СОДЕРЖАНИЕ

ГЛАВА 1. НАЧАЛО РАБОТЫ С АЛГОРИТМАМИ

РАЗДЕЛ 1.1. ПРИМЕР АЛГОРИТМИЧЕСКОЙ ЗАДАЧИ

Алгоритмическая задача, решаемая с помощью алгоритма, задается описанием всего множества экземпляров, которые должен обработать алгоритм, и выхода, то есть результата, получаемого после обработки одного из этих экземпляров. Описание одного из экземпляров задачи может заметно отличаться от формулировки общей задачи.

Например, постановка задача сортировки делается таким образом: [Skiena:2008:ADM:1410219]

- Задача: Сортировка
- Входные данные: Последовательность из n элементов: `a_1, a_2, ..., a_n.`
- Выход: Перестановка входной последовательности таким образом, что: `a'_1 <= a'_2 <= ... <= a'_{n-1}<= a'_n`

В качестве примера сортировки может выступать массив строк, например { Haskell, Emacs }, или последовательность чисел, например `{ 154, 245, 1337 }`.

Алгоритм — это процедура, которая принимает любой из возможных входных экземпляров и преобразует его в соответствии с требованиями, указанными в условии задачи.

РАЗДЕЛ 1.2. НАЧАЛО РАБОТЫ С ПРОСТЫМ АЛГОРИТМОМ FIZZ BUZZ НА ЯЗЫКЕ SWIFT

Для тех, кто только начинает программировать на языке программирования Swift, а также для тех, кто пришел из других языков программирования, таких как Python или Java, этот раздел будет весьма полезен. В нем мы рассмотрим простое решение для реализации алгоритма на основе возможностей языка Swift.

Fizz Buzz

Возможно, вы встречали термин Fizz Buzz под другими названиями, такими как FizzBuzz или Fizz-Buzz — все они обозначают одно и то же. Этот «предмет» и является основной темой сегодняшнего разговора. Во-первых, что такое Fizz Buzz?

Этот вопрос часто задают на собеседованиях при приеме на работу. Представьте себе ряд чисел от 1 до 10.

```
1 2 3 4 5 6 7 8 9 10
```

Fizz и Buzz относятся к любым числам, кратным 3 и 5 соответственно. Другими словами, если число кратно 3, то оно заменяется на «fizz», а если кратно 5, то на «buzz». Если число одновременно кратно 3 и 5, то оно заменяется на слвосочетание «fizz buzz». По сути, это эмуляция известной детской игры «fizz buzz».

Чтобы решить эту задачу, откройте Xcode для создания новой программы и инициализируйте массив, как показано ниже:

```
// например
let number = [1,2,3,4,5]
// здесь 3 — это fizz, а 5 — buzz
```

Чтобы найти все ‘fizz’ и ‘buzz’ числа, мы должны обойти весь массив и проверить, какие числа являются ‘fizz’, а какие — ‘buzz’. Для этого создадим цикл for, который будет выполнять итерацию по инициализированному массиву:

```
for num in number {
// Тело программы и расчеты
}
```

После этого мы можем просто использовать условный оператор «if else» и оператор деления по модулю. В swift это — % для того, чтобы найти все ‘fizz’ и ‘buzz’.

```
for num in number {
  if num % 3 == 0 {
    print(“\(num) fizz”)
  } else {
    print(num)
  }
}
```

Отлично! Вы можете перейти в консоль отладки в приложении Xcode Swift Playground, чтобы посмотреть вывод результата работы программы. Вы увидите, что все вхождения «fizz» были найдены в вашем массиве.

Для оставшейся части чисел, которые являются «buzz», мы будем использовать ту же самую технику. Попробуйте сами дописать оставшуюся часть решения, прежде чем дочитаете этот раздел до конца, а потом сравните результаты.

```
for num in number {
  if num % 3 == 0 {
    print(“\(num) fizz”)
  } else if num % 5 == 0 {
    print(“\(num) buzz”)
  } else {
    print(num)
  }
}
```

Проверьте ваш вывод на экран.

Все довольно просто — вы выполнили деление по модулю на число 3, и если остаток от деления получили 0 (а значит проверямое число делится нацело на 3), то вывели все fizz, то же самое проделали с числом 5 и вывели все buzz. Теперь увеличьте количество чисел в массиве:

```
let number = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

Мы увеличили диапазон чисел с 1–10 до 1–15, чтобы продемонстрировать, как найти комбинацию «fizz buzz». Поскольку число 15 кратно 3 и 5, его следует заменить на «fizz buzz». Попробуйте решить эту задачу самостоятельно и проверьте ответ.

Вот решение:

```
for num in number {
  if num % 3 == 0 && num % 5 == 0 {
    print("\(num) fizz buzz")
  } else if num % 3 == 0 {
    print("\(num) fizz")
  } else if num % 5 == 0 {
    print("\(num) buzz")
  } else {
    print(num)
  }
}
```

Но постойте! Это еще не все! Основная цель алгоритма — оптимизировать время выполнения. Представьте, что диапазон увеличится с 1–15 до 1–100. Программа проверит каждое число, чтобы определить, делится ли оно на 3 и на 5 одновременно. Затем программа снова «прогонит» все числа, чтобы проверить, являются ли они кратными 3 или 5. То есть по сути программе придется дважды проверять в массиве одно и тоже число. Сначала надо выполнить деление по модулю на 3, а потом и деление по модулю на 5.

Чтобы ускорить процесс, мы можем просто указать нашей программе делить числа сразу на 15.

Вот окончательный вариант программы:

```
for num in number {
  if num % 15 == 0 {
    print("\(num) fizz buzz")
  } else if num % 3 == 0 {
    print("\(num) fizz")
  } else if num % 5 == 0 {
    print("\(num) buzz")
  } else {
    print(num)
  }
}
```

Как видите, ничего сложного нет. Решение по этому способу подойдет для любого другого языка программирования.

ГЛАВА 2. СЛОЖНОСТЬ АЛГОРИТМОВ

РАЗДЕЛ 2.1. «Θ БОЛЬШОЕ»

В отличие от математического обозначения «O большое», которое представляет собой только верхнюю границу времени работы некоторого алгоритма, «Θ большое» — это более точная величина, отражающая время работы алгоритма и лежащая между верхней и нижней границами времени работы алгоритма. Точная граница является наиболее приближеной к реальному времени работы алгоритма, но она в то же время и более сложна для вычисления.

Математическая функция «Θ большое» симметрична: `f(x) = Θ(g(x)) <=> g(x) = Θ(f(x))`. Интуитивно можно понять, что `f(x) = Θ(g(x))` означает, что графики `f(x)` и `g(x)` растут с одинаковой скоростью, или что графики «ведут себя» одинаково при достаточно больших значениях `x`.

Полное математическое выражение «Θ большое»выглядит следующим образом:

```
Θ(f(x)) = {g: N0 -> R и c1, c2, n0 > 0, где c1 < abs(g(n) / f(n)), для
каждого n > n0, а abs – модуль числа}
```

Пример

Если алгоритм для входного `n` требует `42n^2 + 25n + 4` операций, то мы говорим, что его сложность составляет `O(n^2)`, но с таким же успехом она может быть равна и `O(n^3)`, и `O(n^100)` и т. д. Однако в этом конкретном случае вычислительная сложность, выраженная через «O большое», равна именно `O(n^2)`, а не `O(n^3)` и не `O(n^4)` и т. д. Алгоритм, вычислительная сложность которго равна `Θ(f(n))`, также может иметь вычислительную сложность, равную `O(f(n))`, но не наоборот.

Формальное математическое определение

Пусть `Θ(g(x))` — некоторое множество функций.

```
Θ(g(x)) = {f(x) такая, что существуют положительные константы c1, c2,
N такие, что 0 <= c1*g(x) <= f(x)<= c2*g(x) для всех x > N}
```

Функция `f(x)` принадлежит множеству `Θ(g(x))`, если существуют положительные константы `c1` и `c2`, такие, что при достаточно больших `n` эта функция может быть заключена в рамки между `c1*g(x)` и `c2*g(x)`. Поскольку `Θ(g(x))` представляет собой множество, можно записать «`f(x) ∈ Θ(g(x))`», чтобы указать тот факт, что f(x) является членом `Θ(g(x))`. Вместо этого обычно используют эквивалентную запись «`f(x) = Θ(g(x))`».

Всякий раз, когда в формуле появляется `Θ(g(x))`, мы интерпретируем ее как обозначение некоторой анонимной функции, которую мы можем не называть. Например, уравнение `T(n) = T(n/2) + Θ(n)` означает, что `T(n) = T(n/2) + f(n)`, где `f(n)` — функция, принадлежащая множеству `Θ(n)`.

Пусть `f` и `g` — две функции, определенные на некотором подмножестве действительных чисел. `f(x) = Θ(g(x))` при `x`, стремящемся к бесконечности, выполняется при условии, когда существуют такие положительные постоянные `K` и `L` и действительное число `x0`, что выполняется равенство:

`K|g(x)| <= f(x) <= L|g(x)|` для всех `x >= x0`.

Данное определение равносильно тому, что:

`f(x) = O(g(x))` и `f(x) = Ω(g(x))`

Определение на основе предела функции

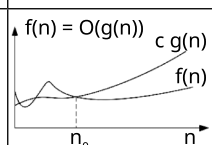
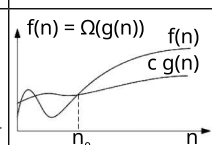
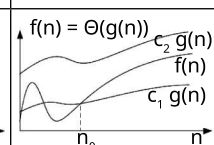
Если существует предел `limit(x->бесконечность) f(x)/g(x) = c ∈ (0,∞)`, и он равен некоторому неотрицательному числу, то выполняется `f(x) = θ(g(x))`.

Общие классы вычислительной сложности

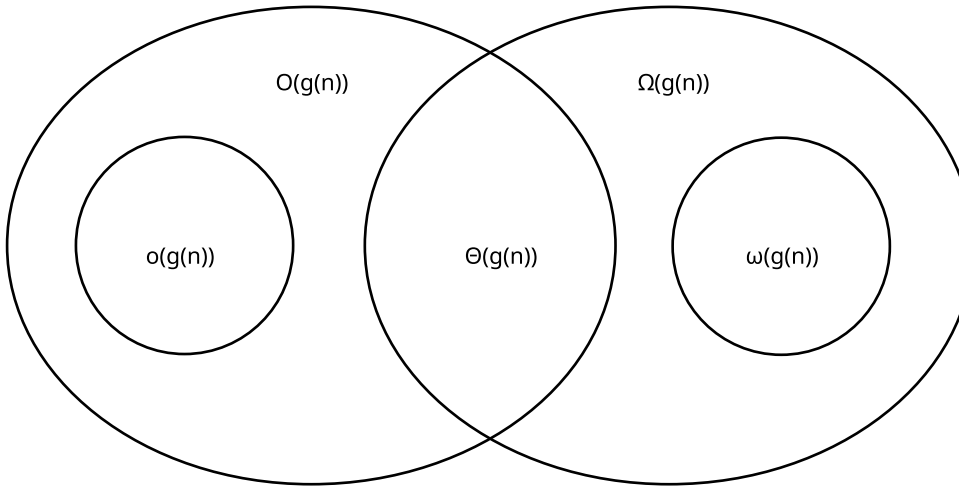
Название	Условные обозначения	n = 10	n = 100
Константное время	θ(1)	1	1
Логарифмическое время	θ(log(n))	3	7
Линейное время	θ(n)	10	100
Линейно-арифметическое время	θ(n*log(n))	30	70
Квадратичное время	θ(n^2)	100	10 000
Экспоненциальное время (с линейной экспонентой)	θ(2^n)	1 024	1.267650e+ 30
Факториальное время	θ(n!)	3 628 800	9.332622e+157

РАЗДЕЛ 2.2. СРАВНЕНИЕ АСИМПТОТИЧЕСКИХ ОБОЗНАЧЕНИЙ

Пусть `f(n)` и `g(n)` — две функции, определенные на множестве положительных вещественных чисел, и c, c1, c2, n0 — положительные вещественные константы.

Условные обозначения	f(n) = O(g(n))	f(n) = Ω(g(n))	f(n) = Θ(g(n))	f(n) = o(g(n))	f(n) = ω(g(n))
Формальное определение	∃ c > 0, ∃ n0 > 0 : ∀ n ≥ n0, 0 ≤ f(n) ≤ c g(n)	∃ c > 0, ∃ n0 > 0 : ∀ n ≥ n0, 0 ≤ c g(n) ≤ f(n)	∃ c1, c2 > 0, ∃ n0 > 0 : ∀ n ≥ n0, 0 ≤ c1 g(n) f(n) ≤ c2 g(n)	∀ c > 0, ∃ n0 > 0 : ∀ n ≥ n0, 0 ≤ f(n) < c g(n)	∀ c > 0, ∃ n0 >0 : ∀n ≥n0, 0 ≤ c g(n) < f(n)
Аналогия между асимптотическим сравнением f, g и вещественными числами a, b	a ≤ b	a ≥ b	a = b	a < b	a > b
Пример	7n + 10 = O(n^2 + n - 9)	n^3 - 34 = Ω(10n^2 - 7n + 1)	1/2 n^2 - 7n = Θ(n^2)	5n^2 = o(n^3)	7n^ 2 = ω(n^ 3)
Графическая интерпретация					

Асимптотические обозначения могут быть представлены на диаграмме Венна следующим образом:



Ссылки

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms.

РАЗДЕЛ 2.3. «Ω БОЛЬШОЕ»

Термин «Ω большое» используется для обозначения нижней асимптотической границы.

Формальное определение

Пусть `f(n)` и `g(n)` — две функции, определенные на множестве положительных действительных чисел. Тогда выполняется `f(n) = Ω(g(n))`, но только в том случае, если существуют положительные константы c и n0, такие, что выполняется следующее условие:

`0 ≤ c g(n) ≤ f(n) для всех n ≥ n0.`

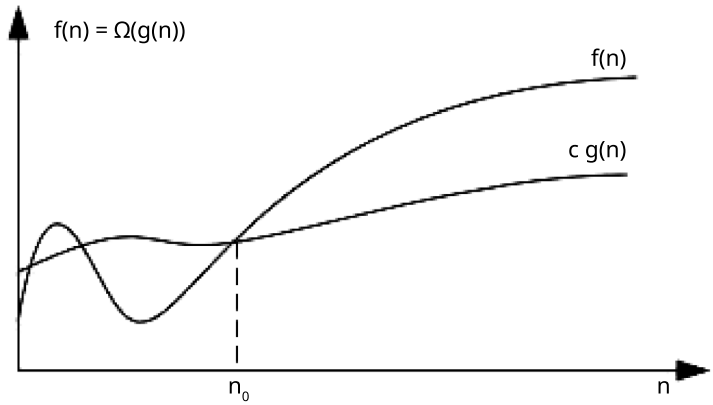
Иными словами, `f(n) = Ω(g(n))` означает, что функция `f(n)` ограничена снизу функцией `c • g(n)`. То есть существует такая константа c, для которой `f(n) ≥ c • g(n)` для всех `n ≥ n0`.

Примечания

Запись `f(n) = Ω(g(n))` означает, что `f(n)` асимптотически растет не медленнее, чем `g(n)`. То есть когда нельзя сделать однозначных выводов о быстродействии на основе полученных значений `θ(g(n))` и/или `O(g(n))`, тогда мы обращаемся к величине `Ω(g(n))`.

Следствием из вышеприведенных определений является теорема:

Для двух любых функций `f(n)` и `g(n)` выполняется `f(n) = θ(g(n))`, но лишь тогда, когда `f(n) = O(g(n)) и f(n) =Ω(g(n))`. Графически «Ω большое» можно представить следующим образом:



Рассмотрим пример. Пусть у нас есть функция `f(n) = 3n^2 + 5n – 4`. Тогда нижней границей вычислительной сложности для нее будет `f(n)= Ω(n^2)`. Также верно и утверждение `f(n) = Ω(n)`, или даже `f(n)= Ω(1)`.

Рассмотрим другой пример. Решение алгоритма попарного совпадения ребер графа: если число вершин нечетное, то должно быть выведено «Нет попарного совпадения», иначе необходимо перебрать все возможные совпадения.

Мы хотели бы сказать, что алгоритм требует экспоненциального времени, но на самом деле невозможно доказать нижнюю границу `Ω(n^2)`, используя обычное определение Ω, поскольку алгоритм выполняется за линейное время для нечетных n. Вместо этого мы должны определить `f(n)=Ω(g(n))`, при условии, что существует некоторая константа `c>0`, такая, что выполняется `f(n)≥ c g(n)` для любых n. Это дает хорошее соответствие между верхней и нижней границами: `f(n)=Ω(g(n))`, если `f(n) ! = o(g(n))`.

Ссылки

Формальное определение и теорема взяты из книги Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms.

ГЛАВА 3. «О БОЛЬШОЕ»

Определение

Обозначение «O большое» по своей сути является математическим обозначением, используемым для сравнения скорости сходимости функций. Пусть `n -> f(n)` и `n -> g(n)` — функции, определенные на множестве натуральных чисел. Тогда мы говорим, что `f = O(g)` тогда и только тогда, когда `f(n)/g(n)` имеет конечный предел при стремлении n к бесконечности. Другими словами, `f = O(g)` тогда и только тогда, когда существует константа A, такая, что для всех n, `f(n)/g(n) ≤ A`.

На самом деле область применения обозначения «O большое» в математике несколько шире, но для простоты мы сузили ее до функции, которая используется при анализе вычислительной сложности алгоритмов: функции, определенной на множестве натуральных чисел, имеющей ненулевые значения, включая случай, когда n растет до бесконечности.

Что это означает?

Рассмотрим случай `f(n) = 100n^2 + 10n + 1` и `g(n) = n^2`. Совершенно очевидно, что обе эти функции стремятся к бесконечности, поскольку n стремится к бесконечности. Но иногда знать предел недостаточно. Мы также хотим знать скорость, с которой функции приближаются к своему пределу. Такие понятия, как «O большое», помогают сравнивать и классифицировать функции по скорости их сходимости.

Выясним, принадлежит ли функция f множесту функций `O(g)` `f = O(g)`, применив ряд вычислений. Имеем `f(n)/g(n) = 100 + 10/n + 1/n^2`. Поскольку слагаемое `10/n` равно 10, когда n равно 1 и убывает по мере возрастания значения n, и поскольку `1/n^2` равно 1, когда n равно 1 и также убывает при возрастании значения n, то получим `f(n)/g(n) <= 100 + 10 + 1 = 111`. Условие выполняется, так как мы нашли границу `f(n)/g(n)` (111), и поэтому `f = O(g)` (в этом случае говорят, что f является функцией O от `n^2` или `O(n^2)`).

Это означает, что `f` стремится к бесконечности примерно с той же скоростью, что и `g`. Это может показаться странным, поскольку мы обнаружили, что `f` почти в 111 раз больше, чем `g`, или, другими словами, когда `g` увеличивается на 1, `f` увеличивается максимум на 111. Может показаться, что рост в 111 раз — это не совсем «примерно та же скорость». И действительно, «O большое» — не очень точный способ классификации скорости сходимости функций, поэтому в математике, когда нам нужна точная оценка скорости сходимости, мы используем асимптотический анализ. Но для целей разделения алгоритмов на большие классы по вычислительной сложности достаточно и величины «O большое». Нам не надо разделять между собой функции, которые растут в фиксированное число раз быстрее друг друга, а только функции, которые растут на порядок быстрее друг друга. Например, если мы возьмем `h(n) = n^2*log(n)`, то увидим, что `h(n)/g(n) = log(n)`, который стремится к бесконечности с ростом n, поэтому h не принадлежит к классу функций с вычислительной сложностью `O(n^2)`, так как h растет на порядок быстрее, чем `n^2`.

Теперь следует сделать небольшое замечание: вы, наверное, заметили, что если `f = O(g)`, а `g = O(h)`, то `f = O(h)`. Например, в нашем случае мы имеем `f = O(n^3)`, при этом `f = O(n^4)...` В анализе сложности алгоритмов часто говорят `f = O(g)`, имея в виду, что `f = O(g)` и `g = O(f)`, что можно понимать как «g — наименьшее «O большое» для f». В математике говорят, что такие функции являются «Θ большими» друг друга.

Как все это применять?

При сравнении производительности алгоритмов нас интересует количество операций, которые выполняет алгоритм. Это называется вычислительной сложностью. В этой модели мы считаем, что каждая базовая операция (сложение, умножение, сравнение, присваивание и т. д.) занимает конечное количество времени, и подсчитываем количество таких операций. Обычно мы можем выразить это число как функцию от размера входных данных n. И, к сожалению, это число очень быстро возрастает до бесконечности с увеличением n (если этого не происходит, мы говорим, что алгоритм является `O(1)`). Мы разделяем наши алгоритмы на большие классы скорости, в зависимости от «O большого»: когда мы говорим об «`O(n^2)` алгоритме», мы имеем в виду, что количество выполняемых им операций, выраженное как функция от n, равно `O(n^2)`. Это означает, что наш алгоритм работает примерно с такой же скоростью, как и алгоритм, выполняющий количество операций, равное квадрату размера его входных данных, или немного быстрее. Словосочетание «немного быстрее» употребляется, поскольку мы использовали значение «O большое» вместо «Θ большое», но обычно говорят «O большое», подразумевая под этим «Θ большое».

При подсчете операций обычно рассматривается худший случай: например, если у нас есть цикл, который может выполняться не более n раз и содержит 5 операций, то общее количество операций будет равно 5n. Можно также рассматривать и среднюю сложность.

Небольшое замечание: быстрый алгоритм — это тот, который выполняет мало операций, поэтому если количество операций возрастает до бесконечности быстрее, то алгоритм работает медленнее: `O(n)` лучше, чем `O(n^2)`.

Иногда нас также интересует пространственная сложность нашего алгоритма. Для этого мы считаем количество байт в памяти, занимаемых алгоритмом, как функцию от размера входных данных и аналогично используем «O большое».

РАЗДЕЛ 3.1. ПРОСТОЙ ЦИКЛ

Следующая функция находит максимальный элемент в массиве:

```
int find_max(const int *array, size_t len) {
    int max = INT_MIN;
    for (size_t i = 0; i < len; i++) {
        if (max < array[i]) {
            max = array[i];
        }
    }
    return max;
}
```

Входной размер — это размер массива, который задан в переменной `len`. Давайте посчитаем количество операций.

```
int max = INT_MIN;
size_t i = 0;
```

Эти два присваивания выполняются только один раз, так что это две операции. Операции, которые выполняются в цикле, следующие:

```
if (max < array[i]) i++;
max = array[i]
```

Так как в цикле три операции, а цикл выполняется n раз, то к уже имеющимся двум операциям добавляем 3n и получаем 3n + 2. Таким образом, для нахождения максимального значения нашей функции требуется 3n + 2 операции (ее вычислительная сложность равна 3n + 2). Это многочлен, в котором самый быстрорастущий член равен n, поэтому его сложность составляет O(n).

Вы, наверное, заметили, что понятие «операция» не очень удачно определено. Например, было сказано, что if (max < array[i]) — это одна операция, но в зависимости от архитектуры после компиляции она может преобразоваться, например, в три инструкции: одна — чтение из памяти, одна — сравнение и одна — ветвление. Мы также считали все операции одинаковыми, несмотря на то, что, например, операции с памятью будут медленнее остальных, и их производительность будет сильно отличаться, например, из-за эффекта кэширования. Мы также полностью проигнорировали оператор return, при срабатывании которого произойдет измеение стека памяти и т. д. В конечном итоге для анализа сложности это не имеет значения, поскольку, какой бы способ подсчета операций ни был бы выбран, изменятся только коэффициент при множителе n и константа, поэтому результат все равно будет O(n). Сложность показывает, как алгоритм масштабируется с размером входных данных, но это не единственный аспект производительности.

РАЗДЕЛ 3.2. ВЛОЖЕННЫЙ ЦИКЛ

Рассмотрим функцию, которая проверяет наличие дубликатов в массиве, беря каждый элемент, а затем выполняя итерацию по всему массиву, чтобы проверить, встречается ли еще в нем этот элемент:

```
_Bool contains_duplicates(const int *array, size_t len) {
    for (int i = 0; i < len - 1; i++) {
        for (int j = 0; j < len; j++) {
            if (i != j && array[i] == array[j]) {
                return 1;
            }
        }
    }
    return 0;
}
```

Внутренний цикл выполняет на каждой итерации количество операций, постоянное по отношению к n. Внешний цикл также выполняет несколько постоянных операций и запускает внутренний цикл n раз. Сам внешний цикл выполняется n раз. Таким образом, операции внутри внутреннего цикла выполняются раз, операции во внешнем цикле выполняются n раз, а присваивание i выполняется один раз. Таким образом, сложность будет иметь вид an^2 + bn + c, а поскольку наибольший член равен n^2, то в обозначении «O большое» будет равно O(n^2).

Однако, как можно заметить, мы можем улучшить алгоритм, избежав многократного выполнения одних и тех же сравнений. Во внутреннем цикле мы можем начать с i + 1-й элемент, так как все элементы до него уже будут проверены на соответствие всем элементам массива, включая элемент с индексом i + 1. Это позволяет отказаться от проверки i == j.