


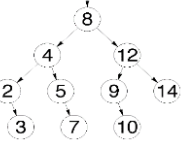


| Алгоритмы | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|--|---|--|----------|---------------------------|----------|---------|------|-------|-----|-------|------|-------|-------|------|--------|----------|--------|-----|-------|-------|----------|--|--|--|--|---------------------------|--|--|--|--|
| Что такое Big O? Как происходит оценка асимптотической сложности алгоритмов? | <p>Big O (О большое / символ Ландау) - математическое обозначение порядка функции для сравнения асимптотического поведения функций.</p> <p>Асимптотика - характер изменения функции при стремлении ее аргумента к определённой точке. Любой алгоритм состоит из неделимых операций процессора(шагов), поэтому нужно измерять время в операциях процессора, вместо секунд.</p> <p>DTIME - количество шагов(операций процессора), необходимых, чтобы алгоритм завершился.</p> <p>Временная сложность обычно оценивается путём подсчёта числа элементарных операций, осуществляемых алгоритмом. Время исполнения одной такой операции при этом берётся константой, то есть асимптотически оценивается как O(1). Сложность алгоритма состоит из двух факторов: временная сложность и сложность по памяти. Временная сложность - функция, представляющая зависимость количество операций процессора, необходимых, чтобы алгоритм завершился, от размера входных данных. Все неделимые операции языка(операции сравнения, арифметические, логические, инициализации и возврата) считаются выполняемыми за 1 операцию процессора, эта погрешность считается приемлемой. При росте N, слагаемые с меньшей скоростью роста всё меньше влияют на значение функции. Поэтому, вне зависимости от констант при слагаемых, слагаемое с большей скоростью роста определяет значение функции. Данное слагаемое называют порядком функции. Пример: $T(N) = 5 * N^2 + 999 * N \dots$ Где $(5 * N^2)$ и $(9999 * N)$ являются слагаемыми функции. Константы(5 и 999) не указываются в рамках нотации Big O, так как не показывают абсолютную сложность алгоритма, так как могут изменяться в зависимости от машины, поэтому сложность равна $O(N^2)$</p> <div><div>Функции по убыванию их порядка</div><div><div><div>$N!$</div><div>2^N</div><div>N^2</div><div>$N * \log_2 N$</div><div>N</div><div>\sqrt{N}</div><div>$\log_2 N$</div><div>1</div></div></div></div> <td><p>Big O нотация нужна для описания сложности алгоритмов.Сложность алгоритма состоит из двух факторов: временная сложность и сложность по памяти.</p><p>Константы откидываются. Нас интересует только часть формулы, которая зависит от размера входных данных. Проще говоря, это само число n, его степени, логарифмы, факториалы и экспоненты, где число находится в степени n.</p><p>Для оценки в качестве N используется бесконечность</p><p>Все константы не влияющие на «бесконечность» будут отброшены. Например, алгоритмы описываемые как $O(2^N)$ и $O(N+100)$ и даже $O(N + \log N)$ - это все равно $O(N)$.</p><p>При этом если алгоритм имеет несколько неизвестных влияющих на сложность например, $O(N+M)$ – M мы не можем отбросить, потому, что</p></td> <td></td> <td></td> <td></td> <td></td> <td></td> | <p>Big O нотация нужна для описания сложности алгоритмов.Сложность алгоритма состоит из двух факторов: временная сложность и сложность по памяти.</p> <p>Константы откидываются. Нас интересует только часть формулы, которая зависит от размера входных данных. Проще говоря, это само число n, его степени, логарифмы, факториалы и экспоненты, где число находится в степени n.</p> <p>Для оценки в качестве N используется бесконечность</p> <p>Все константы не влияющие на «бесконечность» будут отброшены. Например, алгоритмы описываемые как $O(2^N)$ и $O(N+100)$ и даже $O(N + \log N)$ - это все равно $O(N)$.</p> <p>При этом если алгоритм имеет несколько неизвестных влияющих на сложность например, $O(N+M)$ – M мы не можем отбросить, потому, что</p> | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| В порядке возрастания сложности: | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <div><div><div>1. $O(1)$ - константная, чтение по индексу из массива <code>for (let k = 0; k < 100000; k++) {</code></div><div>2. $O(\log(n))$ - логарифмическая, бинарный поиск в отсортированном массиве</div><div>3. $O(\sqrt{n})$ - сублинейная</div><div>4. $O(n)$ - линейная, перебор массива в цикле, два цикла подряд, линейный поиск наименьшего или наибольшего элемента в неотсортированном массиве</div><div>5. $O(n*\log(n))$ - квазилинейная, быстрая сортировка, сортировка слиянием, сортировка кучей</div><div>6. $O(n^2)$ - полиномиальная(квадратичная), вложенный цикл, перебор двумерного массива, сортировка пузырьком, сортировка вставками</div><div>7. $O(2^n)$ - экспоненциальная, алгоритмы разложения на множители целых чисел(последовательность фибоначи)</div><div>8. $O(n!)$ - факториальная, решение задачи коммивояжёра полным перебором</div></div><div>Алгоритм считается приемлемым, если сложность не превышает $O(n*\log(n))$, иначе говнокод.</div></div> | <div><div><div>ПРИМЕР АЛГОРИТМА:</div><div>РАЗМЕР МАССИВА</div></div><div><div>БИНАРНЫЙ ПОИСК</div><div>ПРОСТОЙ ПОИСК</div><div>БЫСТРАЯ СОРТИРОВКА</div><div>СОРТИРОВКА ВЫБОРОМ</div><div>ЗАДАЧА О КОМ-МИВЛЯЖЕРЕ</div></div><div><div></div><table><tr><td>$O(\log n)$</td><td>$O(n)$</td><td>$O(n \log n)$</td><td>$O(n^2)$</td><td>$O(n!)$</td></tr><tr><td>1000</td><td>0.3 с</td><td>1 с</td><td>3.3 с</td><td>10 с</td></tr><tr><td>10000</td><td>0.6 с</td><td>10 с</td><td>66.4 с</td><td>16.6 мин</td></tr><tr><td>100000</td><td>1 с</td><td>100 с</td><td>996 с</td><td>27.7 час</td></tr><tr><td></td><td></td><td></td><td></td><td>1.27x10²³ ЛЕТ</td></tr></table><div>$O(1)$ ПОСТОЯННОЕ ВРЕМЯ (ХЕШ-ТАБЛИЦЫ)</div></div></div> | $O(\log n)$ | $O(n)$ | $O(n \log n)$ | $O(n^2)$ | $O(n!)$ | 1000 | 0.3 с | 1 с | 3.3 с | 10 с | 10000 | 0.6 с | 10 с | 66.4 с | 16.6 мин | 100000 | 1 с | 100 с | 996 с | 27.7 час | | | | | 1.27x10 ²³ ЛЕТ | | | | |
| $O(\log n)$ | $O(n)$ | $O(n \log n)$ | $O(n^2)$ | $O(n!)$ | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1000 | 0.3 с | 1 с | 3.3 с | 10 с | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 10000 | 0.6 с | 10 с | 66.4 с | 16.6 мин | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 100000 | 1 с | 100 с | 996 с | 27.7 час | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | 1.27x10 ²³ ЛЕТ | | | | | | | | | | | | | | | | | | | | | | | | | | |

| | | | |
|--|---|--|------------|
| <p>Что такое рекурсия? Сравните преимущества и недостатки итеративных и рекурсивных алгоритмов. С примерами.</p> | <p>Рекурсия - способ отображения какого-либо процесса внутри самого этого процесса, то есть ситуация, когда процесс является частью самого себя.</p> <p>Рекурсия состоит из базового случая и шага рекурсии. Базовый случай представляет собой самую простую задачу, которая решается за одну итерацию, например, <code>if(n == 0) return 1</code>.</p> <p>В базовом случае обязательно присутствует условие выхода из рекурсии; Смысл рекурсии в движении от исходной задачи к базовому случаю, пошагово уменьшая размер исходной задачи на каждом шаге рекурсии.</p> <p>После того, как будет найден базовый случай, срабатывает условие выхода из рекурсии, и стек рекурсивных вызовов разворачивается в обратном порядке, пересчитывая результат исходной задачи, который основан на результате, найденном в базовом случае.</p> <p>Так работает рекурсивное вычисление факториала:</p> <pre>int factorial(int n) { if(n == 1 n == 0) return 1; // базовый случай с условием выхода else return n * factorial(n - 1); // шаг рекурсии (рекурсивный вызов) }</pre> <p>Или даже так:</p> <pre>return (n == 1 n == 0) ? 1 : n * factorial(n-1);</pre> <p>Рекурсия имеет линейную сложность $O(n)$;</p> <p>Циклы дают лучшую производительность, чем рекурсивные вызовы, поскольку вызовы методов потребляют больше ресурсов, чем исполнение обычных операторов.</p> <p>Циклы гарантируют отсутствие переполнения стека, т.к. не требуется выделения доп. памяти. В случае рекурсии стек вызовов разрастается, и его необходимо просматривать для получения конечного ответа.</p> <p>При использовании головной рекурсии также необходимо принимать во внимание размер стека. Если уровень вложенности много или изменятся, то предпочтительна рекурсия. Если их несколько, то лучше цикл.</p> | <p>factorial(3) внутри себя выполнит следующее:</p> <pre>result = 3 * factorial(2); (рекурсивный вызов)</pre> <p>factorial(2) внутри себя выполнит следующее:</p> <pre>result = 2 * factorial(1); (рекурсивный вызов)</pre> <p>factorial(1) вернет 1 (базис рекурсии)</p> <p>factorial(2) вернет 2 * 1</p> <p>factorial(3) вернет 3 * 2 * 1</p> <pre>public static int getFactorial(int f) { int result = 1; for (int i = 1; i <= f; i++) { result = result * i; } return result; }</pre> | |
| <p>Что такое жадные алгоритмы? Приведите пример.</p> | <p>Жадные алгоритмы являются одной из 3х техник создания алгоритмов, вместе с принципом "Разделяй и властвуй" и динамическим программированием.</p> <p>Жадный алгоритм - это алгоритм, который на каждом шагу совершает локально оптимальные решения, т.е. максимально возможное из допустимых, не учитывая предыдущие или следующие шаги. Последовательность этих локально оптимальных решений приводит (не всегда) к глобально оптимальному решению.</p> <p>Т.е. задача разбивается на подзадачи, в каждой подзадаче делается оптимальное решение и, в итоге, вся задача решается оптимально. При этом важно является ли каждое локальное решение безопасным шагом. Безопасный шаг - приводящий к оптимальному решению.</p> <p>К примеру, алгоритм Дейкстры нахождения кратчайшего пути в графе вполне себе жадный, потому что мы на каждом шагу ищем вершину с наименьшим весом, в которой мы еще не бывали, после чего обновляем значения других вершин. При этом можно доказать, что кратчайшие пути, найденные в вершинах, являются оптимальными.</p> | <p>Заправляться на бензоколонке, которая дальше всех, но не дальше чем 400 км от текущей (оптимальный локальный выбор, который приведет к оптимальному глобальному выбору)</p>  <p>Есть монеты достоинством 1, 5, 10, 20, 25, 50 центов</p> <p>Надо разменять 40 центов этими монетами</p> <p>Ответ: (3 монеты) 40 = 25 + 10 + 5</p> <p>На каждом шаге выбирать наибольшую по достоинству монету - не безопасный ход.</p> <p>Но можно: (2 монеты) 40 = 20 + 20</p> | |
| <p>Расскажите про пузырьковую сортировку.</p> | <p>Будем идти по массиву слева направо. Если текущий элемент больше следующего, меняем их местами. Делаем так, пока массив не будет отсортирован.</p> <p>Асимптотика в худшем и среднем случае – $O(n^2)$, в лучшем случае – $O(n)$ - массив уже отсортирован.</p> | <p>Быстрая сортировка (пример)</p>  | |
| <p>Расскажите про быструю сортировку.</p> | <p>Выберем некоторый опорный элемент (pivot). После этого перекинем все элементы, меньшие его, налево, а большие – направо. Для этого используются дополнительные переменные - значения слева и справа, которые сравниваются с pivot. Рекурсивно вызовемся от каждой из частей, где будет выбран новый pivot. В итоге получим отсортированный массив, так как каждый элемент меньше опорного стоял раньше каждого большего опорного.</p> <p>Асимптотика: $O(n \log(n))$ в среднем и лучшем случае. Наихудшая оценка $O(n^2)$ достигается при неудачном выборе опорного элемента.</p> | <p>Сортировка слиянием (merge sort)</p> <ul style="list-style-type: none"> Массив рекурсивно разбивается пополам до тех пор, пока размер очередного подмассива не станет равен 1 Каждая половина сортируется отдельно Затем выполняется процедура слияния двух упорядоченных подмассивов в один | |
| <p>Расскажите про сортировку слиянием.</p> | <p>Основана на парадигме «разделяй и властвуй». Будем делить массив пополам, пока не получим множество массивов из одного элемента. После чего выполним процедуру слияния: поддерживаем два указателя, один на текущий элемент первой части, второй – на текущий элемент второй части. Из этих двух элементов выбираем минимальный, вставляем в ответ и сдвигаем указатель, соответствующий минимуму. Так сделаем слияния массивов из 1го элемента в массивы по 2 элемента, затем из 2х в 4 и т.д. Слияние работает за $O(n)$, уровней всего $\log(n)$, поэтому асимптотика $O(n \log(n))$.</p> |  | |
| <p>Расскажите про бинарное дерево.</p> | <p>Бинарное дерево - иерархическая структура данных, в которой каждый узел может иметь двух потомков. Как правило, первый называется родительским узлом, а наследники называются левым и правым нодами/узлами. Каждый узел в дереве задаёт поддерево, корнем которого он является. Оба поддерева — левое и правое — тоже являются бинарными деревьями. Ноды, которые не имеют потомков, называются листьями дерева. У всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных самого узла X. У всех узлов правого поддерева произвольного узла X значения ключей данных больше либо равны, нежели значение ключа данных самого узла X. Этим достигается упорядоченная структура данных, то есть всегда отсортированная.</p> <p>Поиск в лучшем случае - $O(\log(n))$, худшем - $O(n)$ - при вырождении в связанный список.</p> |  | <p>102</p> |

| | | |
|--|---|--|
| <p>Расскажите про красно-черное дерево.</p> | <p>Усовершенствованная версия бинарного дерева. Каждый узел в к/ч дереве имеет дополнительное поле - цвет. К/ч дерево отвечает следующим требованиям:</p> <ol style="list-style-type: none"> 1) Узел либо красный, либо черный. 2) Корень - черный. 3) Все листья - черные и не хранят данных. 4) Оба потомка каждого красного узла - черные. 5) Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число черных узлов. Если не одинаковое, то происходит переворот. <p>При добавлении постоянно увеличивающихся/уменьшающихся чисел в бинарное дерево, оно вырождается в связанный список и теряет свои преимущества. Тогда как к/ч дерево может потребовать до двух поворотов для поддержки сбалансированности, чтобы избежать вырождения.</p> <p>При операциях удаления в бинарном дереве для удаляемого узла надо найти замену. К/ч дерево сделает тоже самое, но потребует до трёх поворотов для поддержки сбалансированности.</p> <p>В этом и состоит преимущество.</p> <p>Сложность поиска, вставки и удаления - $O(\log(n))$</p> | |
| <p>Расскажите про линейный и бинарный поиск.</p> | <p>Линейный поиск - сложность $O(n)$, так как все элементы проверяются по очереди.</p> <p>Бинарный поиск - $O(\log(n))$. Массив должен быть отсортирован. Происходит поиск индекса в массиве, содержащего искомое значение.</p> <ol style="list-style-type: none"> 1) Берем значение из середины массива и сравниваем с искомым. Индекс середины считается по формуле $mid = (high + low) / 2$ low - индекс начала левого подмассива, high - индекс конца правого подмассива. 2) Если значение в середине больше искомого, то рассматриваем левый подмассив и $high = middle - 1$ 3) Если меньше, то правый и $low = middle + 1$ 4) Повторяем, пока mid не становится равен искомому элементу или подмассив не станет пустым. <pre> public static int binarySearch(int[] a, int key) { int low = 0; int high = a.length - 1; while (low <= high) { int mid = (low + high)/2; if (key > a[mid]) { low = mid + 1; } else if (key < a[mid]) { high = mid - 1; } else return mid; } return -1; } </pre> | |
| <p>Расскажите про очередь и стек.</p> | <p>Стек реализует принцип «last in - first out», т.е. «последним пришёл - первым вышел». Аналогия из реального мира - это стопка книг на столе (сначала берём верхнюю). В Java есть одноимённый класс Stack. Добавление элементов осуществляется методом push(), а удаление методом pop().</p> <p>Queue - это очередь, которая обычно (но необязательно) строится по принципу FIFO (First-In-First-Out) - соответственно извлечение элемента осуществляется с начала очереди, вставка элемента - в конец очереди. (Добавляется в конец очереди используется из начала)</p> <p>Хотя этот принцип нарушает, к примеру PriorityQueue, использующая «natural ordering» или переданный Comparator при вставке нового элемента.</p> <p>Deque (Double Ended Queue) расширяет Queue и согласно документации это линейная коллекция, поддерживающая вставку/извлечение элементов с обоих концов. Помимо этого реализации интерфейса Deque могут строиться по принципу FIFO, либо LIFO.</p> <p>Количество методов удваивается. Пример:</p> <pre> addFirst(E e); addLast(E e); </pre> <p>Реализации и Deque, и Queue обычно не переопределяют методы equals() и hashCode(), вместо этого используются наследованные методы класса Object, основанные на сравнении ссылок.</p> | <pre> Stack<Integer> stack = new Stack<>(); stack.push(1); stack.push(2); stack.push(3); while (!stack.empty()) { System.out.println(stack.pop()); } </pre> <p>Внутри наших методов вызываются методы класса <code>Stack</code>:</p> <ul style="list-style-type: none"> • push() — добавляет элемент на верх стека. Когда мы отправляем карту в сброс, она ложится поверх сброшенных ранее карт; • pop() — удаляет верхний элемент из стека и возвращает его. Этот метод идеально подходит для реализации механики "игрок берет карту" • peek() — возвращает верхний элемент стека, но не удаляет его из стека |

