GeekBrains

Лекция 1. Алгоритмы, сложность алгоритмов

```java
public static List<Integer> findSimpleNumbers(int max) {
    List<Integer> result = new ArrayList<>();
    for (int i = 1; i <= max; i++) {
        boolean simple = true;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                simple = false;
            }
        }
        if (simple) {
            result.add(i);
        }
    }
```

Оглавление

```java
public static List<Integer> findSimpleNumbers(int max) {
    List<Integer> result = new ArrayList<>();
    for (int i = 1; i <= max; i++) {
        boolean simple = true;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                simple = false;
            }
        }
        if (simple) {
            result.add(i);
        }
    }
```

Цель лекции:

- Узнать, что такое алгоритм, разобрать представление в виде блок-схемы
- Изучить, что такое сложность алгоритма в нотации О
- Разобраться различные типы сложности алгоритма, сравнить графики роста сложности

План лекции:

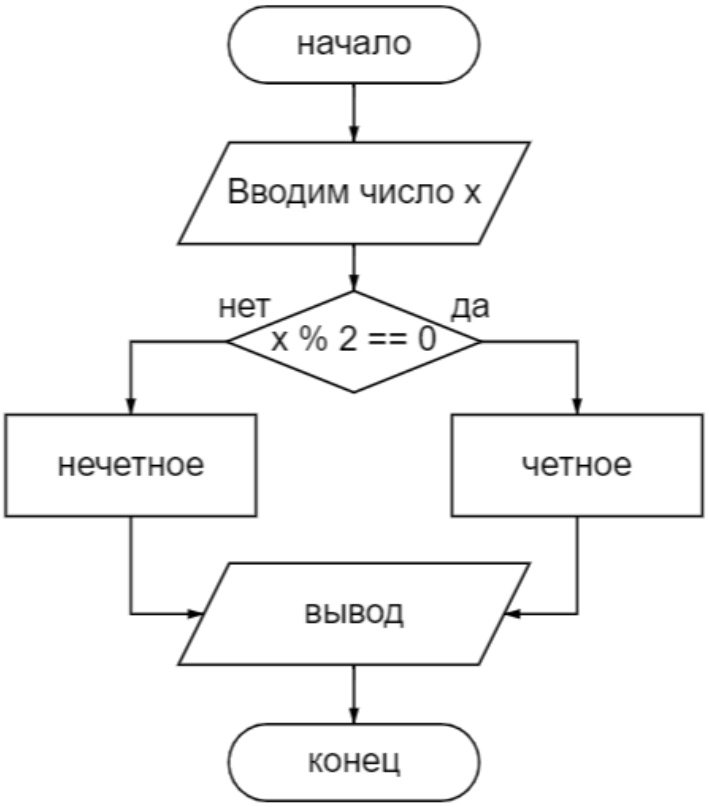
- Знакомство с преподавателем
- Определение алгоритма, для чего нужен, где используется
- Разбор графического представления алгоритма в виде блок-схемы
- Познакомиться с нотацией О, разобрать особенности использования
- Разобрать примеры линейной сложности алгоритма
- Разобрать примеры квадратичной сложности алгоритма
- Разобрать пример экспоненциальной сложности алгоритма
- Разобрать пример квадратичной сложности для рекурсивных алгоритмов
- Привести примеры сложения и перемножения сложности, а также правила сокращения множителей

Алгоритмы

Алгоритмом – это описание порядка действий для решения определённой задачи. Из этого следует, что любая четко определенная последовательность действий, которая способна привести нас к нужному результату – всегда является алгоритмом. Само по себе понятие алгоритма применяется не только в разработке программного обеспечения и появилось оно гораздо раньше, чем компьютеры. На самом деле все люди постоянно используют алгоритмы в своей повседневной жизни. Нас постоянно окружает огромное количество задач, которые необходимо также регулярно решать. Например, определяя, что нужно написать в слове **–тся** или **–ться**, мы используем определенное правило, которое дает явный ответ на

```java
public static List<Integer> findSimpleNumbers(int max) {
    List<Integer> result = new ArrayList<>();
    for (int i = 1; i <= max; i++) {
        boolean simple = true;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                simple = false;
            }
        }
        if (simple) {
            result.add(i);
        }
    }
```

поставленный вопрос. Или переходя дорогу мы смотрит по сторонам, при этом, чаще всего, сначала налево, потом направо. Мы не задумываемся об этом постоянно, но наше поведение и принцип принятия решений во многом основано именно на различных усвоенных нами алгоритмах действий. Любой такой алгоритм можно представить в виде блок-схемы. Например, алгоритм определения является ли число четным.



Как мы видим, у любого алгоритма есть начальная точка и конечные точки, а вот все остальное формирует в зависимости от требуемых действий. Например, в приведенной блок-схеме мы видим, что в начале алгоритма происходит ввод числа,

```java
public static List<Integer> findSimpleNumbers(int max) {
    List<Integer> result = new ArrayList<>();
    for (int i = 1; i <= max; i++) {
        boolean simple = true;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                simple = false;
            }
        }
        if (simple) {
            result.add(i);
        }
    }
```

которое мы и будем анализировать. Далее мы попадаем в блок логического условия, которое звучит как *«равен ли остаток от деления на 2 нулю»*, и, если условие выполняется – мы попадаем в блок результата, где явно задается, что число является четным. В обратной ситуации мы пойдем по левой ветке нашей блок схемы и запишем результат вычисления, как нечетной число. Не зависимо от того, пошли мы по правой или левой ветке, в итоге мы попадем в блок вывода результата и закончим наш алгоритм.

Как мы видим, имея четко определенную последовательность действий, мы можем ее записать и представить в графическом виде. Почему это важно? Когда мы пишем какую-либо инструкцию для человека мы можем пропустить часть информации или оставить нечеткие инструкции, если это не критично. В итоге, человек сможет интерпретировать полученные данные, даже если они оформлены размыто и исполнить инструкцию. Так же человек способен самостоятельно принимать решения по корректировке процесса на основе новых вводных, полученных в процессе исполнения инструкции. Но компьютер не является человеком и не умеет свободно интерпретировать информацию. Формально, компьютер можно назвать полным идиотом, но с феноменальной памятью. Для него важно получить полностью исчерпывающий набор шагов, который он сможет исполнять ровно так, как было описано в его инструкции и упадет с ошибкой в случае не предусмотренного поведения. Именно поэтому для любого разработчика критично важно уметь правильно разбивать процесс решения задачи на четко определенные шаги, чтобы компьютер мог их исполнять.

Где же в программировании мы встречаем алгоритмы? На самом деле везде. Любой метод нашего приложения – это определенный алгоритм действий. Большой или маленький, сложный или простой, с множеством условий или линейный. Одни методы могут вызывать другие методы, т.е. внутри одного алгоритма может оказаться алгоритм меньшего размера. Возвращаясь блок-схеме с проверкой числа на четность, мы можем разбить шаг вычисления остатка от деления на собственный алгоритм с собственной блок-схемой – как именно для компьютера происходит деление и получение остатка? Из этого можно сделать вывод, что

```java
public static List<Integer> findSimpleNumbers(int max) {
    List<Integer> result = new ArrayList<>();
    for (int i = 1; i <= max; i++) {
        boolean simple = true;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                simple = false;
            }
        }
        if (simple) {
            result.add(i);
        }
    }
```

любая программа сама по себе также является алгоритмом, т.к. в ней четко определен порядок действий от момента старта до завершения. А все действия также является алгоритмами, но меньшего размера.

На текущий момент нам уже не требуется визуализировать каждый из наших алгоритмов в виде блок-схемы, но вместо нее выступает языки программирования, которыми можно описать требуемые действия в текстовом виде. А от разработчика требуется умение разбить и записать процесс решения поставленной задачи четкими и понятными для компьютера шагами.

Оценка сложности алгоритмов

Различные алгоритмы требуют различного количества шагов. Какие-то требуют всего несколько шагов, чтобы получить результат, а какие-то миллионы операций, прежде чем алгоритм подойдет к концу.

Также алгоритмы различаются по количеству памяти, которое они потребляют для своего выполнения. Где-то достаточно одной переменной, которая будет изменяться на каждом шаге, а где-то потребуется загрузить в память гигабайты информации, чтобы получить результат.

Несмотря на то, что прогресс ушел далеко вперед и современные процессоры умеют обрабатывать миллиарды операций в секунду, а оперативная память давно исчисляется гигабайтами и легендарная фраза, приписываемая Биллу Гейтсу, что «640кб должно хватить каждому», может вызвать лишь улыбку, уметь оценивать алгоритм по ресурсам, которые ему необходимы для исполнения по-прежнему необходимо.

Чаще всего, когда речь идет про оценку сложности алгоритма, имеется ввиду оценка числа операций относительно размера входящих данных. Но это не единственный показатель, по которому можно оценить алгоритм и определить его пригодность для той или иной ситуации. Также не стоит забывать про оценку затрат оперативной памяти.

Несмотря на то, что оценка памяти производится гораздо реже, есть ощутимое количество операций, где она может играть существенное значение. Например,

```java
public static List<Integer> findSimpleNumbers(int max) {
    List<Integer> result = new ArrayList<>();
    for (int i = 1; i <= max; i++) {
        boolean simple = true;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                simple = false;
            }
        }
        if (simple) {
            result.add(i);
        }
    }
```

если перед нами стоит задача вычислить кратчайший путь между двумя точками на карте города. Классически такая задача решается через поиск кратчайшего пути в графе, но требует построение и вычисление этого графа для каждого из запросов. Этот подход требует регулярных вычислений, но потребуется небольшое количество памяти при использовании. При этом, если потребность в количестве таких вычислений растет, становится рациональным сохранять результаты вычислений и переиспользовать их по мере необходимости. Если на нашей карте количество точек небольшое, то это так же не займет много памяти, например, при наличии 10 точек, нам потребуется хранить результаты в таблице 10 x 10 (работает по принципу таблицы умножения). А если таких точек несколько миллионов, то и таблица получится очень большой и будет занимать много памяти, но и поиск по таблице будет гораздо быстрее чем вычисление значения заново.

Также в качестве примера алгоритма, зависящего от объема памяти, можно взять работу с большими файлами. Например, когда нам необходимо отсортировать данные, хранящиеся в файле, мы можем воспользоваться следующим алгоритмом: Прочитать файл в память -> отсортировать их -> сохранить данные обратно в файл. Но если наш файл весит, например, несколько десятков гигабайт, то загрузить его в память может быть большой проблемой – как правило размер хранилища у компьютера в разы превышает размер оперативной памяти. В такой ситуации указанный выше алгоритм сортировки не пройдет по оценке потребления памяти, т.к. компьютер просто не сможет его исполнить из-за нехватки ресурсов.

Но все же, чаще всегда оценку сложности алгоритма делают именно по оценке числа операций относительно размера входящих данных. Давайте разберем, что же это за оценка.

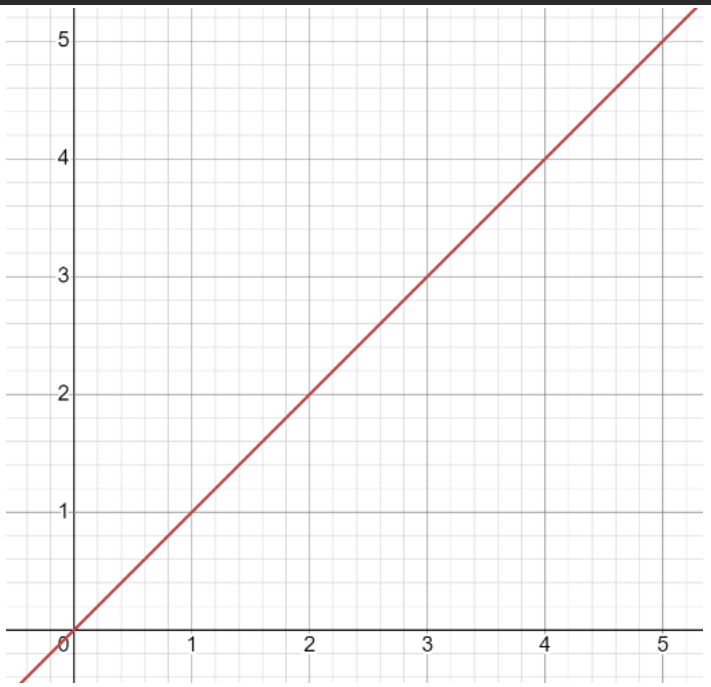
Возьмем для примера задачу по вычислению допустимых делителей для числа n. Алгоритм вычисления весьма простой – нам нужно попробовать разделить n на каждое из чисел последовательности от 1 до n. Если остаток от деления числа n на некоторое число k равняется нулю, чисто k является делителем числа n.

```java
public static List<Integer> findSimpleNumbers(int max) {
    List<Integer> result = new ArrayList<>();
    for (int i = 1; i <= max; i++) {
        boolean simple = true;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                simple = false;
            }
        }
        if (simple) {
            result.add(i);
        }
    }
```

```java
public static List<Integer> availableDivider(int number) {
    List<Integer> result = new ArrayList<>();
    for (int i = 1; i <= number; i++) {
        if (number % i == 0) {
            result.add(i);
        }
    }
    return result;
}
```

Очевидно, что количество операций, которое нужно совершить для вычисления всех доступных делителей равно числу n. Чем больше будет число n, тем больше потребуется сделать вычислений, при этом рост количества операций будет расти линейно относительно роста самого числа. Такую зависимость можно представить в виде линейного графика

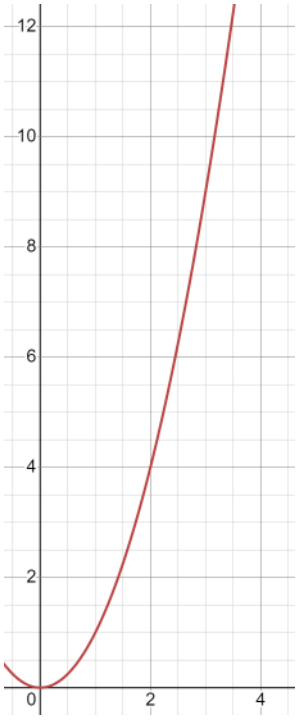
```java
public static List<Integer> findSimpleNumbers(int max) {
    List<Integer> result = new ArrayList<>();
    for (int i = 1; i <= max; i++) {
        boolean simple = true;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                simple = false;
            }
        }
        if (simple) {
            result.add(i);
        }
    }
```



Теперь давайте немного поменяем задачу. Мы уже научились вычислять допустимые делители для числа. Есть множество чисел, для которых делителем является только 1 и само число. Никаких других делителей k в диапазоне от n>k>1 не существует. Такие числа называются простыми. Например, 7 – простое число. Оно делится только на 1 и на 7. Давайте напишем алгоритм поиска простых чисел на промежутке от 1 до n. Для этого нам необходимо вызвать предыдущий алгоритм поиска делителя для каждого из чисел последовательности и если в результате делителями будут только 1 и само число, то такое число считается простым.

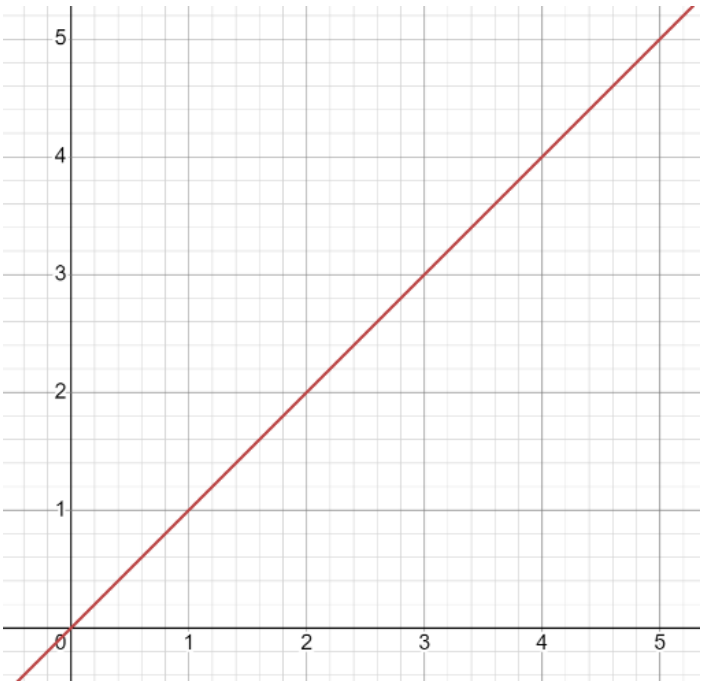
```java
public static List<Integer> findSimpleNumbers(int max) {
    List<Integer> result = new ArrayList<>();
    for (int i = 1; i <= max; i++) {
        boolean simple = true;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                simple = false;
            }
        }
        if (simple) {
            result.add(i);
        }
    }
```

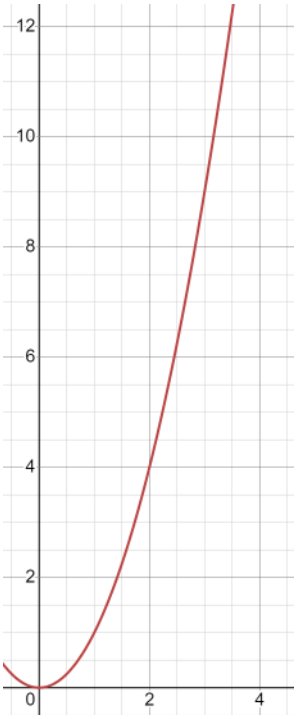
Давайте посчитаем, какое количество операций необходимо нам совершить, чтобы получить результат. Например, для n = 4 мы будем делать 4 повтора алгоритма поиска делителей, где в k = 1 будет 1 операция, в k = 2 будет 2 операции и т.д. В итоге, суммарное количество шагов будет 1 + 2 + 3 + 4 = 10. При n = 5 уже 15, а при n = 6 – 21. Т.е. увеличение n на 1 единицу привносит в алгоритм не 1 шаг, как было при линейном росте, а некоторое весьма значительное число. И чем больше n, тем большее количество шагов будет добавляться в алгоритм с каждым инкрементом. Такой рост сложности можно представить на графике с помощью параболы, где ось x – размер входящего числа, а ось y – требуемое количество операций. При увеличении числа n на 1, количество операций будет увеличиться на n.



```java
public static List<Integer> findSimpleNumbers(int max) {
    List<Integer> result = new ArrayList<>();
    for (int i = 1; i <= max; i++) {
        boolean simple = true;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                simple = false;
            }
        }
        if (simple) {
            result.add(i);
        }
    }
```

Как мы видим, различные алгоритмы по-разному реагируют на увеличение размера входных данных. Именно кривая роста и характеризует сложность алгоритма и понимание, на каком объеме данных его допустимо использовать. Для корректного описания этой кривой используется математическое обозначение – нотация *большого О* – описывающее асимптотическое поведение функции, где под асимптотикой понимается характер изменения функции при ее стремлении к определенной точке. Записывается нотация, как **O(f(x))**, где **f(x)** представляем собой функцию изменения сложности вычисляемого алгоритма. Так, например, для линейной сложности алгоритма, запись выглядит как **O(n)**, что характеризует линейный рост сложности – при изменении n на единицу, количество операций также изменяется на единицу. Для алгоритма поиска простых чисел сложность будет записываться как **O(n^2)** и называться квадратичной.



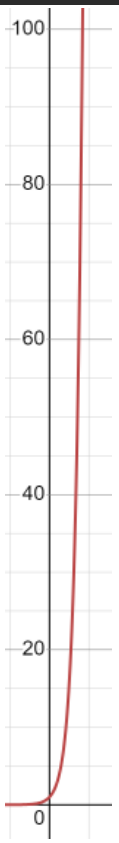


```java
public static List<Integer> findSimpleNumbers(int max) {
    List<Integer> result = new ArrayList<>();
    for (int i = 1; i <= max; i++) {
        boolean simple = true;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                simple = false;
            }
        }
        if (simple) {
            result.add(i);
        }
    }
```

Стоит обратить внимание, что этот алгоритм при подсчете реального количества операций не будет соответствовать функции n^2 – как мы посчитали ранее, для $n = 4$ количество операций является 10, а не 16, но следует понимать, что нотация O используется не для точного расчета предполагаемых операций, а для анализа поведения нашего алгоритма, при росте значения входящих данных. И для этого алгоритма ближе всего именно график квадратичной функции – парабола. Проще всего это понять из характера самой реализации – у нас используется вложенный цикл, чей размер также зависит от размера входящего значения, как и внешний цикл. Соответственно, при использовании вложенных циклов мы будем получать квадратичный, кубические и т.д. сложности, в зависимости от количества вложенных циклов.

Одной из самых плохих вариантов с точки зрения сложности алгоритма является ситуация, когда размер входящих данных отражает количество вложенных циклов, которые необходимо описать. Например, задача по поиску шанса выпадения числа K на игральных кубиках количеством N с количеством граней M. При решении этой задачи «в лоб», нам необходимо перебрать все возможные комбинации, найти те, которые подходят под условие и посчитать их % относительно общего количества комбинаций. Соответственно, цикл размером M будет вложен друг в друга N раз, что даст количество необходимых вычислений как M^N. Например, для трех шестигранных кубиков сложность равна 6^3, что даст примерно 216 операций. А для четырех кубиков уже 1296 операций. График такой функции гораздо более крутой, чем рассмотренная нами ранее парабола и такая сложность называется экспоненциальной и записывается, как **$O(m^n)$**

```java
public static List<Integer> findSimpleNumbers(int max) {
    List<Integer> result = new ArrayList<>();
    for (int i = 1; i <= max; i++) {
        boolean simple = true;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                simple = false;
            }
        }
        if (simple) {
            result.add(i);
        }
    }
```



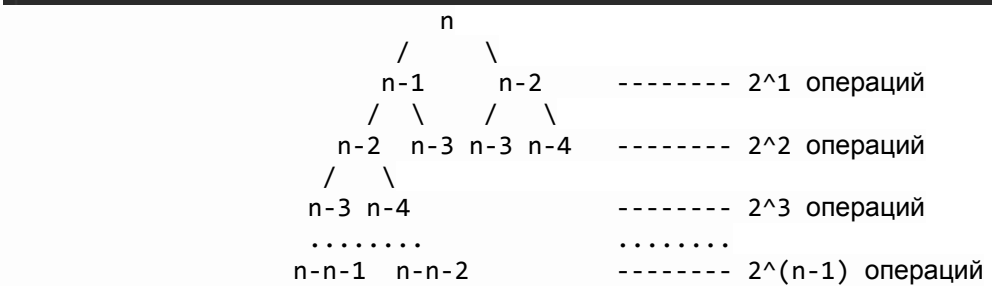
Сложность можно определять не только для алгоритмов, использующих циклы. Анализ сложности применим к любым алгоритмам.
Давайте попробуем оценить сложность рекурсивного алгоритма вычисления чисел Фибоначчи. Это числа, которые вычисляются путем сложения двух предыдущих чисел последовательности, а первые 2 числа в ней 0 и 1.
Алгоритм вычисления проще всего описывается рекурсией

```java
public static List<Integer> findSimpleNumbers(int max) {
    List<Integer> result = new ArrayList<>();
    for (int i = 1; i <= max; i++) {
        boolean simple = true;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                simple = false;
            }
        }
        if (simple) {
            result.add(i);
        }
    }
```

```java
public static int fib(int position) {
    if (position == 1 || position == 2) {
        return 1;
    }
    return fib( position: position - 1) + fib( position: position - 2);
}
```

Как мы видим, для вычисления нужного нам значения запускаются две рекурсивные цепочки, каждая из которых будет запускать еще по 2 рекурсивные цепочки на каждом шаге. При этом, количество операций на 1 уровне равно 2. На втором уровне 4, на третьем уровне 8 и т.д. пока не дойдем до вычисления первого члена последовательности.

```java
public static List<Integer> findSimpleNumbers(int max) {
    List<Integer> result = new ArrayList<>();
    for (int i = 1; i <= max; i++) {
        boolean simple = true;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                simple = false;
            }
        }
        if (simple) {
            result.add(i);
        }
    }
```



Это приводит нас к сложности, примерно равно 2^{n-1}, что является примером экспоненциальной сложности **O(2^n)**. Стоит обратить внимание, что при оценке сложности этого алгоритма в O нотации мы упростили значение степени n-1 до n. Это яркий пример того, что при описании сложности алгоритма следует помнить, что это отражение графика изменения сложности, а не явный расчет количества требуемых шагов.

При оценке сложности всегда сокращаются константные множители и слагаемые, остаются только значимые с точки зрения оценки значения. Так сложность O(2 + n) = O(n), т.к. слагаемое n не несет в себе никакой практической пользы при оценке. Например, при n равной 1000000 дополнительные 2 шага не имеют реальной силы. Так же и O(2n) = O(n), т.к. константный множитель 2 не внесет практической пользы для оценки.

А вот перемножение различных сложностей, например, при использовании одного алгоритма в другом, ведут себя немного иначе. В задаче поиска простых чисел мы использовали алгоритм перебора значений O(n) и алгоритм поиска делителей O(n). Суммарно получили сложность O(n * n), что дает O(n^2). Так вложенные алгоритмы O(n) и O(log n) превратятся в O (n * log n).

При этом, если алгоритмы не вкладываются друг в друга, а используются по очереди, например, сначала алгоритм со сложность O(n), а затем O(n^2), то O(n + n^2) = O(n^2), т.к. при сложении алгоритмов применяется только наибольший из модификаторов.

```java
public static List<Integer> findSimpleNumbers(int max) {
    List<Integer> result = new ArrayList<>();
    for (int i = 1; i <= max; i++) {
        boolean simple = true;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                simple = false;
            }
        }
        if (simple) {
            result.add(i);
        }
    }
```

Чаще всего в работе с алгоритмами можно встретить следующие варианты сложности:

- **O(1)** – константная сложность. Не зависимо от размера входных данных, количество шагов не изменяется. Например, поиск по хэш-таблице.
- **O(log n)** – логарифмическая сложность. Рост размера входных данных почти не оказывает влияния на рост сложности. Для увеличения сложности на 1, размер входящих данных должен вырасти вдвое. Например, бинарный поиск.
- **O(n)** – линейная сложность. Рост сложности линеен по отношению к росту размера входных данных. Например, поиск допустимого делителя, поиск по неотсортированному массиву.
- **O(n * log n)** – рост сложности выше, чем у линейной, но ниже, чем у квадратичной сложности алгоритма. Например, алгоритм быстрой сортировки, сортировки пирамидой.
- **O(n^2)** – квадратичная сложность. Значительный рост сложности от размера входящих данных. Например сортировка пузырьком
- **O(m^n)** – экспоненциальная сложность. При увеличении входящих данных на единицу, сложность растет в m раз.

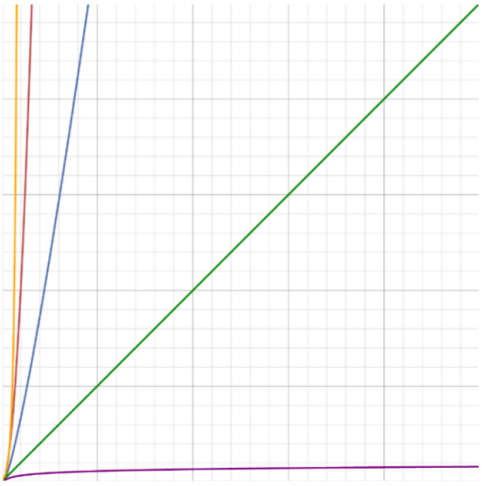
Таким образом, оценить сложность алгоритмов непросто, поскольку:

1. не всегда очевидно, сколько именно итераций будет выполнятся для функции;
2. алгоритм может включать вложенные функции, чью сложность также необходимо учитывать;
3. константные значения могут вводить в заблуждение;
4. подсчет количества итераций может отличаться не только от размера входящих данных, но и от состояния.

```java
public static List<Integer> findSimpleNumbers(int max) {
    List<Integer> result = new ArrayList<>();
    for (int i = 1; i <= max; i++) {
        boolean simple = true;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                simple = false;
            }
        }
        if (simple) {
            result.add(i);
        }
    }
```

График сравнения роста сложности алгоритма

- Желтое - **O(2^n)**
- Красное - **O(n^2)**
- Синее - **O (n * log n)**
- Зеленое - **O (n)**
- Фиолетовое - **O(log n)**



В заключение приведем таблицу, которая показывает, как долго компьютер, осуществляющий миллион операций в секунду, будет выполнять некоторые медленные алгоритмы.

	N=10	N=20	N=30	N=40	N=50
N^2	0,0001 с	0,0004 с	0,0009 с	0,0016 с	0,0025 с
N^3	0,001 с	0,008 с	0,027 с	0,064 с	0,125 с
2^N	0,001 с	1,05 с	17,9 мин	12,7 дней	35,7 лет
3^N	0,05 с	58,1 мин	6,5 лет	3,8 * 10^5 лет	2,27 * 10^10 лет

Подведем итоги:
Умение оценивать сложность выбранного решения крайне важно для разработчика. Использование неоптимального, с точки зрения сложность, решения способно привести к огромной просадке производительности. При этом следует

```java
public static List<Integer> findSimpleNumbers(int max) {
    List<Integer> result = new ArrayList<>();
    for (int i = 1; i <= max; i++) {
        boolean simple = true;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                simple = false;
            }
        }
        if (simple) {
            result.add(i);
        }
    }
```

помнить, что показатель сложности это всего лишь зависимость роста количества операций от размера входящих данных, а не явный расчет требуемого количества шагов. Благодаря тому, что мы оперируем абстрактными данными, мы можем оценивать алгоритмы по их структуре, не вдаваясь в подробности реализации или особенностей входящих данных.

Дополнительные материалы

https://habr.com/ru/post/104219/
https://tproger.ru/articles/computational-complexity-explained/
https://bimlibik.github.io/posts/complexity-of-algorithms/