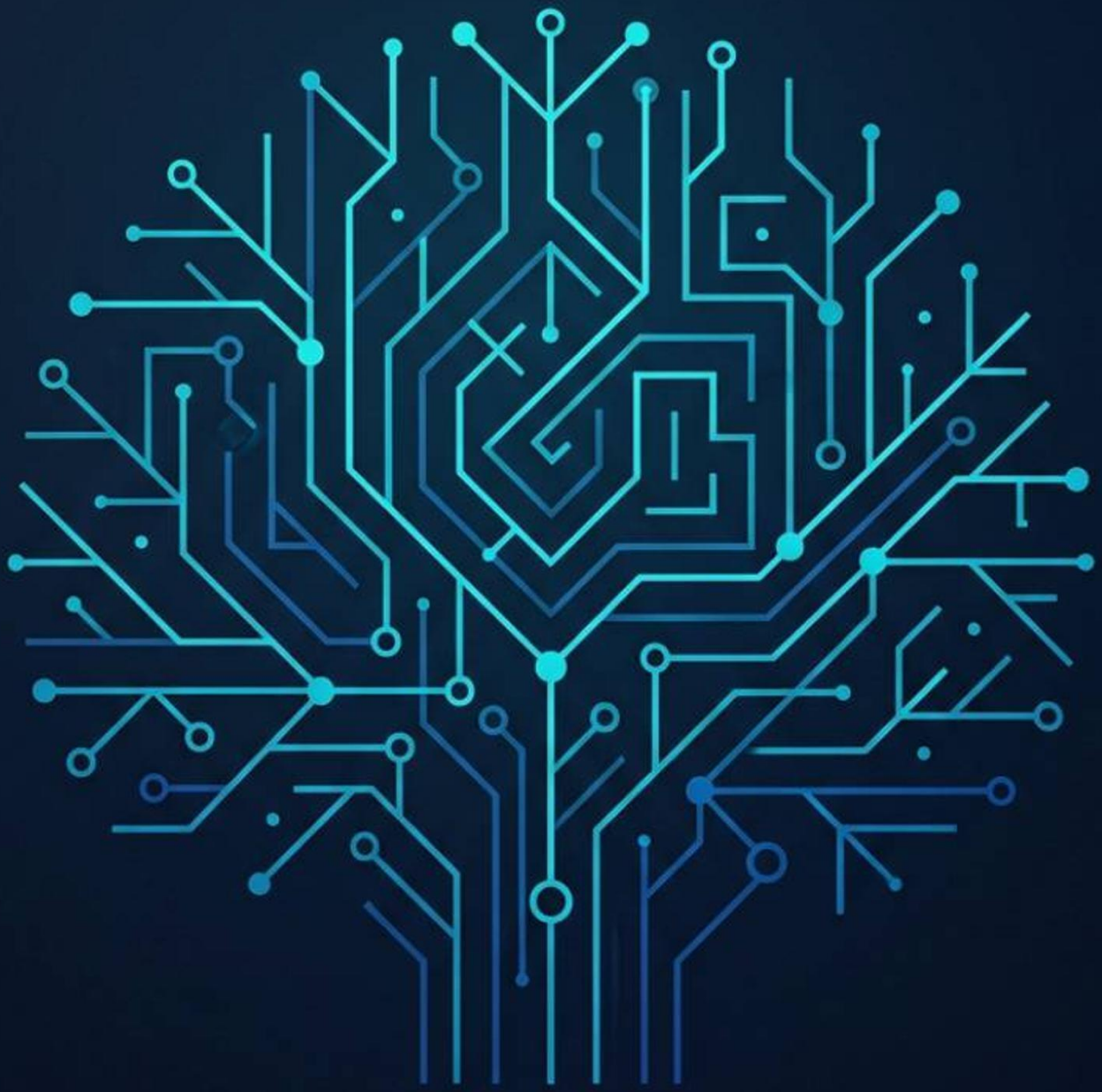

АЛГОРИТМИЧЕСКОЕ МЫШЛЕНИЕ

От структур данных к паттернам решения задач

Максим Байтович

Максим Байтович

Алгоритмическое мышление: От структур данных к паттернам решения задач

«Автор»
2026

Байтович М.

Алгоритмическое мышление: От структур данных к паттернам решения задач / М. Байтович — «Автор», 2026

Алгоритмическое мышление: От структур данных к паттернам решения задач — книга, которая учит видеть за любой алгоритмической задачей знакомый шаблон. Никакой привязки к конкретному языку: только идеи, принципы и ментальные модели. Вы освоите фундаментальные структуры данных (массивы, списки, хеш-таблицы, деревья, кучи, графы), базовые алгоритмы поиска и сортировки, а главное — 20 классических паттернов: два указателя, скользящее окно, быстрый и медленный указатель, BFS и DFS, слияние интервалов, циклическая сортировка, динамическое программирование, жадные алгоритмы и многие другие. Каждый паттерн разбирается на узнаваемых признаках, псевдокоде и примерах с Leetcode. Это не справочник синтаксиса, а тренажёр инженерной интуиции. Для разработчиков, которые готовятся к собеседованиям, участвуют в соревнованиях или просто хотят решать задачи быстрее и элегантнее.

Содержание

Алгоритмическое мышление: От структур данных к паттернам решения задач

Введение: Почему паттерны?

Представьте, что вы учитесь играть в шахматы. Можно заучить, как ходит каждая фигура — это синтаксис языка. Но чтобы выигрывать, нужно знать дебюты, связки, вилки и эндшпили. В программировании всё точно так же.

Большинство разработчиков застревают на алгоритмических собеседованиях или при решении сложных задач не потому, что они плохо знают Python или Java. Они застревают, потому что пытаются изобрести велосипед за тридцать минут, не зная, что задача уже решена десятки раз с помощью определённого паттерна.

Эта книга не учит синтаксису. Она учит узнавать врага в лицо. Мы берём сырую задачу и учимся препарировать её, задавая правильные вопросы.

Данные линейные или иерархические? Нужно перебрать все варианты или можно жадно выбрать лучший? Зависит ли текущий шаг от предыдущих? Ответив на эти вопросы, вы сужаете пространство поиска до двух-трёх паттернов, а дальше — дело техники.

Мы разобьём материал на три логических блока. Инструменты — идеальные формы хранения данных, фундамент, без которого невозможно говорить об алгоритмах. Методы — базовые операции вроде поиска и сортировки, кирпичики, из которых строятся решения. Паттерны — двадцать классических сценариев, покрывающих девяносто процентов задач на Leetcode, от скользящего окна до динамического программирования.

Каждый паттерн мы разбираем по одной схеме: суть идеи в двух-трёх предложениях, как распознать его в незнакомой задаче, ментальная модель или аналогия из реального мира, пошаговый шаблон решения на псевдокоде, типичные ошибки и ловушки, и наконец — несколько конкретных задач с подробным разбором.

К концу книги у вас в голове сформируется дерево решений. Вы увидите задачу — и почти сразу поймёте, к какому паттерну она относится. Это и есть алгоритмическое мышление.

Часть I. Инструментарий: Данные в идеальной форме

Прежде чем применять паттерны, нужно понимать, с каким материалом мы работаем. Каждая структура данных — это не просто способ сложить байтики. Это философия доступа. Выбор правильной структуры часто решает задачу ещё до того, как вы написали первую строчку кода.

Глава 1. Массивы и строки: Линейный порядок

Массив — это непрерывный блок памяти, где каждый элемент сидит в своей ячейке, а ячейки пронумерованы по порядку. Если компьютерная память — это улица, то массив — дом с квартирами ноль, один, два, три.

Главная суперсила массива: зная адрес начала и номер элемента, вы попадаете в него мгновенно, за O(1). Это называется произвольный доступ. Вам не нужно проходить через первые пятьдесят квартир, чтобы попасть в пятьдесят первую.

Массивы бывают статическими и динамическими. Статический имеет фиксированный размер при создании — выделили память под сто элементов, и больше ни байтом. Динамический может расти: когда место заканчивается, он выделяет новый кусок памяти вдвое больше, копирует всё туда, и освобождает старый. Такое копирование случается всё реже по мере роста, и в среднем вставка в конец остаётся O(1). Это называется амортизированная сложность.

Базовые операции и их цена. Доступ по индексу — O(1), это суперсила. Поиск значения — O(N), если массив не отсортирован, приходится проверять каждый элемент. Вставка или удаление в конце — O(1). Вставка или удаление в начале или середине — O(N), потому что нужно сдвинуть всех соседей.

Строка — это, по сути, массив символов. Все операции над массивами применимы и к строкам, но с одной важной оговоркой: в большинстве языков строки иммутабельны. Операция s = s + "!" не меняет старую строку, а создаёт новую. Построение строки в цикле через конкатенацию даёт $O(N^2)$. Всегда используйте StringBuilder или его аналоги для аккумуляции.

Полезный инструмент при работе с массивами — префиксные суммы. Это массив, где каждый элемент хранит сумму всех предыдущих. Строится за O(N): pref[0] = arr[0], затем pref[i] = pref[i-1] + arr[i]. После этого любой запрос «сумма на отрезке от L до R» выполняется за O(1): pref[R] - pref[L-1]. Это не просто трюк, это паттерн мышления — предподсчёт для мгновенных ответов.

Пример. Дан массив цен на акции за несколько дней. Нужно отвечать на множество запросов: «какова средняя цена с дня L по день R?» Предподсчитываем префиксные суммы, и каждый запрос — это одна операция вычитания и деления.

Ещё один инструмент — скользящее окно, но это уже полноценный паттерн, и мы разберём его в отдельной главе.

Сигналы, что нужно использовать массив: данные однотипны и их количество известно или меняется редко, нужен частый доступ по индексу, порядок элементов важен, задача про последовательность или соседей или окно. Сигналы, что массив неудобен: частые вставки или удаления в середину — смотрите на связный список; постоянный поиск «есть ли элемент?» — смотрите на хеш-таблицу; нужен доступ к самому большому или самому маленькому — смотрите на кучу.

Глава 2. Связные списки: Гибкость через указатели

Связный список — это цепочка узлов, разбросанных по памяти. Каждый узел хранит данные и ссылку на следующего соседа. В двусвязном списке — ещё и на предыдущего. Нет сплошного куска памяти. Нет произвольного доступа.

Представьте поезд. Вы можете легко отцепить вагон в середине и вставить новый, просто перецепив сцепки. Но чтобы попасть из локомотива в десятый вагон, вам придётся пройти через все девять предыдущих. Телепорта нет. В этом и заключается философия связного списка: гибкость вставки и удаления ценой потери мгновенного доступа.

Типы списков. Односвязный: каждый узел знает только про следующий, движение только вперёд. Двусвязный: каждый узел знает про следующий и предыдущий, можно двигаться в обе стороны. Кольцевой: последний узел ссылается на первый, нет естественного конца.

Базовые операции и их цена. Доступ по индексу — O(N), это ахиллесова пята. Поиск значения — тоже O(N). Вставка или удаление в начало — O(1), это суперсила. Вставка или удаление в середине — O(1), если узел уже у вас в руках, но поиск этого узла стоит O(N). Вставка или удаление в конце — O(1), если есть хвостовой указатель, иначе O(N).

Главный дзен всех операций над списками — перенаправление указателей. Не нужно двигать данные, только менять адреса в полях next и prev. Удаление узла B из цепочки A стрелка B стрелка C: A.next = C. В двусвязном ещё C.prev = A. Узел B отсоединён. Вставка узла X между A и B: X.next = B, затем A.next = X. Критический порядок действий: если сначала сделать A.next = X, вы потеряете ссылку на B. Поэтому всегда сначала сохраняем хвост, потом рвём связь.

Полезная техника — фиктивный узел, или Dummy Node. Голова списка — особый случай: у неё нет предыдущего узла. Чтобы не писать для неё отдельную логику, создают фиктивный узел перед головой. В конце просто возвращают dummy.next. Это как пришить временный локомотив, чтобы все вагоны обрабатывались одинаково.

Пример. Дано: односвязный список 1 -> 2 -> 3 -> 4 -> 5. Нужно развернуть его. Решение: проходим по списку, на каждом шаге запоминаем следующий узел, перенаправляем текущий на предыдущий, сдвигаем указатели. После цикла бывшая голова стала хвостом, бывший хвост — головой. Результат: 5 -> 4 -> 3 -> 2 -> 1.

Другой пример. Нужно удалить N-й узел с конца списка за один проход. Заводим два указателя, оба на голову. Первый сдвигаем на N шагов вперёд. Затем оба двигаем шаг за шагом, пока первый не дойдёт до конца. Второй окажется ровно на N-м узле с конца. Удаляем его, перецепив ссылку.

Сравнение с массивом. Доступ по индексу: массив O(1), список O(N). Вставка в начало: массив O(N), список O(1). Вставка в конец: оба O(1), но у списка нужен хвостовой указатель. Память: массив компактен, в списке на каждый узел тратится дополнительная память на указатели. Кеш-промахи: массив хранит данные рядом, процессорный кеш работает эффективно; узлы списка разбросаны по памяти, кеш-промахов больше.

Глава 3. Стек и очередь: Дисциплина доступа

Стек и очередь — это не столько структуры данных, сколько дисциплины доступа. Они определяют не то, как данные хранятся, а то, в каком порядке они извлекаются. Реализовать их можно и на массиве, и на связном списке.

Стек работает по принципу LIFO: последним пришёл — первым ушёл. Ментальная модель — стопка тарелок. Вы кладёте тарелку наверх и берёте всегда с верха. Три основные операции: push — положить на верх, pop — снять с верха, peek или top — посмотреть на верхнюю, не снимая. Все за O(1).

Сигналы к применению стека. Классика — сбалансированные скобки. Дана строка из скобок ({[]}), нужно проверить, что каждая открывающая имеет парную закрывающую. Идём по строке: открывающую кладём в стек, закрывающую сравниваем с верхней. Если соответствуют — снимаем верхнюю, иначе — ошибка. В конце стек должен быть пуст.

Другие сигналы: кнопка «Отменить» — история действий это стек; кнопка «Назад» в браузере; обход в глубину, о котором речь пойдёт позже; вычисление арифметических выражений в постфиксной нотации. Отдельно стоит монотонный стек — специальный паттерн для поиска следующего большего или меньшего элемента, но это продвинутая тема.

Очередь работает по принципу FIFO: первым пришёл — первым ушёл. Модель — живая очередь в магазине. Три операции: enqueue — встать в конец, dequeue — уйти из начала, peek — посмотреть на первого. Все за O(1).

Сигналы к применению очереди: обход в ширину — мы разберём его в отдельной главе; очередь задач; буфер; печать документов. В более сложных формах используется двусторонняя очередь, или deque, которая позволяет вставку и удаление с обоих концов и применяется в скользящем окне для хранения кандидатов на максимум или минимум.

Пример. Моделирование касс в супермаркете. Есть несколько касс, у каждой своя очередь. Новый покупатель встаёт в самую короткую очередь. Касса обслуживает покупателя за фиксированное время, затем берёт следующего. Это классическая задача на несколько очередей с событиями.

Глава 4. Бинарное дерево: Иерархия в памяти

Бинарное дерево — это нелинейная иерархическая структура. Каждый узел имеет не более двух детей: левого и правого. Это основа для множества более сложных структур — BST и кучи — и для множества алгоритмов — рекурсия, обходы, поиск.

Терминология. Корень — самый верхний узел. Лист — узел без детей. Родитель и потомок. Поддерево — узел и все его потомки. Высота — длина самого длинного пути от узла до листа. Глубина — длина пути от корня до узла.

Типы бинарных деревьев. Полное, или Full: у каждого узла либо ноль, либо два ребёнка. Совершенное, или Perfect: все листья на одном уровне, у всех внутренних узлов два ребёнка. Завершённое, или Complete: все уровни, кроме, возможно, последнего, заполнены полностью, последний заполняется слева направо. Сбалансированное: для каждого узла разница высот левого и правого поддеревьев не превышает единицу.

Обходы дерева — это три способа увидеть его содержимое, и у каждого своё предназначение. Pre-order: сначала корень, потом левое поддерево, потом правое. Используется для сериализации дерева — сохранения его в файл или передачи по сети, потому что структура восстанавливается однозначно. In-order: сначала левое поддерево, потом корень, потом правое. Для бинарного дерева поиска даёт элементы в строго отсортированном порядке. Post-order: сначала левое, потом правое, потом корень. Используется для удаления дерева: сначала удаляем детей, потом родителя.

Четвёртый способ — level-order, обход по уровням, или BFS. Обрабатываем дерево слой за слоем, слева направо. О нём мы подробно поговорим в главе про поиск в ширину.

Правило выбора обхода. Нужно обработать корень до детей — pre-order. Нужно обработать детей до корня — post-order. Нужен сортированный вывод BST — in-order. Нужен уровень за уровнем — BFS.

Представление в памяти. Ссылочное: каждый узел — объект с полями left и right, указателями на детей. Массив: для завершённого дерева корень в индексе один, левый ребёнок узла i в индексе 2i, правый в 2i+1, родитель в i делить на два. Это основа для кучи, которую мы разберём следующей.

Пример. Дано бинарное дерево. Нужно проверить, симметрично ли оно относительно центра. Решение: два указателя, оба стартуют из корня, но один идёт налево, другой направо. Рекурсивно сравниваем значения и зеркально обходим поддеревья: левый левого с правым правого, левый правого с правым левого.

Другой пример. Найти максимальную глубину дерева. Рекурсивно: глубина равна единица плюс максимум из глубин левого и правого поддеревьев. Базовый случай: пустой узел имеет глубину ноль.

Глава 5. Куча: Машина для экстремумов

Куча — это специализированное бинарное дерево, которое удовлетворяет свойству кучи. В Min-Heap значение в любом узле не больше значений в его детях, и минимум всегда в корне. В Max-Heap — наоборот, максимум в корне.

Куча не является структурой для поиска произвольного элемента. Она — машина для быстрого извлечения экстремума. Почти всегда куча реализуется как массив, используя свойство завершённого бинарного дерева. Корень в индексе ноль, дети узла i в индексах 2i+1 и 2i +2, родитель в индексе (i-1)//2.

Базовые операции. Добавить элемент: кладём в конец массива и просеиваем вверх — меняем с родителем, пока свойство кучи не восстановится. Сложность O(log N). Извлечь корень: забираем корень, на его место ставим последний элемент массива и просеиваем вниз — меняем с меньшим из детей, пока свойство кучи не восстановится. Сложность O(log N). Посмотреть на корень: O(1), это суперсила. Построить кучу из массива: проходим по внутренним узлам снизу вверх и для каждого выполняем просеивание вниз. Сложность O(N), а не O(N log N), потому что большинство узлов находятся близко к листьям и делают мало шагов вниз.

Пример. Дан массив чисел, нужно найти K самых больших элементов. Решение через Min-Heap размера K. Идём по массиву, кладём элементы в кучу. Если размер кучи превысил K, извлекаем минимум. В конце в куче остаются ровно K крупнейших элементов. Сложность O(N log K) вместо O(N log N) при полной сортировке.

Другой пример. Медиана в потоке данных. Числа приходят одно за другим, нужно в любой момент уметь сказать медиану. Решение через две кучи: Max-Heap для левой половины и Min-Heap для правой. Разбираем этот паттерн подробно в отдельной главе.

Что куча не делает. Не ищет произвольный элемент быстро — для этого хеш-таблица. Не хранит отсортированный порядок — для этого BST. Не является заменой массиву для общего доступа.

Глава 6. Хеш-таблицы: Мгновенный доступ

Хеш-таблица — это структура, обеспечивающая мгновенный доступ по ключу. Вы даёте ключ — строку, число, объект — она возвращает значение в среднем за O(1). Это магия, стоящая за словарями, map и object в разных языках.

Как работает магия. Хеш-функция берёт ключ и превращает его в число — индекс в массиве. Одинаковый ключ всегда даёт одинаковый индекс. Но два разных ключа могут дать один индекс — это коллизия. Хеш-таблица умеет с коллизиями жить.

Два основных способа разрешения коллизий. Метод цепочек: каждая ячейка массива — это связный список или дерево всех элементов с одинаковым хешем. При поиске вычисляем индекс и проходим по цепочке, сравнивая ключи. Плюсы: простота и неограниченный размер. Минусы: дополнительная память на указатели.

Открытая адресация: все элементы хранятся прямо в массиве. Если ячейка занята, ищем другую по правилу — линейное пробирование, квадратичное, двойное хеширование. Плюсы: нет накладных расходов на указатели, хорошая локальность данных. Минусы: требуется аккуратное удаление через пометку «удалено», производительность деградирует при заполнении.

Фактор загрузки — отношение количества элементов к размеру массива. Когда он превышает порог, обычно 0.75, таблица увеличивается вдвое, и все элементы перехешируются заново. Это дорого, O(N), но случается редко. Амортизированно вставка остаётся O(1).

Требования к ключам. Ключи должны быть хешируемыми и неизменяемыми, иммутабельными. Числа, строки, кортежи из неизменяемых — можно. Списки, словари — нельзя, потому что изменение списка изменит его хеш, и таблица его не найдёт.

Пример. Дана строка, нужно найти первый неповторяющийся символ. Решение: проходим по строке, считаем частоту каждого символа в хеш-таблице. Затем проходим ещё раз и находим первый символ с частотой один. O(N) времени.

Другой пример. Дано два массива, нужно найти их пересечение — элементы, которые есть в обоих. Решение: кидаем первый массив в хеш-множество. Проходим по второму массиву, если элемент есть в множестве — добавляем в результат и удаляем из множества, чтобы не дублировать. O(N+M) времени.

Сигналы к применению. Найти пару с суммой X — для каждого элемента проверяем, есть ли X минус элемент в таблице. Посчитать частоту слов. Проверить наличие элемента. Кешировать результат — мемоизация в DP. Сгруппировать по признаку — например, анаграммы. Везде, где нужна ассоциация «ключ — значение» и быстрый доступ.

Сравнение с массивом. Массив: ключ — только целое число, индекс, быстрее, но не гибко. Хеш-таблица: ключ — что угодно, чуть медленнее из-за хеширования, но невероятно гибко.

Глава 7. Бинарное дерево поиска: Компас в данных

Бинарное дерево поиска, или BST, — это бинарное дерево с жёстким правилом: для каждого узла все ключи в левом поддереве меньше, все ключи в правом — больше. Это правило как компас, позволяющий на каждом шаге отбрасывать половину дерева, как в бинарном поиске, но с возможностью динамической вставки.

Представьте книгу-игру: если значение меньше текущего — иди налево, если больше — направо. Вы находите элемент за O(log N) в сбалансированном дереве.

Базовые операции. Поиск: идём от корня, поворачивая налево или направо, O(log N) в среднем. Вставка: ищем место, куда должен попасть новый элемент, и вешаем его листом, O(log N). Удаление: три случая. Удаляемый — лист: просто удаляем. Имеет одного ребёнка: заменяем удаляемый на ребёнка. Имеет двух детей: находим преемника — самый левый узел в правом поддереве, или inorder successor — копируем его значение, рекурсивно удаляем преемника.

Проблема: несбалансированность. Если вставлять элементы в порядке возрастания — 1, 2, 3, 4, 5 — BST вырождается в связный список, и поиск становится O(N). Чтобы этого избежать, придуманы самобалансирующиеся варианты. AVL-дерево: строгий баланс, разница высот не больше единицы. Красно-чёрное дерево: менее строгий баланс, но более быстрые вставки и удаления. Используется в стандартных библиотеках — std::map, TreeMap.

Конец ознакомительного фрагмента.