«Основные алгоритмы»

план-конспект лекций

Содержание

Лекция 1

Введение. Верхние и нижние оценки сложности алгоритмов

Литература: [ДПВ12; 05; КЛР02]

Содержание лекции

1. Язык Си как исполнители алгоритмов.
2. Сложность по времени и по памяти. Верхние и нижние оценки. Примеры:
   - Задача о поиске максимума последовательности: верхняя и нижняя оценки (последняя — через связность графа)
   - Проверка числа n на простоту перебором делителей до $\sqrt{n}$ — экспоненциальный алгоритм (сложность измеряется по длине входа.).
3. O, Ω, Θ обозначения — формальные определения.
   - $f(n) = \Theta(g(n))$ — отношение эквивалентности.
   - Если $P(n)$ — многочлен степени k, то $P(n) = \Theta(n^k)$.
   - Сумма чисел от 1 до n есть $\Theta(n^2)$.
   - $1^k + 2^k + \ldots + n^k = \Theta(n^{k+1})$.
   - Оценка суммы через интеграл (без доказательства).

1.1 Введение

Курс «Основные алгоритмы» посвящен решению алгоритмических задач. Под решением мы понимаем построение алгоритма, решающего задачу, доказательство его корректности и получение верхних и нижних оценок на время его работы и используемую память — оценку сложности. Мы не будем уделять особое внимание формализации понятий «алгоритмическая задача» и даже «алгоритм» — дело это непростое, и этому посвящена значительная часть курса «Теория формальных систем и алгоритмов» в следующем семестре. Тем не менее, каждый из вас интуитивно понимает смысл этих понятий и уже работал с ними на курсе информатики.

Давайте проиллюстрируем описанные выше этапы решения задачи на примере простой и хорошо известной задачи — задачи о поиске максимума последовательности.

Пример 1. *На вход задачи подаётся последовательность целых чисел $x_1, \ldots, x_n$, ввод которой оканчивается маркером конца строки. Необходимо найти наибольший элемент этой последовательности.*

Сначала нам нужно убедиться, что задача алгоритмически разрешима — увы, не все задачи имеют решение. Для этих целей достаточно построить даже не самый оптимальный алгоритм — приведём таковой.
Алгоритм. Запишем все элементы последовательности в массив. Будем сравнивать все возможные пары x_i и x_j и хранить результат сравнения в матрице A (двумерном массиве): $a_{i,j} = 1$, если $x_i \geqslant x_j$ и $a_{i,j} = 0$ иначе. Найдём в получившейся матрице строку, состоящую из одних единиц — пусть её номер i. Выведем элемент x_i в качестве ответа.

После того как алгоритм описан, нужно доказать его корректность.
Корректность. Поскольку элементов последовательности конечное число, значит найдётся элемент x_k, который не меньше всех остальных — по построению матрицы A, все элементы k-ой строки будут тогда равны единице. Если алгоритм вывел элемент x_k, то он сработал корректно, потому что x_k — максимум по определению; если же алгоритм вывел другой элемент — x_i, то $x_i \geqslant x_k$, поскольку $a_{i,k} = 1$, а значит x_i также является максимумом.

Программисты часто считают, что если алгоритм описан, то это описание и является доказательством его корректности, однако как легко видеть, длина предыдущего абзаца даже немного больше, чем длина

описания самого алгоритма. Бесспорно, корректность этого алгоритма очевидна, но мы здесь специально обращаем внимание на то, что доказательство корректности является важным шагом в решении задачи, которым нельзя пренебрегать.

После того как корректность алгоритма доказано, возникает вопрос о том, насколько этот алгоритм эффективен. В качестве основных показателей эффективности чаще всего исследуют время работы алгоритма (количество операций, которые он совершает) и размер потребляемой памяти. Для того, чтобы формально оценить эти показатели, требуется зафиксировать формальную модель вычислений — исполнителя алгоритма. Такими моделями как правила являются машина Тьюринга и разные вариации RAM-модели. Однако изучение этих моделей мы также оставим для второго курса. В качестве исполнителя алгоритма, мы будем использовать язык Си.

Чтобы оценить временную сложность алгоритма, достаточно записать его код на Си и посчитать количество операций, выполняемых программой. Сложность оценивается от длины входа. В данном примере, чтобы оценить сложность, нам нужно наложить дополнительное условие на задачу: нужно договориться, считаем ли мы арифметические операции константными или зависящими от длины записи числа. На практике выбор условия зависит от задачи. Будем считать, что в нашем случае все числа x_i укладываются в диапазон `Integer` — выбираем первое условие.

Оценка сложности. На вход программе из нашего примера подаётся n чисел — длина входа порядка n. Программа сравнивает каждый элемент с каждым и заполняет матрицу — на это уходит порядка n^2 операций, и потом ищет единичную строку — это тоже порядка n^2 операций за один проход по матрице. Выполнение каждой из упомянутых операций стоит некоторую константу (как и вспомогательных операций в процессе исполнения алгоритма), поэтому не говорят, что программа работает за n^2, а говорят что время работы программы *порядка* n^2 или, что тоже самое — время работы алгоритма $\Theta(n^2)$. Далее мы приведём формальное определение этого обозначения.

Со сложностью по памяти дело обстоит также: программа хранит массив из n элементов и матрицу из n^2 элементов, а значит использует $\Theta(n^2)$ битов памяти. Обратим внимание, что тут мы считаем, что целые числа укладываются в диапазон `Integer`, и потому операции считывания, присваивания и сравнения стоят константу. Фактически, этот выбор является нашим выбором модели вычислений, в которой и происходит оценка

сложности работы алгоритма. Если бы мы считали, что числа могут быть очень большими, то ни в один числовой тип языка Си такое бы число не поместилось, а значит нам бы пришлось считать сложность операций присваивания и сравнения не константной, а зависящей от длины записи чисел и тогда оценка времени работы алгоритма изменилась бы.

1.1.1 Верхние и нижние оценки

Мы оценили сложность по времени и по памяти только одного алгоритма, решающего нашу задачу. Существование алгоритма гарантирует разрешимость задачи, а сложность алгоритма даёт верхнюю оценку на сложность задачи, под которой мы понимаем сложность самого лучшего алгоритма, решающего задачу. Заметим, что может так получиться, что один алгоритм может быть самым быстрым, в то время как минимальную память может использовать другой, не самый быстрый, алгоритм. Поэтому мы всегда разделяем сложность задачи по времени и по памяти, и наличие у задачи двух оценок на сложность по времени и сложность по памяти не влечёт существование алгоритма, для которого обе эти оценки выполняются.

Подобно символу Θ, который обозначает порядок роста, для обозначения верхних оценок используют символ O: мы показали, что сложность задачи по времени и по памяти есть $O(n^2)$. Оценки только верхние, потому что могут быть алгоритмы, которые решают задачу лучше — и такой алгоритм есть для нашего примера.

Алгоритм. Считаем первые два элемента и сравним их, запомнив максимальный из них, затем считаем третий элемент и сравним его с максимумом из первых двух и так далее:

$$m = \max(x_1, x_2); \quad m = \max(m, x_3); \quad \dots \quad m = \max(m, x_n).$$

В результате исполнения такого алгоритма (написать код на Си мы оставляем читателю) в переменной m очевидно окажется максимум последовательности. Такой алгоритм работает за время $\Theta(n)$, поскольку выполняет $n-1$ операцию сравнения в процессе вычисления максимумов, и требует константу памяти, то есть $\Theta(1)$ памяти. Итак, мы получили верхние оценки гораздо лучше: оценку по времени $O(n)$ и оценку по памяти $O(1)$!

Насколько эти оценки хорошие? Понятно, что лучшая оценка по памяти невозможна: алгоритм должен хранить в памяти хотя бы макси-

мальный элемент, чтобы его вывести. Но что насчёт оценки по времени? Здравый смысл подсказывает, что лучше эту задачу не решить, но как это доказать? Почему не найдётся кто-то очень умный, который придумает алгоритм лучше?

Для того, чтобы это доказать нам нужна формализация как алгоритма, так и задачи. От алгоритма нам потребуется свойство *детерминированности*: если в процессе вычислений на двух разных входах алгоритм выполнил одну и ту же последовательность действий первые n шагов и хранит в памяти одинаковые значения, то следующий шаг для обоих входов будет одинаковым. От задачи мы потребуем следующее: теперь алгоритму не будут сообщать значения самих чисел, но алгоритм может попросить сравнить два числа и получить в качестве ответа на вопрос $x_i \overset{?}{\geqslant} x_j$ один бит. Теперь задачу можно сформулировать так: есть n монет разного веса и чашечные весы — необходимо найти самую тяжёлую монету, совершив как можно меньше взвешиваний.

Утверждение 1. *Для поиска самой тяжёлой монеты необходимо совершить $n-1$ взвешивание.*

Доказательство. Возьмём произвольный набор из n монет с весами $x_1, \ldots, x_n$ ($x_i > 0$) и отдадим их на вход алгоритму, который делает меньше, чем $n-1$ взвешивание. Запомним какие монеты сравнивались между собой и построим граф, вершины которого — монеты, а ребро есть только между монетами, которые сравнивались между собой.

Поскольку всего было меньше, чем $n-1$ взвешивание, то в графе меньше $n-1$ ребра, а значит граф несвязен. Максимум x_k лежит в некоторой компоненте связности — назовём её первой. В качестве второй компоненты связности возьмём любую другую и пусть в ней максимум $x_m \leqslant x_k$.

Увеличим вес каждой монеты во второй компоненте связности на x_k. Это не поменяет результатов сравнений, но максимумом теперь станет монета под номером m с весом $x_m + x_k$. В силу детерминированности, алгоритм на изменённом входе должен вернуть монету x_k как самую тяжёлую, но значит алгоритм решает задачу неверно. □

Замечание 1. *Детерминированности алгоритма чаще всего достаточно, чтобы доказать, что алгоритм не может работать сублинейное*

время[1]*. Если алгоритм работает за сублинейное время в худшем случае, то он не считывает некоторую часть входа, а чтобы решить задачу чаще всего нужно знать весь вход. Так, если алгоритм не узнал значение одного из x_i, то именно этот элемент может оказаться максимальным.*

1.2 Сложность алгоритмов

Определим формально временную сложность работы алгоритмов (сложность по памяти определяется аналогично). Временная сложность алгоритма — это количество операций, которые алгоритм выполняет в худшем случае на входе длины n, таким образом, это функция $f : \mathbb{N}_+ \to \mathbb{N}_+$ (под $\mathbb{N}_+$ мы понимаем положительные целые числа). Поскольку нас интересуют не численные значения этой функции, а порядок роста, то дабы упростить жизнь в случае функций вида $f(n) = n\lceil \log n \rceil$ (здесь и далее $\lceil x \rceil$ — округленик числа x вверх до целого), мы будем рассматривать функции $f : \mathbb{N}_+ \to \mathbb{R}_+$, где $\mathbb{R}_+$ — это положительные вещественные числа.

Пример 2. *На вход подаётся число N, необходимо проверить, является ли оно простым.*

Задачу легко решить, деля с остатком N на каждое из чисел $a \leqslant \sqrt{N}$: если один из остатков ноль, то число составное, иначе — простое. Этот алгоритм имеет сложность $O(\sqrt{N})$ (мы считаем, что арифметические операции стоят константу), но N в данном примере не длина входа! Чтобы записать число N потребуется $n = \log_2 N$ бит, поэтому сложность алгоритма по длине входа есть $O(2^{n/2})$ — это экспоненциальный алгоритм! Поэтому простые числа ищут нетривиальным вероятностным алгоритмом, а полиномиальный детерминированный алгоритм является значимым результатом в Computer Science.

1.2.1 O-Ω-Θ-обозначения

Выше мы уже использовали обозначения Θ и O для оценки сложности алгоритмов. Дадим теперь формальное определение этим обозначениям.

[1] Алгоритм работает сублинейное время, если совершает меньше, чем Cn операций — например, порядка $\lceil \log n \rceil$.

Определение 1. Говорят, что $f(n) = O(g(n))$, если $f(n) \leqslant Cg(n)$ начиная с некоторого N; здесь C — положительная константа. Формально

$$\exists C > 0, N \in \mathbb{N}_+ \; \forall n \geqslant N : f(n) \leqslant Cg(n).$$

Таким образом, функция g является верхней оценкой для функции f. Заметим, что тогда f — это нижняя оценка для функции g. Для описания нижних оценок используют обозначение Ω. Формально,

$$g(n) = \Omega(f(n)), \text{ если } f(n) = O(g(n)).$$

В случае, когда функция g является как верхней, так и нижней оценкой для функции f, используют обозначение Θ:

$$f(n) = \Theta(g(n)), \text{ если } f(n) = O(g(n)) \text{ и } f(n) = \Omega(g(n)).$$

Упражнение 1. Покажите, что если $f(n) = \Theta(g(n))$, то $g(n) = \Theta(f(n))$.

1.2.2 Примеры и свойства

Пример 3. *Порядок роста многочлена степени k с неотрицательными коэффициентами — n^k:*

$$P_k(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 = \Theta(n^k).$$

Действительно, пусть a — максимальное число среди коэффициентов: $a = \max\limits_i a_i$, тогда при $n > 1$, $P_k(n) \leqslant kan^k$:

$$a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 \leqslant an^k + an^{k-1} + \ldots + an + a \leqslant kan^k,$$

а значит $P_k(n) = O(n^k)$. А также, при достаточно больших n (например, $n > ka$), $P_k(n) \geqslant \frac{1}{a} n^k$ и $P_k(n) = \Omega(n^k)$.

Замечание 2. *Естественно хочется заявить, что порядок роста произвольного многочлена P_k степени k с положительным коэффициентом при старшем члене есть n^k. Формально, этого нельзя сделать в тех случаях, когда $P_k(n) < 0$ для некоторого n — мы не рассматриваем функции такого вида, согласно определению асимптотических обозначений. Однако, незаконная промежуточная выкладка, подобная $P_k(n) = \Theta(n^k)$, часто удобна при оценке сложности алгоритмов. Чтобы узаконить её, мы будем считать, что нам годятся не только функции из $\mathbb{N}_+$ в $\mathbb{R}_+$, но и функции из $\mathbb{N}_+$ в $\mathbb{R}$, которые начиная с некоторого n принимают только положительные значения.*

Пример 4. *Сумма чисел от* 1 *до* n *есть* $\Theta(n^2)$.

Действительно, согласно формуле арифметической прогрессии, данная сумма есть $\dfrac{n(n+1)}{2}$.

Обобщим этот пример.

Пример 5. $1^k + 2^k + \ldots + n^k = \Theta(n^{k+1})$

Оценим сверху каждый член суммы как n^k, получаем

$$1^k + 2^k + \ldots + n^k \leqslant n^k + n^k + \ldots + n^k = n \times n^k = O(n^{k+1}).$$

Отбросим из суммы первую половину членов и заметим, что каждый оставшийся член не меньше $(n/2)^k$:

$$1^k + 2^k + \ldots + n^k \geqslant \underbrace{\left(\frac{n}{2}\right)^k + \left(\frac{n}{2}\right)^k + \ldots + \left(\frac{n}{2}\right)^k}_{\frac{n}{2}} = \frac{n}{2} \times \left(\frac{n}{2}\right)^k = \Omega(n^{k+1}).$$

Приведённые примеры требуют некоторой изобретательности для доказательства верхних и нижних оценок. Вместо них можно пользоваться фактом из математического анализа:

Утверждение 2. *Пусть* $f : [0, +\infty) \to \mathbb{R}_+$ *— непрерывная функция и* $F(x)$ *— её первообразная. Тогда*

$$\sum_{k=0}^{n} f(k) = \Theta(F(n)).$$

Так, $\sum\limits_{k=1}^{n} \dfrac{1}{n^2} = \Theta(\dfrac{1}{n})$.

1.2.3 Эффективные алгоритмы

Определение 2. Пусть $f(n)$ — временная сложность работы алгоритма (максимальное число шагов на входе длины n). Алгоритм называется *полиномиальным*, если $f(n) = O(n^k)$ для некоторого k.

Мы будем считать алгоритм эффективным, если он полиномиален. Так, приведённый в примере 2 алгоритм проверки числа на простоту не эффективен, поскольку экспоненциален.

1.2.4 Θ-эквивалентность и сравнение функций

Оставляем читателю проверить, что отношение « $f(n) = \Theta(g(n))$ » на множестве функций $\mathbb{N}_+ \to \mathbb{R}_+$ является отношением эквивалентности; будем называть его Θ-эквивалентность. Если $f(n) = \Theta(g(n))$, то часто можно в формуле заменить $f(n)$ на $g(n)$, не изменив при этом порядок роста функции, заданной формуле. Однако такая замена справедлива не для всех формул.

Упражнение 2. Проверьте, что $2^n \neq \Theta(2^{2n})$, хотя $n = \Theta(2n)$.

Упражнение 3. Проверьте, что если $f(n) = \Theta(g(n))$, то $f(n) + h(n) = \Theta(g(n)+h(n))$ и $f(n)\times h(n) = \Theta(g(n)\times h(n))$ для произвольной функции h.

Знак равенства в обозначении $f(n) = \Theta(g(n))$ возник при упрощении нотации. С формальной точки зрения, $O(g(n)), \Omega(g(n))$ и $\Theta(g(n))$ — множества функций. Правильно было бы писать $f(n) \in \Theta(g(n))$.

Θ-эквивалентность функций означает их асимптотическое равенство, если $f(n) = O(g(n))$, то g асимптотически не меньше f, а если $f(n) = \Omega(g(n))$, то f асимптотически не больше g.

Множество функций разбивается на классы Θ-эквивалентности. Если f и g лежат в разных классах и $f(n) = O(g(n))$, то отсюда следует что все функции эквивалентные f асимптотически меньше всех функций, эквивалентных g. Таким образом определено отношение порядка на классах эквивалентности. Однако отсюда не следует, что если функции f и g принадлежат разным классам, то одна из них асимптотически меньше другой. Например, функции n^2 и $n^{2+\sin n}$ несравнимы. Это отношение порядка отличается от порядка на числах: оно не линейно, то есть, не все элементы сравнимы.

Лекция 2

Жадные алгоритмы

Литература: [Шен04; КФ12]

1. Пример: задача о поиске треугольника максимальной площади, сторона которого лежит на оси Ox.
2. Задача о рюкзаке.
   - Жадный алгоритм для непрерывной задачи о рюкзаке [КФ12].
   - Жадный алгоритм для 2-приближённого алгоритма для дискретной задачи о рюкзаке [КФ12].
   - Коротко о **P** и **NP**, почему безнадёжно искать точное решение некоторых задач.
3. Жадные алгоритмы и индуктивные функции [Шен04].
   - функции максимума и суммы — индуктивные
   - индуктивное расширение на примере поиска максимума произведения
4. Онлайн-алгоритмы
5. Пример нетривиального жадного алгоритма для следующей задачи. На вход подаётся последовательность чисел $a_1, a_2, \ldots, a_n$, при этом все числа, за исключением одного, входят в последовательность ровно два раза. Необходимо найти число, которое встречается в последовательности один раз.

2.1 Примеры

Программисты относят к жадным алгоритмам алгоритмы, подчиняющиеся следующему принципу. На каждом шаге алгоритм находит локально-оптимальное решение задачи (то есть, лучшее на данный момент), и хранит в памяти только его (возможно с небольшим объёмом вспомогательных данных). Алгоритм поиска максимума в последовательности с предыдущей лекции является классическим примером жадного алгоритма: на каждом шаге алгоритм помнит только максимум считанного начального отрезка последовательности и обновляет его по мере появления новых элементов последовательности.

Начнём с примера задачи, для которой жадный алгоритм устроен чуть более сложно.

Пример 6. *На вход подаются координаты точек плоскости* $(x_0, y_0), \ldots, (x_n, y_n)$*; вход заканчивается маркером конца строки. Нужно найти треугольник максимальной площади, одна из сторон которого лежит на оси* Ox *(вершины треугольника присутствуют в последовательности).*

Подобно задачам на построение в геометрии, начнём с анализа задачи. Если треугольник уже найден, то его сторона, лежащая на оси Ox, самая длинная среди сторон на Ox по всем треугольникам. Действительно, иначе можно заменить сторону треугольника на более длинную, не поменяв высоту и увеличить тем самым площадь. Рассуждая аналогично заключаем, что вершина (x, y) треугольника, не лежащая на Ox имеет максимальное значение $|y|$ — иначе можно заменить вершину и увеличить высоту.

Проведя анализ заключаем, что для решения задачи достаточно найти самую длинную сторону на оси Ox и точку, не лежащую на оси Ox с максимальным $|y|$. Чтобы найти сторону достаточно найти самую левую и самую правую точки: то есть точки $(x_{\min}, 0)$ и $(x_{\max}, 0)$.

Для решения задачи можно воспользоваться «принципом чайника» — так программисты называют использование уже готового решения. Согласно анекдоту, чтобы программисту вскипятить чайник, ему нужно взять чайник, налить в него воду и нажать на кнопку. Но если программисту дать чайник с водой, то вместо того, чтобы нажать на кнопку, он выльет воду и сведёт тем самым задачу к предыдущей.

Итак, чтобы решить задачу, достаточно найти два максимума (среди точек вида $(x, 0)$ и вида $(x, y), y \neq 0$ по y) и один минимум — ясно, что

минимум можно искать также как и максимум. Для этих целей достаточно считать все координаты в массив и выполнить три линейных алгоритма. Таким образом, мы получили линейный по времени алгоритм для решения задачи.

Этот алгоритм не считается жадным, потому что решение он находит только в самом конце и хранит много лишней информации в памяти — $\Theta(n)$ битов. Однако этот алгоритм легко преобразовать в жадный алгоритм, линейный по времени и использующий константную память.

Вместо того, чтобы искать максимум и минимум последовательно, будем искать их параллельно. Считав координаты (x, y) очередной точки, определим, лежит ли она на оси Ox, и если да, то меньше ли x текущего минимума $x_{\min}$ или больше ли x текущего максимума $x_{\max}$; если же точка не лежит на оси Ox (т.е. $y \neq 0$), то проверим больше ли $|y|$ текущего $|y|_{\max}$.

Заметим, что часть доказательства корректности была приведена выше при анализе задачи, а оставшаяся часть состоит в повторении доказательство корректности для алгоритма поиска максимума в последовательности.

Пока мы используем неформальное понятие жадного алгоритма, мы приступим к его формализации после следующего классического примера.

2.2 Задача о рюкзаке

Понятие жадности хорошо проиллюстрировать на примере задачи о рюкзаке.

Пример 7. *Вор забирается в ювелирный магазин с мешком, вместимостью M килограмм. Каждое ювелирное изделие стоит c_i\$ и имеет вес m_i. Задача вора — собрать мешок с максимальным весом.*

У этой задачи существует два разных вида — в первом случае вор может либо взять каждое изделие или нет, а во втором, ценность ювелирной работы не очень важна, а важен только драгоценный метал, и вор может отпилить кусок изделия, не влезающий в мешок. Первая вариация известна как задача о *дискретном рюкзаке*, а вторая — как задача о *непрерывном рюкзаке*.

На первый взгляд, задачи эти отличаются незначительно. Но построив полиномиальный алгоритм для первой, вы получите \$1 000 000 от инсти-

тута Клэя за решение задачи тысячелетия: вы докажете, что $\mathbf{P} = \mathbf{NP}$, решив **NP**-полную задачу (о рюкзаке). **P** и **NP** — это два класса задач: первый класс состоит из задач, разрешимых полиномиальным алгоритмом, а второй — из задач, проверяемых за полиномиальное время. Если кто-то очень умный сообщит вору какие вещи ему брать, чтобы набить рюкзак лучшим способом, то вор сможет это проверить. Изучение класса **NP** не входит в этот курс, однако мы обращаем внимание, что задачи из этого класса достаточно естественны и многие из них никто не умеет решать за полиномиальное время.

Жадный алгоритм, решающий непрерывную версию задачи о рюкзаке описан в [КФ12]. Если кратко, то вору нужно отсортировать драгоценности по их удельной стоимости $\rho_i = \frac{c_i}{m_i}$ и класть в мешок в первую очередь те драгоценности, удельная стоимость которых больше ещё не взятых. Если место в рюкзаке ещё осталось, а самый ценный на данном шаге предмет в него не влезает, то его нужно распилить и поместить в мешок влезающий в него кусок. Оставляем доказательство корректности этого алгоритма читателю в качестве упражнения и рекомендуем обратиться к указанной книге для самопроверки.

Также в [КФ12] приведён жадный алгоритм 2-приближённого решения дискретной задачи о рюкзаке — это значит, что вор жадным образом может набить свой мешок драгоценностями, стоимость которых не меньше, чем половина от максимальной стоимости. Так что факт **NP**-полноты задачи ещё не мешает воровать достаточно эффективно, а дальше мы изучим приближённый алгоритм решения этой задачи (который позволит воровать ещё эффективнее, но пожалуйста используйте его в мирных целях).

Неформально, алгоритм называется *жадным*, если для него выполняется принцип «дают бери, а бьют — беги» как в примере с вором. Чуть более формально, жадный алгоритм на каждом шаге ищет локально-оптимальное решение и в итоге приходит к глобальному оптимуму.

Класс задач, допускающих жадное решение описывается с помощью математического понятия «матроид». Это понятие сложно для начального курса, поэтому мы рекомендуем познакомиться с ним только после изучения нашего курса, а пока рассмотрим один из способов формализации «жадности», не претендующий на полноту.

2.3 Индуктивные функции

Рассмотрим функции, которые определены на конечных последовательностях произвольной длины $(x_1, \dots x_n)$ с элементами из множества A, и принимают значение в множестве B. Функция f данного вида называется *индуктивной*, если существует функция $F : B \times A \to B$, такая что

$$f(x_1, \dots, x_n, x_{n+1}) = F(f(x_1, \dots, x_n), x_{n+1}).$$

Пример 8. *Функции* $\max$ *и* $\mathrm{sum}(x_1, \dots, x_n) = \sum_{i=1}^{n} x_i$ *являются индуктивными:*

- $\max(x_1, \dots, x_n, x_{n+1}) = \max(\max(x_1, \dots, x_n), x_{n+1})$
- $\mathrm{sum}(x_1, \dots, x_n, x_{n+1}) = \mathrm{sum}(\mathrm{sum}(x_1, \dots, x_n), x_{n+1})$

Для каждой функции из примера $f = F$, но в общем случае это необязательно.

Жадный алгоритм можно получить для задачи, которая состоит в вычислении индуктивной функцию f, путём нахождения функции F. Если F известна, то достаточно в цикле считывать следующий элемент последовательности и вычислять $y_i = F(y_{i-1}, x_i)$, где значение $y_{i-1} = f(x_1, \dots, x_i)$ было вычислено на предыдущем шаге цикла.

Однако, часто бывает, что требуемая функция не является индуктивной, но если её чуть-чуть поправить, то она будет уже индуктивной.

Пример 9. *Функция* $f(x_1, \dots x_n) = \max_{i \neq j} x_i \times x_j$*, определённая на положительный целых числах, не индуктивная. Однако её можно превратить в индуктивную, если хранить в памяти пару* (m_1, m_2) *из первого и второго максимума последовательности.*

Этот приём называют индуктивным расширением. Формально, индуктивная функция g называется *индуктивным расширением* функции f, если существует такая функция $t : B \to B$, что

$$t(g(x_1, \dots, x_n)) = f(x_1, \dots, x_n).$$

Для примера с максимальным произведением можно, взять в качестве g функцию, возвращающую пару (m_1, m_2), тогда $t(m_1, m_2) = m_1 \times m_2$.

Одним из подходов решения алгоритмических задач является выбор математического инварианта, который поддерживается в ходе исполнения программы. В случае жадных алгоритмов, такой инвариант часто получается найти, сформулировав задачу в терминах индуктивных функций (возможно с расширением). Этот подход отражён в книге [Шен04], в которой индуктивные функции освящены более подробно.

2.4 Онлайн-алгоритмы

Индуктивные функции естественным образом применяются для решения задач специального вида:

Вход: последовательность $x_1, x_2, \ldots, x_n$ (n заранее не задано);
Выход: $f(x_1, x_2, \ldots, x_n)$.

Функция f является параметром и фактически определяет задачу. Тип элементов последовательности зависит от задачи. Обратим внимание, что f определена для всех конечных последовательностях, или что то же самое на любом массиве.

Решение задачи состоит в вычислении $f(x_1, x_2, \ldots, x_n)$, но помимо итогового результата требуется вычислять значение $f(x_1, x_2, \ldots, x_k)$ после обработки каждого элемента последовательности x_k на входе.

Такие задачи часто встречаются в реальной жизни. Например, при реализации электронной очереди в банке, заранее неизвестно кто из клиентов придёт по какому вопросу, и нужно распределять всех клиентов по сотрудникам по мере прихода. Другой пример, при работе агентства недвижимости, необходимо продавать квартиру первому покупателю, пожелавшему её приобрести. Агентству было бы выгоднее подождать всех покупателей за год, узнать их предпочтения и только после этого продать максимальное число квартир по лучшим (для агентства) ценам. Но покупателей приходится обслуживать в порядке их прихода.

Алгоритмы для таких задач называют *онлайн-алгоритмами*. То есть, онлайн-алгоритм вычисляет значение $f(x_1, x_2, \ldots, x_k)$ после считывания каждого элемента x_k.

Заметим, что если алгоритм на каждом шаге вычисляет индукитвную функцию, то это онлайн алгоритм. Если же он вычисляет индуктивное расширение g, то он не обязательно онлайн — для того, чтобы стать

онлайн-алгоритмом, ему нужно ещё на каждом шаге вычислять и значение $t(g(x_1, \ldots, x_i))$.

Онлайн-алгоритмы возникают не только при изучении жадных задач. Например, на практике бывают нужны онлайн-алгоритмы сортировки. Такие алгоритмы на каждом шаге хранят в памяти отсортированный начальный отрезок последовательности. Представьте, что в библиотеку за неделю завозят несколько партий книг. Конечно, чтобы все их расставить на полки лучше знать заранее какие книги и сколько привезут (чтобы оставить нужное место на полках), но если этого не знать, то книги всё равно нужно расставить на полки в отсортированном порядке, дабы обслуживать читателей. Возможно, этот пример станет более убедительным, если мы заменим библиотеку на базу данных.

Офлайн-алгоритмами называют алгоритмы, которые решают задачи обработав весь вход. Ясно, что производительность офлайн-алгоритмов не хуже, чем онлайн (первые могут работать как онлайн). Для некоторых практических задач нужны именно онлайн алгоритмы, поэтому в computer science исследуют отношение производительности онлайн-алгоритмам к офлайн для этих задач.

2.5 Связь между ключевыми понятиями лекции

У нас были формальные определения индуктивной функции (с расширением) и онлайн-алгоритмов. Мы не привели формального определения жадного алгоритма в силу его трудности для вводного курса. Заметим, что эти понятия находятся в общем положении: жадные алгоритмы используются для задач оптимизации (таких как поиск максимума какой-то функции), и эти задачи могут как иметь, так и не иметь формулировку, требуемую для онлайн-алгоритмов. Онлайн-алгоритмы, в свою очередь, могут применяться не для задач оптимизации: например, для задачи сортировки.

Индуктивные функции удобно использовать для поиска инварианта, пересчёт которого по мере обработки данных приводит к решению задачи. С их помощью легко найти жадный алгоритм для элементарных задач. В то же время, в формулировке задач для онлайн-алгоритмов фактически используется индуктивная функция.

Мы будем использовать понятие жадного алгоритма неформально. У студентов возникают сомнения: является ли алгоритм в их решении жадным? Если при решении задачи была получена индуктивная функция (возможно с расширением), то мы считаем (для простоты), что в решении получился жадный алгоритм. Хотя жадными (с точки зрения математики) бывают не только такие алгоритмы, а с точки зрения программистов, жадность — понятие растяжимое.

2.6 Нетривиальный «жадный» алгоритм

Слово «жадный» взято в кавычки, поскольку в следующем примере решается не задача оптимизации. Для её решения («жадно») поддерживается нетривиальный инвариант.

Задача 1. На вход подаётся последовательность чисел $x_1, x_2, \ldots, x_n$, при этом все числа, за исключением одного, входят в последовательность ровно два раза. Необходимо найти число, которое встречается в последовательности один раз.

Решение. Пусть b_i — двоичная запись числа x_i, а $b_i[k]$ — её k-ый бит. Будем проводить побитовый XOR[1] двоичных записей. Сначала пусть $c = b_1$, на i-ом шаге $c = c \oplus b_i$, то есть $c[k] = c[k] \oplus b_i[k]$.

Таким образом, $c = b_1 \oplus b_2 \oplus \ldots \oplus b_n$. Заметим что, операция XOR коммутативна и $b_i \oplus b_j = 00\ldots0$, если $x_i = x_j$. Значит в c после исполнения останется двоичная запись элемента x_k, который встречается в последовательности ровно один раз.

Приведённый алгоритм является «жадным», поскольку на каждом шаге поддерживает и пересчитывает инвариант, который на последнем шаге оказывается правильным ответом. Сама задача не сформулирована явно с помощью индуктивной функции, однако в качестве таковой можно было бы взять функцию $f(b_1, \ldots, b_n) = b_1 \oplus b_2 \oplus \cdots \oplus b_n$. Промежуточные шаги вычисления этой функции не соотносятся с условием задачи, однако из-за ограничений на вход, на последнем шаге функция принимает искомое значение.

[1]Битовую операцию XOR (исключающее или) обозначают $a \oplus b$. Если значения a и b не совпадают, то $a \oplus b = 1$, а если совпадают, то $a \oplus b = 0$.