Professor Hans Noodles

41 уровень

29.07.2019 34188 12

Сложность алгоритмов

Статья из группы Java Developer

43414 участников

Вы в группе

Привет!

Сегодняшняя лекция будет немного отличаться от остальных. Отличаться она будет тем, что имеет лишь косвенное отношение к Java.

Тем не менее, эта тема очень важна для каждого программиста. Мы поговорим об **алгоритмах**.

Что такое алгоритм?

Говоря простым языком, **это некоторая последовательность действий, которые необходимо совершить для достижения нужного результата**.

Мы часто используем алгоритмы в повседневной жизни.

Например, каждое утро перед тобой стоит задача: прийти на учебу или работу, и быть при этом:

- Одетым
- Чистым
- Сытым

Какой **алгоритм** позволит тебе добиться этого результата?

НАЧАТЬ ОБУЧЕНИЕ

3. Приготовить завтрак, сварить кофе/заварить чай.
4. Поесть.
5. Если не погладил одежду с вечера — погладить.
6. Одеться.
7. Выйти из дома.

Эта последовательность действий точно позволит тебе получить необходимый результат.

В программировании вся суть нашей работы заключается в постоянном решении задач.

Значительную часть этих задач можно выполнить, используя уже известные алгоритмы.

К примеру, перед тобой стоит задача: **отсортировать список из 100 имен в массиве**.

Задача это довольно проста, но решить ее можно разными способами.

Вот один из вариантов решения:

Алгоритм сортировки имен по алфавиту:

1. Купить или скачать в Интернете “Словарь русских личных имен” 1966 года издания.
2. Находить каждое имя из нашего списка в этом словаре.
3. Записывать на бумажку, на какой странице словаря находится имя.
4. Расставить имена по порядку, используя записи на бумажке.

Позволит ли такая последовательность действий решить нашу задачу? Да, вполне позволит.

Будет ли это решение **эффективным**? Вряд ли.

Здесь мы подошли к еще одному очень важному свойству алгоритмов — их **эффективности**.

Решить задачу можно разными способами. Но и в программировании, и в обычной жизни мы выбираем способ, который будет наиболее эффективным.

Если твоя задача — сделать бутерброд со сливочным маслом, ты, конечно, можешь начать с того, что посеешь пшеницу и подоишь корову.

Но это будет **неэффективное** решение — оно займет очень много времени и будет стоить много денег.

Для решения твоей простой задачи хлеб и масло можно просто купить. А алгоритм с пшеницей и коровой хоть и позволяет решить задачу, слишком сложный, чтобы применять его на практике.

Для оценки сложности алгоритмов в программировании создали специальное обозначение под названием **Big-O (“большая О”)**.

Big-O позволяет оценить, насколько время выполнения алгоритма зависит от переданных в него данных.

Давай рассмотрим самый простой пример — передачу данных.

Представь, что тебе нужно передать некоторую информацию в виде файла на большое расстояние (например, 5000 километров).

Какой алгоритм будет наиболее эффективным?

Это зависит от тех данных, с которыми ему предстоит работать. К примеру, у нас есть аудиофайл размером 10 мегабайт.

В этом случае, самым эффективным алгоритмом будет передать файл через Интернет. Это займет максимум пару минут!

Итак, давай еще раз озвучим наш алгоритм:

“Если требуется передать информацию в виде файлов на расстояние 5000 километров, нужно использовать передачу данных через Интернет”.

Отлично. Теперь давай проанализируем его.

Решает ли он нашу задачу? В общем-то да, вполне решает.

А вот что можно сказать насчет его сложности?

Хм, а вот тут уже все интереснее. Дело в том, что наш алгоритм очень сильно зависит от входящих данных, а именно — от размера файлов.

Сейчас у нас 10 мегабайт, и все в порядке.

А что, если нам нужно будет передать 500 мегабайт?
20 гигабайт?
500 терабайт?
30 петабайт?

Перестанет ли наш алгоритм работать? Нет, все эти объемы данных все равно можно передать.

Станет ли он выполняться дольше? Да, станет!

Теперь нам известна важная особенность нашего алгоритма: **чем больше размер данных для передачи, тем дольше времени займет выполнение алгоритма**.

Но нам хотелось бы более точно понимать, как выглядит эта зависимость (между размером данных и временем на их передачу).

В нашем случае сложность алгоритма будет **линейной**.

“Линейная” означает, что при увеличении объема данных время на их передачу вырастет примерно пропорционально. Если данных станет в 2 раза больше, и времени на их передачу понадобится в 2 раза больше. Если данных станет больше в 10 раз, и время передачи увеличится в 10 раз.

Используя обозначение Big-O, сложность нашего алгоритма определяется как **O(N)**. Это обозначение лучше всего запомнить

Обрати внимание: мы вообще не говорим здесь о разных “переменных” вещах: скорости интернета, мощности нашего компьютера и так далее. При оценке сложности алгоритма в этом просто нет смысла — мы в любом случае не можем это контролировать. **Big-O оценивает именно сам алгоритм, независимо от “окружающей среды” в которой ему придется работать.**

Продолжим работать с нашим примером. Допустим, в итоге выяснилось, что размер файлов для передачи составляет 800 терабайт.

Если мы будем передавать их через Интернет, задача, конечно, будет решена. Есть только одна проблема: передача по стандартному современному каналу (со скоростью 100 мегабит в секунду), который используется дома у большинства из нас, займет примерно 708 дней.

Почти 2 года! :O

Так, наш алгоритм тут явно не подходит. Нужно какое-то другое решение!

Неожиданно на помощь к нам приходит IT-гигант — компания Amazon! Ее сервис Amazon Snowmobile позволяет загрузить большой объем данных в передвижные хранилища и доставить по нужному адресу на грузовике!

Итак, у нас есть новый алгоритм!

“Если требуется передать информацию в виде файлов на расстояние 5000 километров и этот процесс займет больше 14 дней при передаче через Интернет, нужно использовать перевозку данных на грузовике Amazon”.

Цифра 14 дней здесь выбрана случайно: допустим, это максимальный срок, который мы можем себе позволить.

Давай проанализируем наш алгоритм. Что насчет скорости? Даже если грузовик поедет со скоростью всего 50 км/ч, он преодолеет 5000 километров всего за 100 часов. Это чуть больше четырех дней! Это намного лучше, чем вариант с передачей по интернету.

А что со сложностью этого алгоритма? Будет ли она тоже линейной, O(N)?

Нет, не будет. Ведь грузовику без разницы, как сильно ты его нагрузишь — он все равно поедет примерно с одной и той же скоростью и приедет в срок.
Будет ли у нас 800 терабайт, или в 10 раз больше данных, грузовик все равно доедет до места за 5 дней.

Иными словами, у алгоритма доставки данных через грузовик **постоянная сложность**. “Постоянная” означает, что она не зависит от передаваемых в алгоритм данных.

дней.

При использовании Big-O постоянная сложность обозначается как **O(1)**.

Раз уж мы познакомились с **O(N)** и **O(1)**, давай теперь рассмотрим более “программистские” примеры :)

Допустим, тебе дан массив из 100 чисел, и задача — вывести в консоль каждое из них.

Ты пишешь обычный цикл `for`, который выполняет эту задачу

```
int[] numbers = new int[100];
// ..заполняем массив числами

for (int i: numbers) {
   System.out.println(i);
}
```

Какая сложность у написанного алгоритма? **Линейная, O(N).**

Число действий, которые должна совершить программа, зависит от того, сколько именно чисел в нее передали.

Если в массиве будет 100 чисел, действий (выводов на экран) будет 100. Если чисел в массиве будет 10000, нужно будет совершить 10000 действий.

Можно ли улучшить наш алгоритм? Нет.

Нам в любом случае придется совершить **N проходов по массиву** и выполнить N выводов в консоль.

Рассмотрим другой пример.

```
public static void main(String[] args) {

   LinkedList<Integer> numbers = new LinkedList<>();
   numbers.add(0, 20202);
   numbers.add(0, 123);
   numbers.add(0, 8283);
}
```

У нас есть пустой `LinkedList`, в который мы вставляем несколько чисел. Нам нужно оценить сложность алгоритма вставки одного числа в `LinkedList` в нашем примере, и как она зависит от числа элементов, находящихся в списке.

Ответом будет **O(1) — постоянная сложность**. Почему?

Обрати внимание: каждый раз мы вставляем число в начало списка. К тому же, как ты помнишь, при вставке числа в `LinkedList` элементы никуда не сдвигаются — происходит переопределение ссылок (если вдруг забыл, как работает LinkedList, загляни в одну из наших старых лекций).

Если сейчас первое число в нашем списке — число `x`, а мы вставляем в начало списка число y, все, что для этого нужно:

```
x.previous  = y;
y.previous = null;
y.next = x;
```

Для этого переопределения ссылок **нам неважно, сколько чисел сейчас в** `LinkedList` — хоть одно, хоть миллиард.

Сложность алгоритма будет постоянной — O(1).

Логарифмическая сложность

Без паники! :)

Если при слове “логарифмический” тебе захотелось закрыть лекцию и не читать дальше - подожди пару минут. Никаких математических сложностей здесь не будет (таких объяснений полно и в других местах), а все примеры разберем “на пальцах”.

Представь, что твоя задача — найти одно конкретное число в массиве из 100 чисел. Точнее, проверить, есть ли оно там вообще.

Как только нужное число найдено, поиск нужно прекратить, а в консоль вывести запись “Нужное число обнаружено! Его индекс в массиве = ...”

Как бы ты решил такую задачу?

Здесь решение очевидно: нужно перебрать элементы массива по очереди начиная с первого (или с последнего) и проверять, совпадает ли текущее число с искомым.

Соответственно, количество действий прямо зависит от числа элементов в массиве.

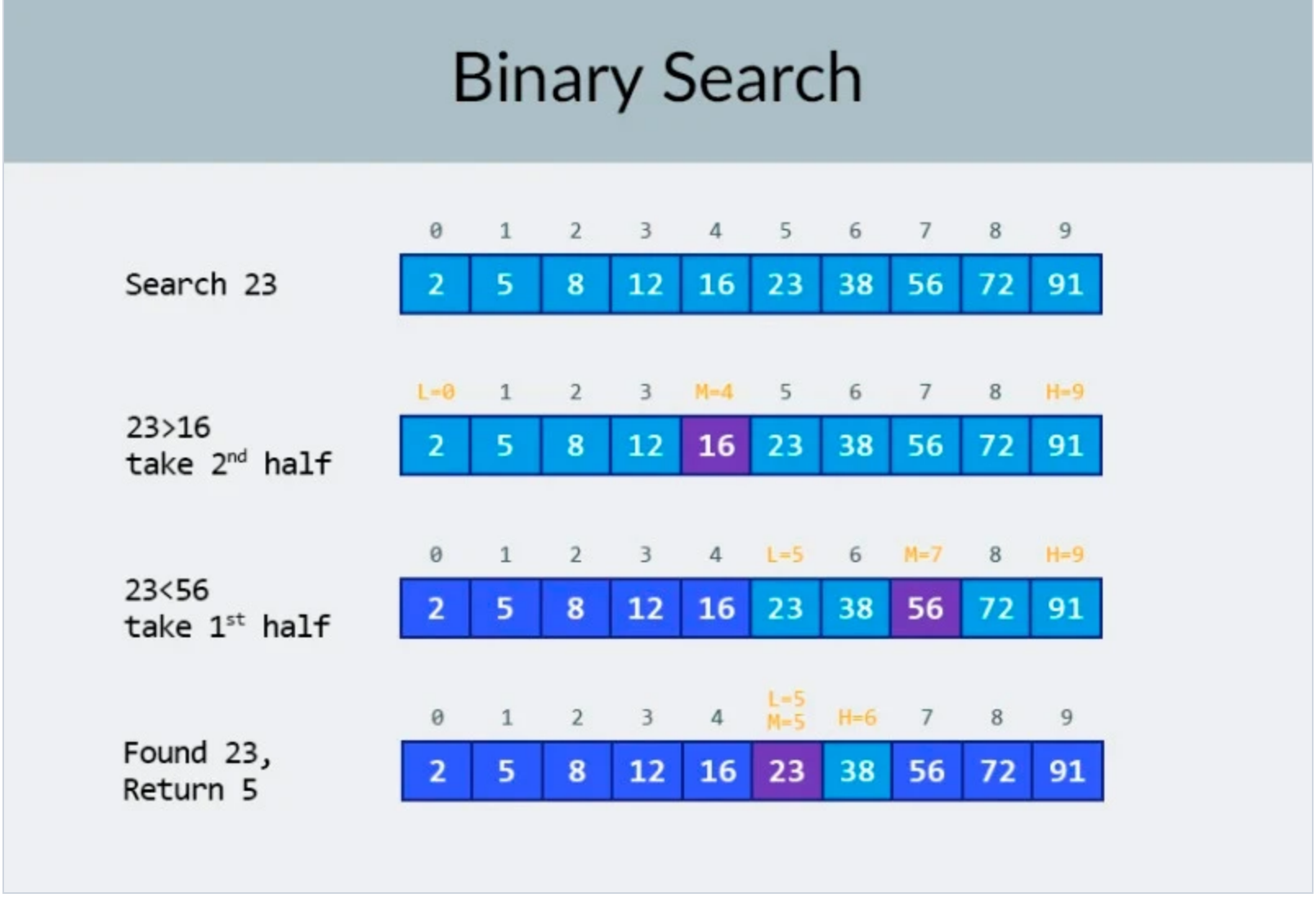
Если у нас 100 чисел, значит, нам нужно 100 раз перейти к следующему элементу и 100 раз проверить число на совпадение. Если чисел будет 1000, значит и шагов-проверок будет 1000.

Это очевидно линейная сложность, **O(N)**.

А теперь мы добавим в наш пример одно уточнение: **массив, в котором тебе нужно найти число, отсортирован по возрастанию**.

Меняет ли это что-то для нашей задачи? Мы по-прежнему можем искать нужное число перебором.

Но вместо этого мы можем использовать известный **алгоритм двоичного поиска**.



В верхнем ряду на изображении мы видим отсортированный массив. В нем нам необходимо найти число 23.

Вместо того, чтобы перебирать числа, мы просто делим массив на 2 части и проверяем среднее число в массиве.

Находим число, которое располагается в ячейке 4 и проверяем его (второй ряд на картинке).

Это число равно 16, а мы ищем 23. Текущее число меньше. Что это означает? Что **все предыдущие числа (которые расположены до числа 16) можно не проверять**: они точно будут меньше того, которое мы ищем, ведь наш массив отсортирован!

Продолжим поиск среди оставшихся 5 элементов.

Обрати внимание: мы сделали всего одну проверку, но уже отмели половину возможных вариантов.

У нас осталось всего 5 элементов. Мы повторим наш шаг — снова разделим оставшийся массив на 2 и снова возьмем средний элемент (строка 3 на рисунке). Это число 56, и оно больше того, которое мы ищем.

Что это означает? Что мы отметаем еще 3 варианта — само число 56, и два числа после него (они точно больше 23, ведь массив отсортирован).

У нас осталось всего 2 числа для проверки (последний ряд на рисунке) — числа с индексами массива 5 и 6.
Проверяем первое из них, и это то что мы искали — число 23! Его индекс = 5!

Давай рассмотрим результаты работы нашего алгоритма, а потом разберемся с его сложностью. (Кстати, теперь ты понимаешь, почему его называют двоичным: его суть заключается в постоянном делении данных на 2).

Результат впечатляет! Если бы мы искали нужное число линейным поиском, нам понадобилось бы 10 проверок, а с двоичным поиском мы уложились в 3!
В худшем случае их было бы 4, если бы на последнем шаге нужным нам числом оказалось второе, а не первое.

А что с его сложностью? Это очень интересный момент :)

Алгоритм двоичного поиска гораздо меньше зависит от числа элементов в массиве, чем алгоритм линейного поиска (то есть, простого перебора).

НАЧАТЬ ОБУЧЕНИЕ

Разница в 2,5 раза.

Но для массива в **1000 элементов** линейному поиску понадобится 1000 проверок, а двоичному — **всего 10**! Разница уже в 100 раз!

Обрати внимание: число элементов в массиве увеличилось в 100 раз (с 10 до 1000), а количество необходимых проверок для двоичного поиска увеличилось всего в 2,5 раза — с 4 до 10.

Если мы дойдем до **10000 элементов**, разница будет еще более впечатляющей: 10000 проверок для линейного поиска, и **всего 14 проверок** для двоичного.

И снова: число элементов увеличилось в 1000 раз (с 10 до 10000), а число проверок увеличилось всего в 3,5 раза (с 4 до 14).

Сложность алгоритма двоичного поиска логарифмическая, или,если использовать обозначения Big-O, — **O(log n)**. Почему она так называется?

Логарифм — это такая штуковина, обратная возведению в степень. Двоичный логарифм использует для подсчета степени числа 2.

Вот, например, у нас есть 10000 элементов, которые нам надо перебрать двоичным поиском.

Array Size	Linear — N	Binary — Log_2N
10	10	4
50	50	6
100	100	7
500	500	9
1000	1000	10
2000	2000	11
3000	3000	12
4000	4000	12
5000	5000	13
6000	6000	13
7000	7000	13
8000	8000	13
9000	9000	14
10000	10000	14

Сейчас у тебя есть картинка перед глазами, и ты знаешь что для этого нужно максимум 14 проверок. Но что если картинки перед глазами не будет, а тебе нужно посчитать точное число необходимых проверок?

Достаточно ответить на простой вопрос: **в какую степень надо возвести число 2, чтобы полученный результат был >= числу проверяемых элементов?**

Для 10000 это будет 14 степень.

2 в 13 степени — это слишком мало (8192)

А вот **2 в 14 степени = 16384**, это число удовлетворяет нашему условию (оно >= числу элементов в массиве).

Мы нашли логарифм — 14. Столько проверок нам и нужно! :)

НАЧАТЬ ОБУЧЕНИЕ

Но знать ее очень важно: на многих собеседованиях ты получишь алгоритмические задачи.

Для теории я могу порекомендовать тебе несколько книг.

Начать можно с “Грокаем алгоритмы”: хотя примеры в книге написаны на Python, язык книги и примеры очень просты. Лучший вариант для новичка, к тому же она небольшая по объему.

Из чтения посерьезней — книги Роберта Лафоре и Роберта Седжвика. Обе написаны на Java, что сделает изучение для тебя немного попроще. Ведь ты неплохо знаком с этим языком! :)

Для учеников с хорошей математической подготовкой лучшим вариантом будет книга Томаса Кормена.

Но одной теорией сыт не будешь!

“Знать” != “уметь”

Практиковать решения задач на алгоритмы можно на HackerRank и Leetcode. Задачки оттуда частенько используют даже на собеседованиях в Google и Facebook, так что скучно тебе точно не будет :)

Для закрепления материала лекции, советую посмотреть отличное видео про Big-O на YouTube.

но можно было бы и больше информации с удовольствием прочитать

Ответить +3

darimmiKto Уровень 1, Russian Federation 2 июня, 14:13

Представь, ты инженер в ракето-строительной компании и тебе сказали написать алгоритм для двигателей, но ты не понимаешь как работают алгоритмы. Допустим ты сделал свою задачу, но кое-как, этот алгоритм оказался линейным, то есть он перебирает все данные и поэтому нагрузка на двигатели будет большая и ракета может попросту взорваться.

Ответить 0

Pig Man Главная свинья в **Свинарнике** 15 февраля 2021, 19:13

"*Говоря простым языком, это некоторая последовательность действий, которые необходимо совершить для достижения нужного результата*" - причем тут простой язык? Это и есть определение алгоритма, другого не дать. Не знаю, зачем докопался, просто скучно

Ответить +1

funbiscuit Уровень 41, Россия 20 января 2021, 16:58

Если при слове “логарифмический” тебе захотелось закрыть лекцию и не читать дальше, то возможно тебе не стоит быть программистом.
Не представляю себе квалифицированного программиста без математики.

Ответить 0

Pig Man Главная свинья в **Свинарнике** 15 февраля 2021, 19:23

Нужна ли математика программисту

Ответить +3

Дима Уровень 41, Хабаровск, Россия 23 декабря 2020, 08:33

хорошая лекция

Ответить +3

Александр Уровень 16, Ангарск, Россия 6 августа 2019, 09:16

23 < 56 :D

Ответить 0