Ozon Masters. Алгоритмы

план-конспект лекций

Александр Рубцов, Дмитрий Киранов, Кирилл Чеканов
alex@rubtsov.su

Содержание

Об этом файле

В этом файле будут приведены конспекты первых лекций по курсу «Алгоритмы», читаемому автором в Ozon Masters 2020, а далее будет приведён план-конспект остальных лекций. Файл будет обновляться каждую неделю. Курс сильно пересекается с курсом МФТИ, конспекты к которому подготовлены совместно с Дмитрием Кирановым и Кириллом Чекановым.

Лекция 0

Введение. Верхние и нижние оценки сложности алгоритмов

Литература: [ДПВ12; КЛРШ05; 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).

0.1 Введение

Курс «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, поэтому мы не будем тратить время на формализацию этих понятий. Тем более, что каждый из вас интуитивно понимает смысл этих понятий и вряд ли никогда не сталкивался с алгоритмами в современном мире.

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.

0.1.1 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 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 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,*

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

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.

0.2 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 $f : \mathbb{N}_1 \to \mathbb{N}_1$ (под $\mathbb{N}_1$ 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 $f : \mathbb{N}_1 \to \mathbb{R}_+$, 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.

0.2.1 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

$$\exists C > 0, N \in \mathbb{N}_1 \; \forall n \geqslant N : f(n) \leqslant Cg(n).$$

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

0.2.2 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 $\mathbb{N}_1$ в $\mathbb{R}_+$, но и функции из $\mathbb{N}_1$ 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 $f : [0, +\infty) \to \mathbb{R}_+$ — монотонная и непрерывная функция и $F(x)$ — её первообразная. Тогда*

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

Так, $\displaystyle\sum_{k=1}^{n} \frac{1}{n^2} = \Theta\left(\frac{1}{n}\right)$.

0.2.3 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.

0.2.4 Θ-эквивалентность и сравнение функций

Оставляем читателю проверить, что отношение « $f(n) = \Theta(g(n))$ » на множестве функций $\mathbb{N}_1 \to \mathbb{R}_+$ 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 формулой. 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 1

Жадные алгоритмы

Литература: [Шен04; КФ12]

1. Пример: задача о поиске треугольника максимальной площади, сторона которого лежит на оси Ox.
2. Жадные алгоритмы и индуктивные функции [Шен04].
   - функции максимума и суммы — индуктивные
   - индуктивное расширение на примере поиска максимума произведения
3. Онлайн-алгоритмы

1.1 Пример

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.

Неформально, алгоритм называется *жадным*, если для него выполняется принцип «дают бери, а бьют — беги». 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.

1.2 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 7. *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 $y_{i-1} = f(x_1, \ldots, x_i)$ 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 8. *Функция* $f(x_1, \ldots x_n) = \max\limits_{i \neq j} x_i \times x_j$, *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

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

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

Пример 9 (продолжение примера 6)**.** *Пусть* $f((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n))$ *возвращает максимальную площадь треугольника, одна из сторон которого лежит на оси* Ox. *Построим для неё индуктивное расширение* g.

Функция g возвращает координаты трёх точек $((x_{\max}, 0), (x_{\min}, 0), (x, y))$, которые являются вершинами треугольника максимальной площади. Чтобы определение g было формально корректным потребуем, что бы в случае, если одна из подходящих точек не была ещё считана, в этой компоненте было записано число -1, а не пара чисел. Функция g индуктивна: построим для неё функцию G, такую что

$$G(g((x_1, y_1), (x_2, y_2), \ldots, (x_{n-1}, y_{n-1})), (x_n, y_n)) = g((x_1, y_1), \ldots, (x_n, y_n)).$$

Функция G получат на вход выход $((x_{\max}, 0), (x_{\min}, 0), (x, y))$ функции g и точку (x_n, y_n). Если $y_n = 0$, то в случае если $x_n > x_{\max}$, $x_{\max}$ меняется на x_n, а если $x_n < x_{\min}$, то $x_{\min}$ меняется на x_n. Если же $y_n \neq 0$, то если $|y_n| > |y|$, то точка (x, y) меняется на (x_n, y_n). В остальных случаях, выход функции g остаётся без изменений. Корректность такого пересчёта ясна из корректности исходного алгоритма; в этом примере мы просто формализовали его через индуктивные функции. Осталось определить

функцию $t(g(\ldots))$, которая возвращает искомую площадь $\frac{1}{2}(x_{\max} - x_{\min})\, y$ или -1, если одна из компонент входа равна -1. □

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 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 индуктив-

ное 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.

1.4 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.

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.

- Напоминание об используемой модели вычислений и оценке сложности алгоритмов
- Алгоритм Евклида ([ДПВ12] 1.2.3)
- Доказательство нижних оценок на время работы алгоритма Евклида через числа Фибоначчи
- Общая схема перехода от рекурсии к итерации
- Расширенный алгоритм Евклида ([ДПВ12] 1.2.3)
- Числа Фибоначчи ([ДПВ12] 0.2, задача 0.4). Вычисление через
  - рекурсию
  - рекурсию с запоминанием
  - итерацию
  - возведение матрицы в степень

2.1 Напоминание об используемой модели вычислений и оценке сложности алгоритмов

Предложим читателю обратиться к примеру 2, дабы напомнить, что сложность алгоритма измеряется от длины входа. Так, если на вход алгоритма подаётся число N, то длина входа n есть $\lceil \log_2 N \rceil$; мы берём двоичный логарифм, поскольку в компьютерах все числа хранятся в двоичной системе счисления. Также напомним, что функция $f(n)$, возвращающая время работы алгоритма на входе длины n возвращает максимальное время работы, т. е. сложность оценивается для худшего случая (а не среднего и не лучшего!).

Эти замечания существенны, когда речь заходит про числовые алгоритмы, с которыми мы познакомимся на этой неделе. Один из самых древних из них — алгоритм Евклида, который находит наибольший общий делитель двух чисел x и y (НОД(x, y)). Поскольку на вход алгоритму подаётся два числа, то мы считаем, что длина входа равна сумме длин чисел: $n = \lceil \log_2 x \rceil + \lceil \log_2 y \rceil$. На практике это не так: если на вход подаётся два числа, то нужно знать где заканчиваются биты одного и начинаются биты другого, а для этого используют либо хитрую кодировку, либо требуют чтобы оба числа имели запись одинаковой длины (одна из них может содержать ведущие нули) но мы не обращаем внимание на эти технические детали, поскольку нас интересует только порядок роста.

Что же важно для нас, так это правильно учитывать время исполнения операций. Поскольку мы работаем с алгоритмами, реализующими элементарную арифметику быть может на очень больших числах (что бывает на практике в криптографии, как мы увидим дальше), мы не можем считать, что арифметические операции и операции сравнения выполняются за $O(1)$.

Напомним, что в качестве исполнителя алгоритмов мы используем язык Си и считаем, что каждая допустимая операция в нём и есть элементарная операция в нашей модели. Поскольку в теоретических моделях длина входа не ограничена, то мы разделяем два случая модели. Первый — когда переменные вмещают сколь угодно большие числа и все операции с ними (арифметические, сравнения и т. п.) стоят $O(1)$; такие переменные мы называем *регистрами*. Вторая, когда никаких чудо-регистров нет и в качестве единицы памяти выступает бит — эта модель ближе к практике, чтобы оценить сложность в ней нужно подсчитать количество операций с

битами. Первую модель мы использовали на прошлой неделе, поскольку часто на практике известно, что реальные входные данные ограничены (пусть большим числом N), однако явные ограничения задавать в модели нехорошо, потому что используя ограниченность и формально константную память (порядка 2^N) можно получить для задач линейные алгоритмы, которые будут абсолютно не практичны, в силу невозможности использовать огромную память. Формальным аналогом нашей первой модели является RAM-модель.

2.2 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 рекурсии к итерации. 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 неотрицательные целые значения, то y_i рано или поздно окажется нулём.

Получение верхней оценки на алгоритм Евклида, а также расширенный алгоритм Евклида описаны в [ДПВ12]. Приведём здесь нижнюю оценку.

Утверждение 3. *Алгоритм Евклида в худшем случае делает $\Omega(n)$ рекурсивных вызовов, где $n = \lceil \log_2 x \rceil + \lceil \log_2 y \rceil$ — длина входа.*

Замечание 3. *Для доказательства нижней оценки нам достаточно привести бесконечную последовательность входных данных (пар чисел), на которых алгоритм Евклида делает $\Omega(n)$ операций. Напомним, что время работы алгоритма оценивается как время его работы в худшем случае на входе длины n.*

Доказательство утверждения 3. В качестве последовательности возьмём соседние числа Фибоначчи (F_k, F_{k-1}) и докажем, что на них алгоритм Евклида совершает порядка k операций. Докажем для этого, что

$$\text{НОД}(F_k, F_{k-1}) = \text{НОД}(F_{k-1}, F_{k-2}).$$

Числа Фибоначчи монотонно неубывают, а потому для каждого k справедливо $F_{k-1} \geqslant F_{k-2}$, и поскольку $F_k = F_{k-1} + F_{k-2}$, то отсюда $F_{k-1} \geqslant F_k/2$ (максимальное слагаемое не меньше половины суммы). Значит остаток от деления F_k на F_{k-1} равен разности этих чисел, которая равна F_{k-2}.

Итак, мы доказали, что на входе (F_k, F_{k-1}) алгоритм Евклида делает порядка k рекурсивных вызова (если быть точными, то $k-2$). Для получения нижней оценки осталось доказать, что длина k-ого числа Фибоначчи порядка k. Для этого достаточно вспомнить формулу Бине: согласно ней $F_k \sim \phi^k$, отсюда, и вытекает требуемый порядок роста. Однако эту оценку легко получить и явно, доказав, что

$$2^{\frac{k-1}{2}} \leqslant F_k \leqslant 2^k,\ k \geqslant 3,$$

что равносильно

$$\frac{k-1}{2} \leqslant \log_2 F_k \leqslant k.$$

Верхнюю оценку докажем по индукции для $k \geqslant 1$. База: $F_1 = 1 \leqslant 2^1$.

$$\text{Шаг: } F_k = F_{k-1} + F_{k-2} \leqslant 2 \times F_{k-1} \leqslant 2 \times 2^{k-1} = 2^k,$$

где переход в последнем неравенстве выполнен по предложению индукции. Доказательство нижней оценки аналогичное: $F_3 = 2 \geqslant 2^1$;

$$F_k = F_{k-1} + F_{k-2} \geqslant 2 \times F_{k-2} \geqslant 2 \times 2^{\frac{k-3}{2}} = 2^{\frac{k-1}{2}}.$$

□

Упражнение 4. Обратите внимание, что помимо рекурсивных вызовов алгоритм Евклида делает и иные операции, и среди них есть те, время выполнения которых зависят от длины входа, а потому нельзя считать, что они выполняются за $O(1)$. Оцените время работы алгоритма Евклида в битовой модели.

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

Вход : (x, y) — неотрицательные целые числа, $x > 0$
Выход: НОД(x, y)
while $y > 0$ **do**
    b = $x \mod y$
    x = y
    y = b
end
return x

Рис. 2.2: Итеративный алгоритм Евклида.

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

```
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

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 вызова. 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 вызова функции), то полученное значение возвращается функцией `Iter`: в начале исполнения, в стек кладётся набор переменных (**11**, $0, \ldots, 0$), 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 алгоритмах, 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!).

2.2.1 Расширенный итеративный алгоритм Евклида

Для начала рассмотрим конкретный пример, а затем приведем общий алгоритм и обоснование его корректности.

Рассмотрим уравнение $715x + 273y = 91$ и найдем его решение в целых числах:

В общем случае для того, чтобы существовало решение уравнения $ax + by = d$ в целых числах, необходимо и достаточно, чтобы d делилось на `gcd`(a, b). Проверим это, воспользовавшись для нахождения `gcd`$(715, 273)$ обычным алгоритмом Евклида:

$$\texttt{gcd}(715, 273) = \texttt{gcd}(273, 715 \pmod{273}) = \texttt{gcd}(273, 169) = \texttt{gcd}(169, 104) =$$
$$\texttt{gcd}(104, 65) = \texttt{gcd}(65, 39) = \texttt{gcd}(39, 26) = \texttt{gcd}(26, 13) = \texttt{gcd}(13, 13)$$
$$= \texttt{gcd}(13, 0) = 13$$

Теперь приступим к решению уравнения, будем заполнять таблицу, которую инициализируем следующим образом.

x	y	715x + 273y
1	0	715
0	1	273

На следующих шагах из предпоследней строки таблицы «покомпонентно» вычитаем последнюю строку таблицы, умноженную на целую часть от деления предпоследнего значения в 3-ей колонке на последнее: в данном случае $\frac{715}{273} = 2$. Тогда новая строка в таблице: $(1, 0, 715) - 2(0, 1, 273) = (1, -2, 169)$

x	y	715x + 273y
1	0	715
0	1	273
1	-2	169

Далее результатом взятия остатка фактически будет вычитание из предпоследнего значения 3-го столбца последнего: $273 - 169 = 104$, а значит параметры x и y определяются следующим образом: $(0, 1, 273) - (1, -2, 169) = (-1, 3, 104)$

Будем продолжать выполнять те же операции, пока в правом нижнем углу таблицы не появится $\texttt{gcd}(715, 273) = 13$.

x	y	715x + 273y
1	0	715
0	1	273
1	-2	169
-1	3	104
2	-5	65
-3	8	39
5	-13	26
-8	21	13

Теперь домножим последнюю строку таблицы на $\frac{91}{13} = 7$
$\rightarrow (7 \cdot -8, 7 \cdot 21, 7 \cdot 13)$
Откуда приходим к равенству: $715 \cdot (-56) + 273 \cdot 147 = 91$
То есть одним из решений уравнения является пара: $(-56, 147)$. Обратите внимание, что этот алгоритм находит только одно (частное решение), нахождение общего решения оставляем читателю в качестве упражнения (или на изучение литературы).

Алгоритм

Вход: уравнение $ax + by = d$
Выход: одно из целочисленных решений уравнения

1. Проверяем, что d делится на $\texttt{gcd}(a, b)$.

2. Инициализируем таблицу следующим образом:

x	y	$ax + by$
1	0	a
0	1	b

3. Затем будем заполнять таблицу в соотвествии с правилом, описанным выше до тех пор, пока в правом нижнем углу не окажется значение $\texttt{gcd}(a, b)$

x	y	$ax + by$
1	0	a
0	1	b
...	...	...
a_i	b_i	c_i
a_{i+1}	b_{i+1}	c_{i+1}
$a_i - [\frac{c_i}{c_{i+1}}]a_{i+1}$	$b_i - [\frac{c_i}{c_{i+1}}]b_{i+1}$	$c_i - [\frac{c_i}{c_{i+1}}]c_{i+1}$
...	...	...
a_n	b_n	$\texttt{gcd}(a, b)$

4. Домножаем последнюю строку таблицы на $\frac{d}{\texttt{gcd}(a,b)}$, как не трудно заметить полученные значения x и y будут являться решением исходного уравнения.

Корректность. В первых двух столбцах мы задаем значения аргументов x и y, в третьем значение линейной функции по этим аргументам, поэтому при сложении двух строк с произвольными коэффициентами получаем новую строку, где значения в соотвествующих столбцах будут линейной комбинацией исходных строк с выбранными коэффициентами. При этом на каждом шаге алгоритма в третьем столбце «идут» шаги обычного алгоритма Евклида, который заканчивается наибольшим общим делителем коэффициентов исходного уравнения. А зная значения параметров x и y для НОД(a, b), по линейности можно найти решение исходного уравнения.

2.2.2 Числа Фибоначчи

Начнём с рекурсивного алгоритма, описанного псевдокодом 2.4. Очевидно, что это экспоненциальный алгоритм.

```
1 Function F(n) :
  |  Вход  : n — целое положительное число
  |  Выход : n-ое число Фибоначчи
2 |  if n < 3 then
3 |  |  return 1
4 |  end
5 |  return F(n − 1) + F(n − 2)
6 end
```

Рис. 2.4: Рекурсивный алгоритм вычисления чисел Фибоначчи

Рассмотрим для приведенного алгоритма дерево рекурсивных вызовов при $n = 5$ (рис. 2.5).

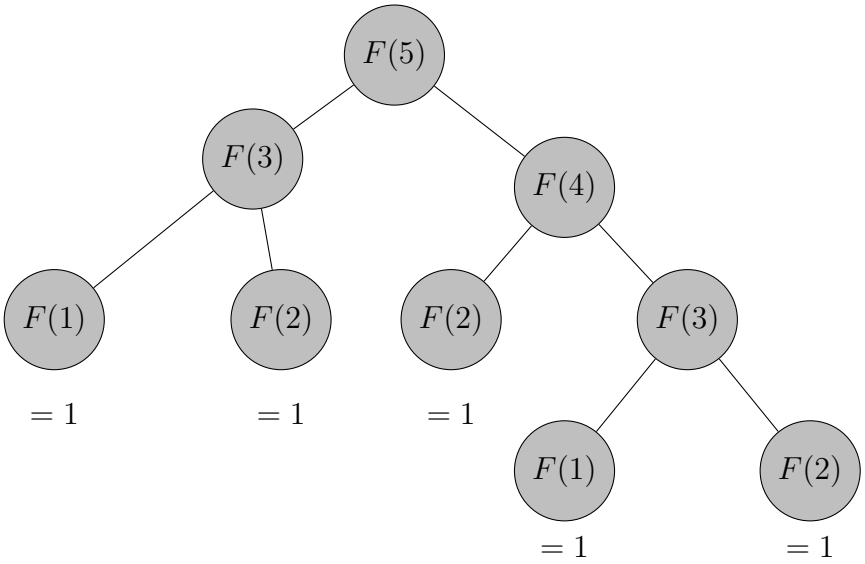


Рис. 2.5: Дерево рекурсивных вызовов алгоритма 2.4

Из рисунка видно, что даже при малых n алгоритм вычисляет одно и то же значение несколько раз. Чтобы решить эту проблему, можно ис-

пользовать модифицированный алгоритм 2.6, который будет запоминать уже вычисленные значения и не будет вычислять их заново, его дерево вызовов приведено на рис. 2.7.

```
1 //f[] — глобальный массив, заполненный -1
2 f[1] = 1, f[2] = 1;
3 Function F2(n) :
      Вход  : n — целое положительное число
      Выход : n-ое число Фибоначчи
4     if f[n] ≠ −1 then
5         return f[n]
6     end
7     f[n] = F2(n − 1) + F2(n − 2);
8     return f[n]
9 end
```

Рис. 2.6: Вычисление через рекурсию с запоминанием

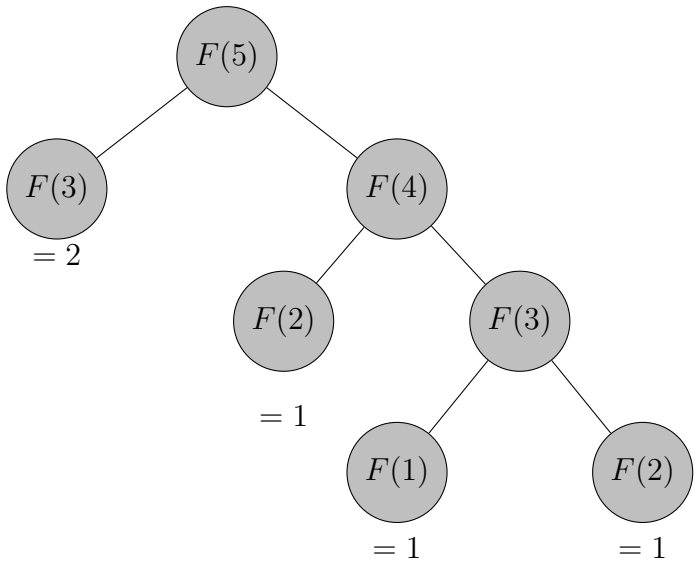


Рис. 2.7: Дерево рекурсивных вызовов алгоритма 2.6

Здесь мы полагали, что алгоритм сначала пошёл в правое поддерево,

где было вычисленно и сохранено значение $F(3)$, тогда попадая затем в левую часть поддерева значение $F(3)$ будет восстановлено из сохраненного, а не перевычисленно заново. Данный алгоритм является значительно более эффективным на практике, особенно при малых n.

Итеративный псевдополиномиальный алгоритм

Алгоритм 2.6 хоть и значительно лучше простого рекурсивного алгоритма 2.4, но не эффективен по памяти, поскольку хранит в ней всё дерево рекурсивных вызовов. Эффективность по памяти можно существенно улучшить, перейдя от рекурсии с запоминанием к итерации.

```
1 Function F3(n) :
      Вход  : n — целое положительное число
      Выход : n-ое число Фибоначчи
2     a = 1, b = 1
3     for i ← 3 to n do
4         c = a;
5         a = a + b;
6         b = c;
7     end
8     return a
9 end
```

Рис. 2.8: Итеративный алгоритм

В зависимости от рассматриваемой модели, можно по разному оценивать сложность приведенного выше алгоритма: если считать, что арифметические операции с числами стоят $O(1)$, то время работы равно $O(n)$. Заметим, что даже в этом случае алгоритм не полиномиален, поскольку сложность измеряется от длины входа, то есть от $\lceil \log_2 n \rceil$ — число n поступает на вход в двоичной записи; такие алгоритмы называют псевдополиномиальными: алгоритм будет полиномиальным, если подать на вход число n в унарной записи, т. е. n единиц.

Если же учитывать размер слагаемых($2^{\frac{n-1}{2}} \leqslant F_n \leqslant 2^n$), то на сложение двух чисел $a, b \in (2^{\lfloor \log_2(n-1) \rfloor - 1}, 2^{\lceil \log_2 n \rceil})$ потребуется $\Theta(\log n)$ битовых операций. Учитывая оценку на последние, числа Фибоначчи получаем, что в конце на каждое сложение приходится $O(n)$ битовых операций, откуда получаем, что время работы пропорцианально $O(n^2)$, что все равно

существенно лучше рекурсивного алгоритма.
Однако по длине входа этот алгоритм тоже будет экспоненциальным: длина входа $l = \lceil \log n \rceil$ битов, откуда в соотвествии с приведенными рассуждениями на n, убеждаемся, что зависимость от l экспоненциальная.

Лекция 3

Алгоритмы «разделяй и властвуй»

Литература: [ДПВ12; КЛРШ05; КЛР02]

- Деревья рекурсии. Доказательство Θ-оценок для алгоритмов:
  - Сортировка слиянием

[ДПВ12] Алгоритм Карацубы

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

Приведём используемую нами формулировку основной теоремы о рекурсии (использование более сильных теорем в качестве основной теоремы будет незасчитано).

Теорема 1. *Пусть* $n, a, b \in \mathbb{N} \setminus \{0\}$ *и* $d = \log_b a$. *Для рекуррентных соотношений вида* $T(n) = aT(n/b) + f(n)$, $T(n) = aT(\lfloor n/b \rfloor) + f(n)$ *и* $T(n) = aT(\lceil n/b \rceil) + f(n)$ *справедливо следующее утверждение.*

1. *Если* $f(n) = O(n^{d-\varepsilon})$ *для некоторого* $\varepsilon > 0$, *то* $T(n) = \Theta(n^d)$.
2. *Если* $f(n) = \Theta(n^d)$, *то* $T(n) = \Theta(n^d \log n)$.
3. *Если а)* $f(n) = \Omega(n^{d+\varepsilon})$ *для некоторого* $\varepsilon > 0$ *и*
   б) $\exists c : 0 < c < 1, af(n/b) \leqslant cf(n)$, *то* $T(n) = \Theta(f(n))$.

3.0.1 Сортировка слиянием

Основной идеей сортировки слиянием является применение принципа «разделяй и властвуй». В его основе лежит следующая идея: исходная задача разбивается несколько простых подзадач, подобных исходной, которые решаются рекурсивно(для простых подзадач решения получаем непосредственно), полученные решения комбинируются для получения решения исходной задачи.

Алгоритм сортировки слиянием

1. Делим n элементную сортируемую последовательность на две подпоследовательности по $n/2$ элементов.
2. Рекурсивно сортируем эти две подпоследовательности с использованием сортировки слиянием.
3. Соединяем две подпоследовательности для получения отсортированной последовательности.

Рассмотрим пример. Требуется с помощью сортировки слиянием отсортировать массив: $[1, 7, 3, 8, 2, 6, 5, 4]$. На первом шаге массив разбивается на две равных половины:

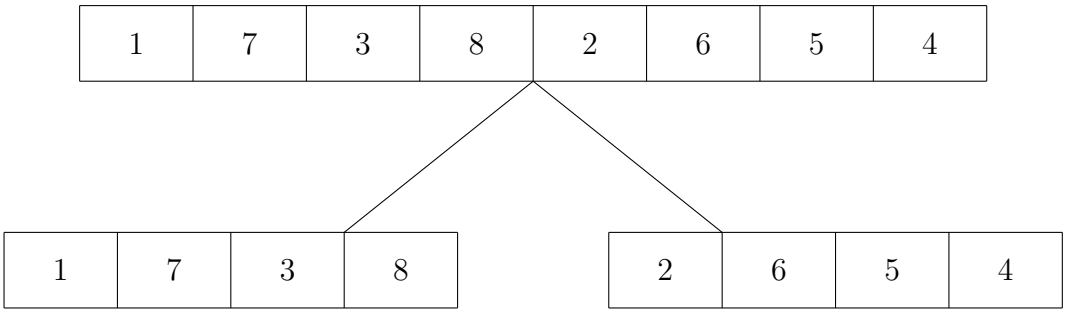
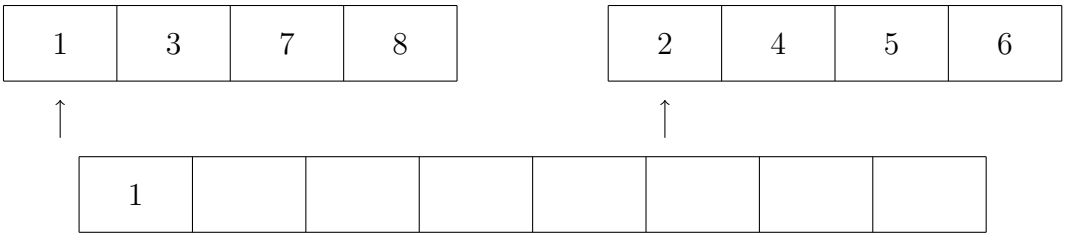


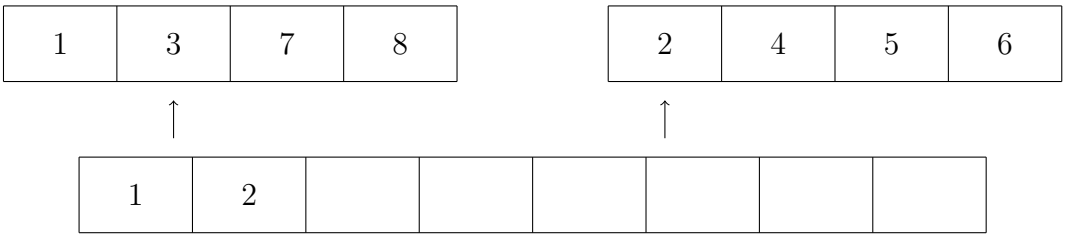
Рис. 3.1: Первый шаг сортировки слиянием.

Далее мы разберем пример до конца, но сначала продемонстрируем процедуру слияния. Предположим, что мы уже отсортировали половины исходного массива. Для двух отсортированных массивов заводим указатели, которые сначала будут указывать на крайние левые элементы, затем

будем сравнивать значения элементов в массивах, на которые указывают указатели, и меньший заносить в новый массив, а указатель этого элемента сдвигать на одно значение вправо:

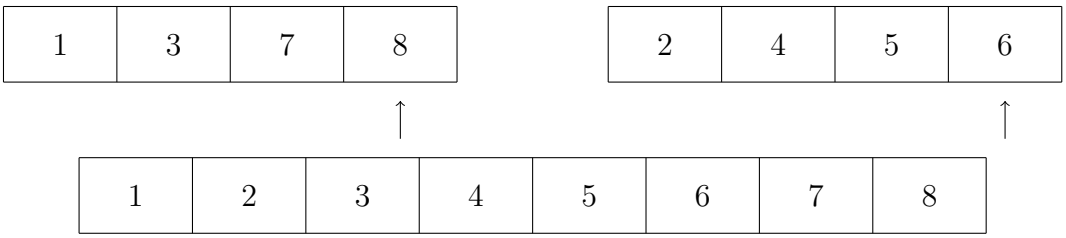


Передвигаем указатель, вновь сравниваем элементы над указателями и заносим меньшее число:



Эта процедура повторяется до тех пор пока весь итоговый массив не будет заполнен.

Конечная конфигурация будет выглядеть так:



Как видно, в результате слияния двух отсортированных половин массива, мы получили отсортированный исходный массив.

Оценить время описанной процедуры, которая называется «Слияние»(англ. «Merge») совсем просто, оно равно кол-ву сдвигов указателей(или числу сравнений элементов) - $O(n)$.

Ознакомившись с процедурой слияния можем вернуться к примеру.

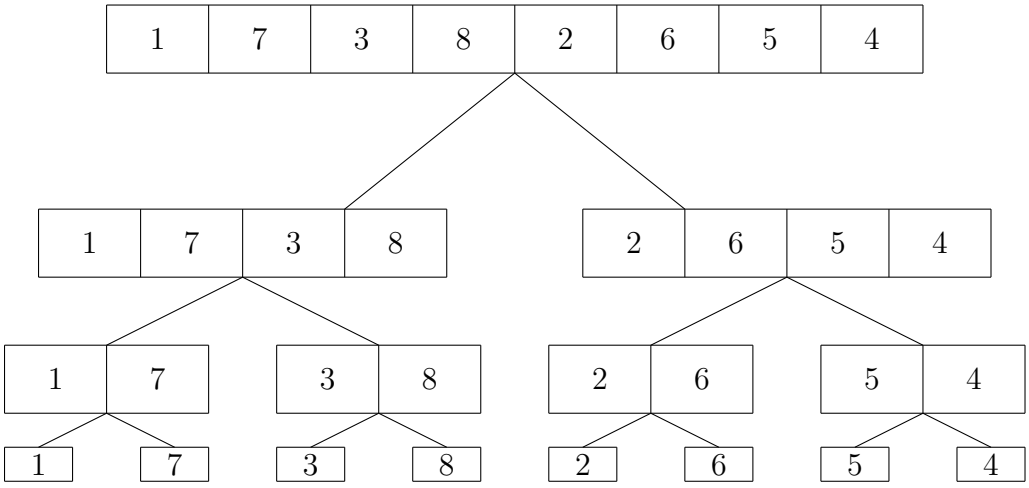


Рис. 3.2: Конфигурация рекурсивных вызовов на этапе разбиения.

Очевидно, что массивы длины один отсортированы, поэтому можно приступать к процедуре слияния, описанной выше.

Слияние массивов длины один и два оставим для читателя, слияние массивов длины четыре приведено выше.

Корректность: для доказательства корректности алгоритма докажем корректность процедуры «слияния». Покажем, что на k-ом шаге работы процедуры слияния алгоритм выбирает k-ую порядковую статистику массива, состоящего из объедения двух уже отсортированных массивов, полученных на предыдущем уровне рекурсии. (k-ой порядковой статистикой набора элементов линейно упорядоченного множества называется такой его элемент, который является k-ым элементом набора в порядке сортировки). Предположим, утверждение выше верно для $k \leqslant n$, покажем, что оно верно для $k = n+1$. Пусть в некоторый момент времени на $n+1$ шаге работы фиксированы указатели p, q для соответствующих

массивов x, y, по предположению $x[i], y[j], i < p, j < q$ формируют первые k порядковых статистик, следовательно $x[p], y[q]$ больше любого из уже рассмотренных элементов и в то же время в силу отсортированности массивов x, y минимальным элементом из нерассмотренных является $\min(x[p], y[q])$. В итоге получаем, что на $n + 1$ шаге алгоритм выбирает $n + 1$ порядковую статистику. База полученной индукции очевидна, из вышесказанного получаем корректность процедуры «слияния».

Оценим время работы описанного алгоритма, оно задаётся рекуррентным соотношением

$$T(n) = 2T(n/2) + cn.$$

На каждом шаге массив разбивается на два подмассива меньшей длины, для которых мы рекурсивно вызываем процедуру сортировки слиянием (за это отвечает слагаемое $2T(n/2)$, выполнив сортировку двух подмассивов их нужно слить, откуда в силу времени работы процедуры «слияния» имеем приведенное рекуррентное соотношение. Построим *дерево рекурсии*, чтобы понять как решать подобные соотношения:

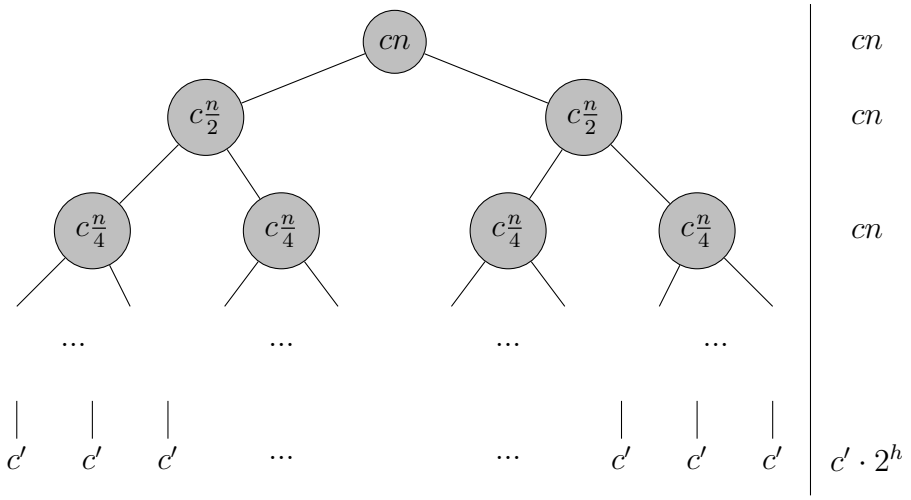


Рис. 3.3: Дерево рекурсии.

Самый верхний уровень будем считать нулевым. В вершинах дерева рекурсии стоит время затраченное алгоритмом только на операции на

этом уровне рекурсии (остальные операции учитываются на тех уровнях, на которых они выполнялись). В случае сортировки слиянием, в вершинах записано время, которое подзадача тратит на дальнейшее разбиение и слияние после окончания работы рекурсивных вызовов (сборки ответа из ответов к подзадачам). В данном случае исходная подзадача разбивается на две подзадачи вдвое меньшего размера, которые в свою очередь разбиваются на две подзадачи вдвое меньшего размера и так далее.

Оценим число операций на каждом уровне: нулевой уровень - cn операций, первый уровень - $cn/2 + cn/2 = cn$, ... так до последнего уровня уровня, на котором для решения одной задачи треубется лишь некоторая константа c', независящая от n. Обозначим за h — высоту полученного дерева, она очевидно вычисляется из уравнения $\frac{n}{2^h} = 1$, откуда $h = \log_2 n$; тогда общее число операций на последнем уровне $c' \cdot 2^h$. Вершины на последнем уровне будем называть *кроной*, а остальные вершины — *внутренними*.

Теперь подсчитаем общее число операций в дереве, как увидим далее, в зависимости от параметров в рекуррентном соотношении основной вклад в число операций могут вносить как внутренние вершины, так и крона, поэтому подсчет для них мы будем проводить раздельно[1].

Внутренние вершины: $cn(h-1) = cn(\log_2 n - 1)$.

Крона: $c' \cdot 2^h = c'n$

Итого: $cn \log_2 n - cn + c'n = \Theta(n \log_2 n)$

3.0.2 Быстрое умножение - Алгоритм Карацубы

Опишем алгоритм позволяющий выполнять опрецию умножения двух чисел A и B битовой длины n за время $O(n^{\log_2 3})$.

Представим эти числа в виде $A = \overline{ab}$, $B = \overline{cd}$ где a, c - старшие половины двоичной записи, b, d - младшие половины двоичной записи (для числа $11 = 1011_2$ $a = 10_2 = 2$, $b = 11_2 = 3$).

В этом случае:

$$A \times B = \overline{ab} \times \overline{cd} = (a \times 2^{n/2} + b)(c \times 2^{n/2} + d) = ab2^n + bd + 2^{n/2}(ad + bc).$$

Заметим, что $(ad+bc) = (a+b)(c+d) - ac - bd$, блогодаря чему получаем, что для вычисления искомого произведения n-битовых чисел, необходимо

[1]При должном старании можно свести всё к анализу только внутренних вершин. Введённое нами разделение нужно только для удобства в доказательствах.

вычислить 3 произведения (ac, bd, $(a+b)(c+d)$) чисел битовой длинны $n/2$, и найти соответсвующую сумму затратив на это $\Theta(n)$ элементарных операций. Таким образом приходим к рекурентной формуле:

$$T(n) = 3T\left(\frac{n}{2}\right) + cn$$

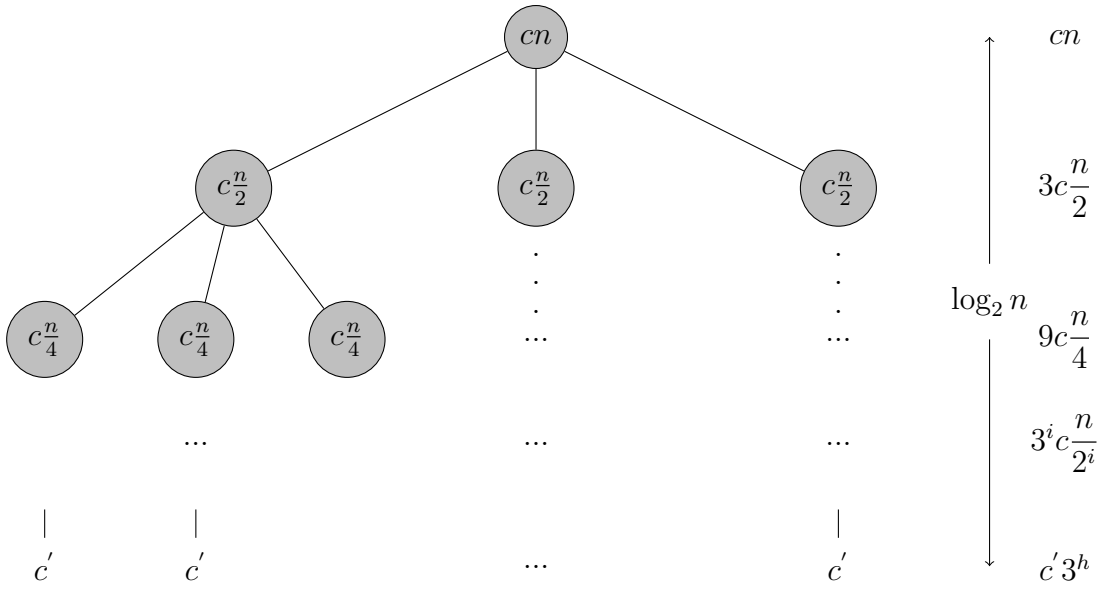


Рис. 3.4: Дерево рекурсии для алгоритма Карацубы

Найдём с помощью дерева рекурсии на рис. 3.4 время работы алгоритма; как и раньше оценим по отдельности вклад внутренних вершин и кроны ($h = \log_2 n$):

$$\sum_{i=0}^{h-1} c\left(\frac{3}{2}\right)^i n + c'3^h = cn\frac{1-(3/2)^{h-1}}{1-3/2} + c'3^h =$$

$$= 2cn\left(\frac{n^{\log_2 3}}{n} - 1\right) + c'n^{\log_2 3} = \Theta(n^{\log_2 3}).$$

Далее рассмотрим теорему, позволяющая быстро и удобно решать подобные рекуррентные соотношения.

3.0.3 Основная теорема(Master theorem)

Имеется рекурентное соотношение

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

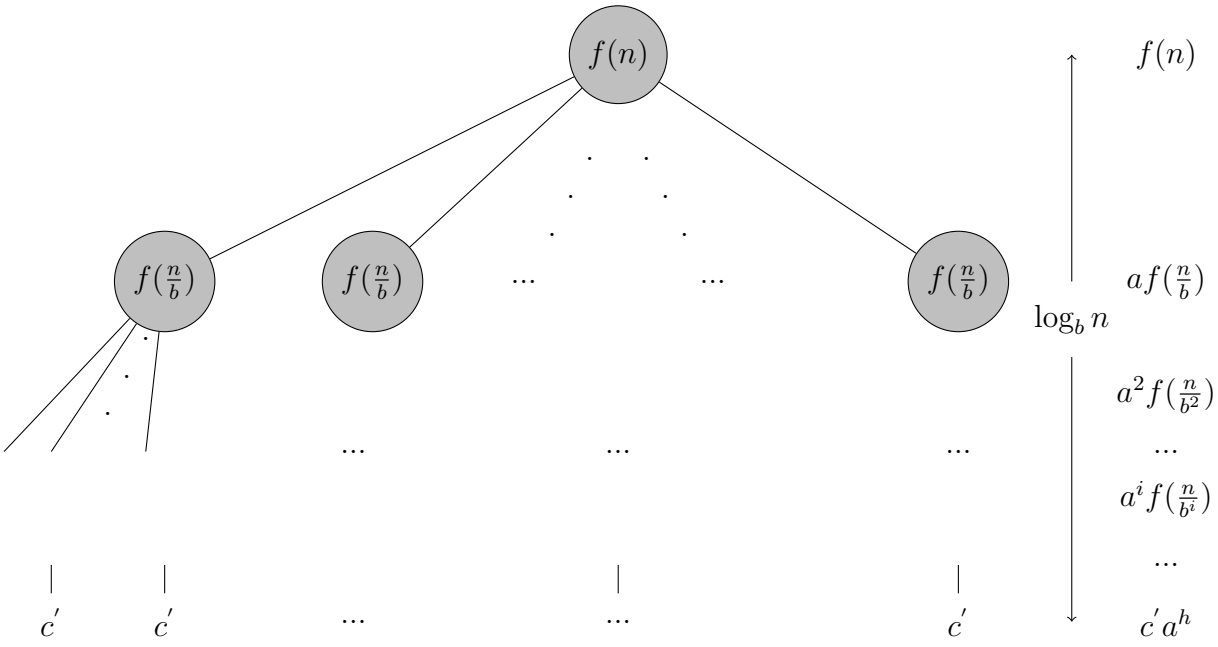
в котором $f(n) = \Theta(1)$ при малых n. Обозначим $d = \log_b a$. Тогда справедливы следующие утверждения.

1. Если $\exists\ \varepsilon > 0 : f(n) = O(n^{d-\varepsilon})$, то $T(n) = \Theta(n^d)$.
2. Если $f(n) = \Theta(n^d)$, то $T(n) = \Theta(n^d \log n)$.
3. Если $\exists\ \varepsilon > 0$ и одновременно выполняются условия
   (a) $f(n) = \Omega(n^{d+\varepsilon})$,
   (b) $\exists c\ \ 0 < c < 1 : af\left(\frac{n}{b}\right) \leqslant cf(n)$,

   то $T(n) = \Theta(f(n))$.

Доказательство

Построим дерево рекурсии для общего случая: исходная подзадача разбивается на a подзадач, каждая размером n/b



Как нетрудно понять, высота дерева $h = \log_b n$. В дальнейшем анализе нам будут часто требоваться следующие соотношения: $a^d = b$ (напомним, что $d = \log_b a$), $a^h = a^{\log_b n} = b^{\log_b a \cdot \log_b n} = n^d$. Выразим $T(n)$ через число операций во внутренних вершинах и в кроне:

$$T(n) = \sum_{i=0}^{h-1} a^i f\left(\frac{n}{b^i}\right) + c'n^d. \tag{1}$$

1. Учтем, что $f(n) = O(n^{d-\varepsilon})$ и оценим первое слагаемое суммы (1); примечания к переходам вынесены в сноски.

   $\sum_{i=0}^{h-1} a^i f\left(\frac{n}{b^i}\right) \leqslant \sum_{i=0}^{h-1} a^i c\left(\frac{n}{b^i}\right)^{d-\varepsilon} \leqslant cn^d \sum_{i=0}^{h-1} \frac{a^i}{(b^d)^i} \cdot \frac{n^{-\varepsilon}}{b^{-i\varepsilon}} \overset{2}{=} cn^{d-\varepsilon} \sum_{i=0}^{h-1} (b^\varepsilon)^i \overset{3}{=}$

   $= cn^{d-\varepsilon} \dfrac{(b^\varepsilon)^h - 1}{b^\varepsilon - 1} \overset{4}{=} \Theta(n^d).$

   Откуда $\sum_{i=0}^{h-1} a^i f\left(\frac{n}{b^i}\right) = O(n^d)$.

   Кроме того, как было показано выше:

   $T(n) = \sum_{i=0}^{h-1} a^i f\left(\frac{n}{b^i}\right) + c'n^d > c'n^d$, а значит $T(n) = \Omega(n^d)$.

   Откуда приходим к утверждению первого пункта теоремы.

2. Также оценим первое слагаемое суммы (1), учитывая теперь, что $f(n) = \Theta(n^d)$.

   $\sum_{i=0}^{h-1} a^i f\left(\frac{n}{b^i}\right) = \sum_{i=0}^{h-1} a^i \left(\frac{n}{b^i}\right)^d = \sum_{i=0}^{h-1} n^d = n^d \cdot h = n^d \cdot \log n = \Theta(n^d \log n).$

   Таким образом, $T(n) = \Theta(n^d \log n) + c'n^d = \Theta(n^d \log n)$.

3. Начнём с условия (b): $af\left(\frac{n}{b}\right) \leqslant cf(n) \overset{5}{\Rightarrow} a^i f(\frac{n}{b^i}) \leqslant c^i f(n)$.

   $\sum_{i=0}^{h-1} a^i f\left(\frac{n}{b^i}\right) \leqslant f(n) \sum_{i=0}^{h-1} c^i \leqslant f(n) \sum_{i=0}^{\infty} c^i \overset{6}{=} \dfrac{f(n)}{1-c} = O(f(n)).$

---

[2] $b^d = a$

[3] воспользуемся формулой для геометрической прогрессии

[4] $b^h = n$

[5] i раз применим это условие

[6] бесконечно убывающая геометрическая прогрессия

То есть $T(n) = O(f(n)) + c'n^d = O(f(n))$, поскольку $n^d = O(f(n))$ в силу условия (a).

С другой стороы $T(n) = aT\left(\frac{n}{b}\right) + f(n) \geqslant f(n)$, т. е. $T(n) = \Omega(f(n))$.

Откуда следует утверждение третьего пунтка теоремы.

Важно отметить, что вышеприведенные оценки верны и для соотношений:

$$T(n) = aT\left(\left\lfloor \frac{n}{b} \right\rfloor\right) + f(n),$$

$$T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + f(n).$$

Это справедливо потому что при работе с рекуррентными соотношениями для алгоритмов мы имеем дело только с монотонными функциями $T(n)$ — если задачу можно решить на входе большей длины за время t, то и на входе меньшей длины её тоже можно решить за время t. Осталось только воспользоваться соотношением

$$\forall b \in \mathbb{N}_1 \;\; \forall n \in \mathbb{N} \;\; \exists h \in \mathbb{N} : b^h \leqslant n < b^{h+1},$$

которое берётся из свойств систем счисления. ($\mathbb{N}_1 = \mathbb{N} \setminus \{$, множество натуральных чисел $\mathbb{N}$ начинается с нуля.)

Список литературы

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

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

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

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

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

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

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

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

[Lei96] Tom Leighton. “Notes on Better Master Theorems for Divide-and-Conquer Recurrences”. В: *Lecture notes, MIT*. 1996. URL: https://courses.csail.mit.edu/6.046/spring04/handouts/akrabazzi.pdf.