«Основные алгоритмы»

план-конспект лекций

Содержание

Лекция 1

Введение. Жадные алгоритмы

Литература: [ДПВ12; КЛРШ05; КЛР02; Шен04; КФ12]

Содержание лекции

Язык Си как исполнители алгоритмов. Сложность по времени и по памяти. O, Ω, Θ обозначения. Жадные алгоритмы и индуктивные функции [Шен04].

Примеры алгоритмов:

- Поиск максимума, верхняя и нижняя оценки (связность графа).
- Проверка числа n на простоту перебором делителей до $\sqrt{n}$ — экспоненциальный алгоритм (сложность измеряется по длине входа.).
- Жадный алгоритм для непрерывной задачи о рюкзаке [КФ12].
- Пример нетривиальной индуктивной функции (см. задачу).

Задача 1. 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.

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 решением 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 `int` — 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. Подобно символу Θ, который обозначает порядок роста, для обозначения верхних оценок используют символ O: мы показали, что сложность алгоритма 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 $\Theta(n)$, поскольку выполняет $n-1$ операцию сравнения в процессе вычисления максимумов, и требует $\Theta(1)$. 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 невозможно: алгоритм должен хранить в памяти хотя бы максимальный 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 сравнений.

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

Доказательство. Возьмём произвольный набор различных монет $x_1, \ldots, x_n$ 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 x_m. 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] Алгоритм работает сублинейное время, если совершает меньше, чем Cn операций — например, порядка $\lceil \log n \rceil$.

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 $f : \mathbb{N} \to \mathbb{N}$. Поскольку нас интересуют не численные значения этой функции, а порядок роста, то дабы упростить жизнь в случае функций вида $f(n) = n\lceil \log n \rceil$, 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.

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 $g(n) = \Theta(f(n))$.

Пример 3. *Порядок роста многочлена степени k — 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 модулей 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 $P_k(n) = \Omega(n^k)$.

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.

1.3 Жадные алгоритмы

Понятие жадности хорошо проиллюстрировать на примере задачи о рюкзаке.

Пример 4. *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 института 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 изу-

чения нашего курса, а пока рассмотрим один из способов формализации «жадности», не претендующий на полноту.

1.3.1 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 5. *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 6. *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.

1.3.2 Онлайн-алгоритмы

Пусть f, как и в предыдущем разделе, — функция, которая определена на последовательностях и задача состоит в том, чтобы вычислить функцию f на входной последовательности $(x_1, \dots x_n)$.

Алгоритм для задачи такого вида называется *онлайн-алгоритмом*, если после считывания каждого элемента x_i он вычисляет $f(x_1, \dots, x_i)$.

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 $t(g(x_1, \dots, x_i))$.

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.

1.3.3 Нетривиальный жадный алгоритм

Жадный алгоритм может оказаться нетривиальным.

Задача 2. На вход подаётся последовательность чисел $x_1, x_2, \dots, x_n$, 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[2] 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.

[2]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 $a \oplus b = 0$.

Лекция 2

Рекурсия и итерация

Содержание лекции

Литература: [ДПВ12; Шен04]

Переход от алгоритмов, заданных рекурсивно, к алгоритмам, заданным итеративно, с использованием стека на примере алгоритма Евклида.

- Алгоритм Евклида.
- Доказательство нижних оценок на время работы алгоритма Евклида через числа Фибоначчи.
- Общая схема перехода от рекурсии к итерации.
- Расширенный алгоритм Евклида

2.1 От рекурсии к итерации

Алгоритм Евклида поиска наибольшего общего делителя — классический пример рекурсивного алгоритма, который легко реализовать итеративно.

```
1 Function gcd(x, y) :
  |  Вход  : (x, y) — неотрицательные целые числа, x > 0
  |  Выход : НОД(x, y)
2 |  if y == 0 then
3 |  |  return x
4 |  end
5 |  return gcd(y, x mod y)
6 end
```

Рис. 2.1: Алгоритм Евклида.

Перед этим, давайте обсудим корректность алгоритма и общий подход перехода от итерации к рекурсии. Пусть $x = ky + b$, где $b = x \mod y$ — остаток от деления x на y. Если d "наибольший общий делитель x и y, то $b = x - ky = x' \times d - ky' \times d$ также делится на d. Если бы вдруг оказалось, что НОД(y, b)= $d' > d$, то получаем, что $x = ky' \times d' + b' \times d'$, а значит, что x также делится на d', что невозможно, т.к. НОД(x, y)= d. Значит, мы доказали, что НОД(y, b)=НОД(x, y). Осталось доказать, что рано или поздно алгоритм остановится: после очередного взятия остатка y будет равен нулю.

Будем предполагать, что $x > y$: если на шаге $x < y$, то со следующего рекурсивного вызова окажется, что $x > y$. Пусть x_i, y_i значение переменных на i-ом вызове. Тогда $y_{i+1} < x_{i+1} = y_i < x_i$, таким образом, последовательности y_i и x_i монотонно убывают, а поскольку они принимают только целые значения, то y_i рано или поздно окажется нулём.

Получение верхних и нижних оценок на алгоритм Евклида, а также расширенный алгоритм Евклида описаны в [ДПВ12].

Алгоритм Евклида задан рекурсивно. Нетрудно получить его итеративный аналог (рис. 2.2) и убедиться, что оба алгоритма находят наибольший общий делитель.

Но как перейти от рекурсии к итерации в общем случае? И всегда ли это возможно? Действительно всегда, ведь любой рекурсивный алгоритм

Вход : (x, y) — неотрицательные целые числа, $x > 0$
Выход: НОД(x, y)

```
1 while y > 0 do
2 |  b = x  mod y
3 |  x = y
4 |  y = b
5 end
6 return  x
```

Рис. 2.2: Итеративный алгоритм Евклида.

исполняется на итеративной модели вычислений (нашем компьютере). Как же компьютер исполняет рекурсивные алгоритмы? Мы опишем общую идею, а технические детали оставим для курсов об архитектуре ЭВМ и языке ассемблер.

В общем случае, рекурсивный алгоритм устроен как описано в псевдокоде слева на рис. 2.3. Внутри рекурсивной функции могут быть как самовызовы (строка 2) либо возвраты значений и выход из рекурсии (строка 3). И то и другое может встречаться неоднократно, но при переходе от рекурсии к итерации строки вида 2 и 3 обрабатываются одинаково.

В случае рекурсивного вызова (строка 2), в какой-то момент произойдёт возврат и переменная z будет определена, остальные переменные будут иметь те же значения, что и до вызовы. Поэтому, для итеративного исполнения рекурсивного алгоритма используют стек. В стеке размещают значения переменных, которые были определены при вызове функции и в результате исполнения кода в «Теле рекурсии-1», а также номер строки, в которую следует вернуться после выхода из рекурсивного вызова.

Сам рекурсивный вызов осуществляется просто: после помещения указанных данных в стек, происходит переход к первой строке first_line (уже бывшей рекурсивной) функции с новым значением входной переменной.

При выходе из рекурсии (строка 3) нужно не просто вернуть вычисленное значение, но и вернуться к исполнению функции, в случае, если выход происходит из рекурсивного вызова. Для этого происходит извлечение значений переменных, определённых на предыдущем уровне рекурсии (строка 8) и возврат командой **goto** к строке, в которой происходит присваивание значения z, вычисленного рекурсивно. В случае, если выход был не из рекурсивного вызова, а из обычного (самого первого

```
1  Function Rec(x) :
       /* first_line */ /* Тело
       рекурсии-1                  */
2      z = Rec(y)
       /* Тело рекурсии-2          */
3      return val
       /* Тело рекурсии-3          */
4  end
```

```
1  Function Iter(x) :
       Вход  : x
       Выход : Rec(x)
2      stack s
3      s.push(11, 0, . . . , 0)
       /* first_line                    */
       /* Тело рекурсии - 1             */
       /* вместо z = Rec(y)             */
4      s.push(7, v_1, . . . , v_n)
5      x = y
6      goto first_line
7      z = rv
       /* Тело рекурсии - 2             */
       /* вместо return val             */
8      (line, v_1, . . . , v_n) = s.pop();
9      rv = val;
10     goto line;
       /* Тело рекурсии-3               */
11     return rv
12 end
```

Рис. 2.3: От рекурсии к итерации

вызова функции), то полученное значение возвращается функцией `Iter`: в начале исполнения, в стек кладётся набор переменных (11, 0, . . . , 0), а значит при последнем возврате произойдёт переход к строке 11, которая и вернёт значение функции `Iter`(x).

Заметим, что в итеративном варианте алгоритма Евклида, как и во многих других итеративных алгоритмов, полученных из рекурсивных, стек не используется. Стек часто можно исключить, разобравшись с тем как устроено исполнение итеративного алгоритма. А порой стек можно исключить чисто механически (что и делают компиляторы при преобразовании рекурсивных функций к итеративным). Если рекурсивный вызов был совершён прямо перед выходом из рекурсивной функции, то такой вызов называют *хвостовым*. Заметим, что в случае возврата к исполнению функции после хвостового вызова никакие операции больше

выполняться не будут, а значит, что можно было бы и не возвращаться — отбросить хвост и не использовать здесь стек. В рекурсивном алгоритме Евклида используется как раз хвостовой вызов (проверьте это!).

Лекция 3

Алгоритмы «разделяй и властвуй»

Литература: [ДПВ12; КЛРШ05; КЛР02]

[ДПВ12] (раздел 0.2 и задачи) Числа Фибоначчи. Вычисление через

- рекурсию
- рекурсию с запоминанием
- итерацию
- возведение матрицы в степень

- Деревья рекурсии. Доказательство Θ-оценок для алгоритмов:

[ДПВ12] Алгоритм Карацубы

- Сортировка слиянием

[КЛР02; КЛРШ05] Анализ рекуррентных соотношений. Доказательство основной теоремы о рекурсии

Лекция 4

Сортировки I

Литература: [КЛРШ05; КЛР02]

- Детерминированный алгоритм поиска k-ой порядковой статистики.
- Быстрая сортировка (детерминированный алгоритм).

Лекция 5

Сортировки II. Нижние оценки

Литература: [Вял и др.18; КЛРШ05; КЛР02] Сортировки сравнениями. Модель разрешающих деревьев, доказательство нижних оценок.

- Игра «угадай число от 1 до N».
- Доказательство оценки $\Omega(n \log n)$ для сортировок сравнениями.
- Бинарный поиск. Нижняя оценка на поиск элемента в отсортированном массиве.
- Задача поиска $F^{-1}(x)$ для монотонной функции.
- Сортировка вставками
- Сортировка за линейное время.
  - Сортировка подсчётами
  - Поразрядная сортировка (Radix sort)

Список литературы

[ДПВ12] *Дасгупта С.*, *Пападимитриу Х.*, *Вазирани У.* Алгоритмы. — М.: МЦНМО, 2012.

[КЛРШ05] *Кормен Т.*, *Лейзерсон Ч.*, *Ривест Р.*, *Штайн К.* Алгоритмы: построение и анализ. — 2-е. — М.: Вильямс, 2005.

[КЛР02] *Кормен Т.*, *Лейзерсон Ч.*, *Ривест Р.* Алгоритмы: построение и анализ. — М.: МЦНМО, 2002.

[Шен04] *Шень А. Х.* Программирование: теоремы и задачи. — М.: МЦНМО, 2004.

[КФ12] *Кузюрин Н. .*, *Фомин С. .* Эффективные алгоритмы и сложность вычислений. — 2012.

[Вял и др.18] *Вялый М.*, *Подольский В.*, *Рубцов А.*, *Шварц Д.*, *Шень А.* Лекции по дискретной математике. — Черновик: http://rubtsov.su/public/hse/2017/DM-HSE-Draft.pdf, 2018.

[ЖФФ12] *Журавлёв Ю. И.*, *Флёров Ю. А.*, *Федько О. С.* Дискретный Анализ. Комбинаторика. Алгебра логики. Теория графов. — М.: МФТИ, 2012.

[ЖФВ07] *Журавлёв Ю. И.*, *Флёров Ю. А.*, *Вялый М. Н.* Дискретный Анализ. Основы высшей алгебры. — М.: МЗ-пресс, 2007.

[Lei96] *Leighton T.* Notes on Better Master Theorems for Divide-and-Conquer Recurrences // Lecture notes, MIT. — 1996.