кажущийся наилучшим здесь и сейчас вариант, — не работает. Эту задачу нужно обдумать более тщательно.

Время для мозгового штурма. Вы собираете свою логистическую команду и вместе ищете решение.

[1] Жадный алгоритм — это стратегия решения задач, которая находит оптимальное решение, делая локально оптимальный выбор на каждом шаге. Пользуясь им, можно найти лучшее решение только для небольшого подкласса задач, но его также можно использовать в качестве эвристики для поиска приближенных (неоптимальных) решений.

Вскоре кто-то предлагает смотреть не на общее число калорий, а на число калорий на килограмм. Таким образом, табл. 1.1 дополняется новым столбцом и соответствующим образом сортируется. Результат представлен в табл. 1.2.

Таблица 1.2. Список из табл. 1.1, отсортированный по числу калорий на килограмм

Продукт	Вес, кг	Общее число калорий	Число калорий на кг
Арахисовое масло	20	117 800	5890
Рис	300	1 122 000	3740
Пшеничная мука	400	1 444 000	3610
Клубничное варенье	50	130 000	2600
Фасоль (консервированная)	300	690 000	2300
Картофель	800	1 501 600	1877
Помидоры (консервированные)	300	237 000	790

Затем вы пытаетесь идти сверху вниз, выбирая еду с самым высоким соотношением калорий на единицу веса, и в итоге получаете набор из арахисового масла, риса, пшеничной муки и клубничного джема, что в сумме составляет 2 813 800 калорий.

Это намного лучше, чем первый результат. Понятно, что шаг был сделан в правильном направлении. Но если присмотреться, становится очевидным: добавление арахисового масла помешало нам взять фасоль, а это позволило бы еще больше увеличить общую ценность ящика. Хорошая новость в том, что вас по крайней мере не заставят соблюдать диету героя из упомянутого «Марсианина» — на сей раз картошка на Марс не отправится.

После еще нескольких часов мозгового штурма вы почти готовы сдаться, согласившись, что единственный способ решить эту задачу — проверить, можно ли получить лучшее решение, включив его или исключив каждый элемент из списка. Единственный известный вам способ — перечислить все возможные решения, отфильтровать те, которые превышают пороговое значение по весу, и выбрать лучшее из оставшихся. Это так называемый *алгоритм грубой силы (полный перебор)*, и из математики известно, что он очень дорогой.

Поскольку каждый предмет можно либо упаковать, либо оставить, то число возможных решений составляет $2^7 = 128$. Конечно, безрадостная перспектива: придется перебрать около сотни решений. Но через несколько часов, обессилев и уяснив, почему это называется алгоритмом грубой силы, вы почти заканчиваете решение этой задачи.

И тут появляются новости: кто-то позвонил из центра управления полетом и сообщил, что после жалоб некоторых будущих поселенцев в список были добавлены 25 новых продуктов питания, включая сахар, апельсины, сою и мармайт (не спрашивайте меня...).

После первой прикидки все приходят в ужас: теперь вам нужно проверить примерно 4 миллиарда различных комбинаций.

1.3.3. Спасительные алгоритмы

В этот момент становится ясно, что нужна компьютерная программа, чтобы обработать все числа и принять наилучшее решение.

Вы пишете ее сами в течение следующих нескольких часов, но она выполняется довольно долго. А тут выясняется, что, как бы вы ни хотели посадить всех колонистов на одну и ту же диету, это невозможно, потому что у некоторых из них есть аллергия: четверть состава не переносят глютен, и многие клянутся, что у них аллергия на мармайт. Так что придется запускать алгоритм еще на несколько прогонов, каждый раз учитывая индивидуальную аллергию. Что еще хуже, центр управления полетами рассматривает возможность добавления в список еще 30 пунктов, чтобы компенсировать то, что не получат люди с аллергией. Если это произойдет, в перечне продуктов окажется 62 возможных элемента и программе придется перебрать около миллиарда миллиардов возможных комбинаций. Программу запускают, проходит день, а она все еще работает, ни на чуть-чуть не приблизившись к завершению.

Вся команда готова сдаться и вернуться к картофельной диете, когда кто-то вспоминает, что в команде запуска есть человек, у которого на столе лежит учебник по алгоритмизации.

Вы вызываете его, и он сразу видит задачу такой, какая она есть на самом деле: это задача о рюкзаке 0-1. Есть и плохая новость: это NP-полная задача[1], а это означает, что ее трудно решить, поскольку не существует «быстрого» (то есть полиномиального по количеству элементов) известного алгоритма, который вычисляет оптимальное решение.

Однако есть и хорошие новости: существует псевдополиномиальное[2] решение, использующее *динамическое программирование*[3], а оно требует времени, пропорционального максимальной вместимости рюкзака. К счастью, вместимость ящика

---

[1] NP-полные задачи — это множество задач, для которых любое данное решение может быть быстро проверено (за полиномиальное время), но не существует известного эффективного способа найти правильное решение в первую очередь. NP-полные задачи по определению в настоящее время не могут быть решены за полиномиальное время на классической детерминированной машине (например, модель RAM, которую мы рассмотрим в следующей главе).

[2] Для псевдополиномиального алгоритма время работы в наихудшем случае зависит (полиномиально) от значения некоторых входных данных, а не только от их количества. Например, для задачи о рюкзаке 0-1 входными данными являются n элементов (с весом и стоимостью) и вместимость ранца C: полиномиальный алгоритм зависит только от числа n, в то время как псевдополиномиальный зависит также (или только) от значения C.

[3] Динамическое программирование — это стратегия решения сложных задач, обладающих определенными характеристиками, а именно рекурсивной структурой подзадач, при которой результат каждой подзадачи требуется использовать несколько раз при вычислении окончательного решения. Окончательное решение вычисляется путем разбиения задачи на набор более простых подзадач, решения каждой из этих подзадач только один раз и сохранения этих решений.

ограниченна, поэтому для решения необходимо количество шагов, равное количеству возможных значений заполненности емкости, умноженному на количество предметов. Таким образом, если наименьшим шагом изменения заполненности будет 1 кг, потребуется всего 1000 × 62 шага. Ого! Это намного лучше, чем 2^{62}! Фактически, как только вы выбираете правильный алгоритм, он находит лучшее решение за считаные секунды.

В какой-то момент вы готовы принять алгоритм как черный ящик и подключить его без лишних вопросов. Но ведь результат расчетов будет иметь решающее значение для вашего выживания... похоже, в этой ситуации стоило бы использовать более глубокие знания о том, как работает алгоритм.

В первоначальном примере оказывается, что наилучшая возможная комбинация — это рис, пшеничная мука и бобы, в общей сложности 3 256 000 калорий. Неплохой результат по сравнению с нашей первой попыткой, не так ли?

Возможно, вы и так уже успели найти наилучшую комбинацию, но если для начального примера, где всего семь элементов, поиск решения мог показаться слишком простым, попробуйте то же самое с сотнями различных продуктов в более близкой к реальному сценарию конфигурации и посмотрите, сколько лет придется искать лучшее решение вручную!

Казалось бы, получено удовлетворительное решение, и это лучшее, что можно найти с учетом ограничений.

1.3.4. Выходим за рамки (в буквальном смысле)

Но неожиданно в повествовании появляется настоящий эксперт по алгоритмам. Например, представьте, что выдающийся ученый посещает ваши объекты во время подготовки полета и его приглашают помочь с расчетом наилучшего маршрута для экономии топлива. Во время обеденного перерыва кто-то с гордостью рассказывает ему о том, как вы блестяще решили проблему с упаковкой товаров. Выслушав, он задает кажущийся наивным вопрос: почему, собственно, вы не можете изменить размер коробок?

Скорее всего, ответ будет либо «так было всегда», либо «товары поставляются уже упакованными, и изменение этого потребует времени и денег».

И тогда эксперт объяснит, что, если удалить ограничение на размер коробки, задача о рюкзаке 0-1, являющаяся NP-полной, станет задачей неограниченного рюкзака, для которой существует жадное решение с линейным временем[1], а оно обычно лучше, чем лучшее решение для версии 0-1.

В переводе на понятный человеку язык: можно превратить эту задачу в такую, которую легче решить и которая позволит наполнять ящики продуктами с максимально

[1] Линейное время, если список товаров уже отсортирован. В противном случае линейно-логарифмическое.

возможным количеством калорий. Теперь формулировка задачи становится такой: для данного набора предметов выберите любое подмножество предметов *или частей* предметов из него, чтобы их общий вес не превышал 1000 кг и, таким образом, чтобы общее количество калорий было максимальным.

И да, стоит потратить время на переупаковку всего, потому что мы получим серьезное улучшение.

В частности, если можно взять любую долю от первоначального веса для каждого продукта, то следует просто упаковывать продукты, начиная с продукта с наибольшим соотношением калорий на килограмм (в этом примере арахисовое масло), и, когда дойдет очередь до коробки, которая не поместится в оставшееся доступное пространство, переупаковать эту коробку, взяв ее часть, достаточную для заполнения этого пространства. Так что в конце концов придется переупаковывать даже не все товары, только один.

Кстати, здесь лучшим решением будет арахисовое масло, рис, пшеничная мука, клубничное варенье и 230 кг бобов, что в сумме составляет 3 342 800 калорий.

1.3.5. Счастливый конец

Итак, в рассказанной истории у будущих поселенцев на Марсе будет больше шансов на выживание, и они не впадут в депрессию из-за диеты, состоящей только из картофеля с арахисовым маслом и клубничного варенья.

С вычислительной точки зрения мы проделали путь от неправильных алгоритмов (жадные решения, которые сначала используют наибольшее значение или наибольшее отношение) к правильному, но невыполнимому алгоритму (решение методом грубой силы, перечисляющее все возможные комбинации) и, наконец, к умному решению, которое помогло организовать вычисления более эффективным образом.

Следующий шаг, столь же важный или даже более важный, заставил нас искать нестандартный подход, чтобы упростить проблему. В результате удалось снять некоторые ограничения и таким образом найти более простой алгоритм и лучшее решение. На самом деле это еще одно золотое правило: всегда тщательно изучайте свои требования, подвергайте их сомнению и по возможности старайтесь убрать ограничения, если это позволит получить по крайней мере не худшее решение за счет гораздо меньших усилий. В некоторых случаях так стоит поступать даже тогда, когда решение может незначительно ухудшиться. Конечно, при этом необходимо учитывать и другие аспекты (например, законы и безопасность на всевозможных уровнях), поэтому некоторые ограничения невозможно устранить.

В процессе описания алгоритмов, как было сказано в предыдущем разделе, я буду подробно описывать предлагаемое решение и предоставлять рекомендации по его реализации.

Для алгоритма динамического программирования решения задачи рюкзака 0-1 эти шаги будут опущены, потому что, во-первых, это алгоритм, а не структура данных

и, во-вторых, он уже подробно описан в литературе. Не говоря уже о том, что в данной главе работа с ним была использована просто для иллюстрации:

- того, как важно избегать неправильного выбора применяемых нами алгоритмов и структур данных;
- процесса, которому я буду следовать в следующих главах книги, вникая в задачу и рассуждая о путях ее решения.

РЕЗЮМЕ

- Алгоритмы нужно описывать с точки зрения их ввода, вывода и последовательности инструкций, которые будут обрабатывать ввод и производить ожидаемый вывод.
- Структура данных — это конкретная реализация абстрактного типа данных, состоящая из структуры для хранения данных как таковой и набора алгоритмов, которые ими управляют.
- Абстрагирование задачи означает формулирование четкой постановки задачи и только потом обсуждение ее решения.
- Эффективно упаковать рюкзак может быть непросто (особенно если вы планируете отправиться на Марс!), но с алгоритмами и правильной структурой данных нет (почти) ничего невозможного!