Подсчет сложности алгоритма

О-большое

Попробуем научиться считать О-сложность алгоритма. Сначала вспомним определения О-большого при x стремящемся к бесконечности и свойства О-большого

Пусть $f(x)$ и $g(x)$ - две функции от x Функция f является О-большим от g при $x \to \inf$, если
существует такая константа $C > 0$, что для всех x

$$|f(x)| <= C|g(x)|$$

Свойства О-большого, которые нам полезны:

1. $C \cdot O(f) = O(f)$
2. $O(C \cdot f) = O(f)$
3. $O(f) + O(f) = O(f)$
4. $O(f) \cdot O(g) = O(fg)$
5. $O(O(f)) = O(f)$

о-малое

Так же нам будет полезно определение о-малого ($o(x)$) Пусть $f(x)$ и $g(x)$ - две функции от x Функция f является о-малым от g при $x \to \inf$, если при $x \to \inf$

$$f(x)/g(x) \to 0$$

и его свойство:

1. o(x) + O(x) = O(x)

Элементарные операции

Есть набор операций, для которых мы подразумеваем **фиксированное** (O(1)) время выполнения. Такие операции считаем **элементарными**. Типичный пример элементарной операции - сложение двух 64-битных целых чисел или присвоение значения переменной. Количество таких операций и говорит нам о сложности кода. Например, приведенный ниже фрагмент кода выполняется за время, которое требуется на то, чтобы совершить 3 элементарных операции.

```
a = 5   // O(1)
b = 10  // O(1)
c = a + b // O(1)
```

Каждая из трех операций выполняется за O(1) и выполняется один раз.
Точной O-оценкой для времени выполнения всего фрагмента является O(1) + O(1) + O(1) = O(1)

Важно

В дальнейшем мы следуем следующему алгоритму подсчета сложности фрагмента кода.

1. Подсчитать сложность КАЖДОЙ строчки данного кода НЕЗАВИСИМО:
    1. Сначала считается, за какое время эта строчка исполняется один раз.
    2. Затем мы считаем, сколько раз эта строчка вообще будет исполняться за время работы программы
    3. После этого мы вычисляем сколько в итоге будет исполнятся по времени эта строчка (в O-нотации) 4.*\ В простых случаях, а так же в случаях, когда такая строчка уже разбиралась, первые два этапа могут опускаться и сразу записываться сложность, посчитанная в третьем пункте.
2. Подсчитать суммарную сложность фрагмента кода путем сложения сложностей каждой из строчек.

Пример 1

```
N: int = input(); // O(1), так как размер int ограничен константой

s = 0; // O(1)
for i in range(0, N) { // O(1) * N, как минимум мы каждую итерацию тратим какое-то время,
   // чтобы вернуться в начало цикла
   // и в первый раз тратим время,  чтобы это начало запомнить
   s = s + 1; // O(1) * N
}
```

В данном случае очевидно, что увеличение s на 1 занимает константное время $O(1)$
Сама строка будет исполнена N раз (из-за того, что она находится внутри цикла).
Потому в целом эта строчка отнимет $N \cdot O(1) = O(N)$ времени.

На одну итерацию даже пустого цикла будет тратиться какое-то время. Почему? Перед первой итерацией мы потратим какое-то время, чтобы запомнить место начала

нашего цикла. В последующих итерациях мы будем тратить время, чтобы вернуться в начало нашего цикла.

Заметим, что $O(1)$ есть $o(N)$, потому из рассмотренного ранее свойства:

$$O(1) + O(N) = O(N)$$

Весь же код займет

$$O(1) + O(1) + O(N) + O(N) = O(N)$$

Пример 2

```
N: int = input(); // O(1)

s = 0; // O(1)
for i in range(0, N) { // O(1) * 11 = O(1)
   if (s == 10) { // 11 * O(1) = O(1) // на одно сравнение int-чисел тратим O(1)
       break;  // O(1)
   }
   s = s + 1; // O(1) * 10 = O(1)
}
```

В данном случае мы встретились с операцией сравнения. На сравнение двух int-чисел тратится O(1) времени (как и для случая чисел с плавающей запятой). Сама строчка будет исполнена не больше 11 раз, так как как только i становится равным 10, выполняется команда break. Строка с увеличением s будет исполнена 10 раз.

Потому суммарное время работы такого кода оказывается равным

$$O(1) + O(1) + O(1) * 11 + O(1) * 11 + O(1) * 11 + O(1) = O(1)$$

Пример 3

```
N: int = input(); // O(1)
M: int = input(); // O(1)

s = 0; // O(1)
for i in range(0, N) { // min(M + 1, N) * O(1) = O(min(M + 1, N))
   if (s == M) { // min(M + 1, N) * O(1) = O(min(M + 1, N))
       break;  // O(1)
   }
   s = s + 1; // O(1) * min(M, N) = O(min(N, M))
}
```

В данном случае мы войдем в цикл M + 1 раз, если M < N и N раз если M >= N. Разные строчки в цикле при этом исполнятся немного разное число раз, но это не

существенно. Потому суммарное время работы такого кода оказывается равным

$$O(min(M+1,N)) + O(min(M+1,N)) + O(min(M,N)) = O(min(M,N))$$

Достаточно очевидно, что 1 в M + 1 можем пренебречь.

Пример 4. Сложение строк

Сложение же двух строк не является элементарной операцией.

В случае приведенного выше кода, если размер строки (в байтах), который подается в *a* равен *M*, а размер строки, который подается в *b*, равен *N*, то итоговое время работы кода будет равняться *O(N) + O(M) + O(A)*, где A - время, которое нам необходимо на сложение двух строк.

Чему равно A? Для этого нам необходимо разобраться, а как работает сложение строк. В упрощенном виде оно выглядит так. Сразу заметим, что удобно писать для каждой строки, сколько времени она будет выполняться за всю работу рассматриваемого фрагмента.:

```
fn add(a: String, b: String) -> String {
    N = a.length(); // O(1), узнаем размеры строк, обе операции займут
константное время
    M = b.length(); // O(1)
    c_length = N + M; // O(1)
    c = String::with_size(c_length); // создаем новую строку размера N + M,
    // строго говоря, может занять разное время, но можем предположить, что это
O(1)

    for i in range(0, N) { // O(N)
        c[i] = a[i]; // O(N), каждое присвоение занимает 1 операцию,
    // кроме того, операция индексации строки занимет 1 операцию,
    // таких индексаций делаем две
    // а выполним мы эту строчку N раз.
    // Следовательно, эта строка выполнится N * (O(1) + 2 * O(1)) = O(N) раз
    }
    cur_size = N; // O(1)
    for i in range(0, M) { // O(M)
        c[cur_size] = b[i]; // O(M) операций
    cur_size = cur_size + 1; // O(M) операций
    }

    return c; // O(1)
}
```

Если сложить все полученные времена для каждой из строк, то получим:

$$O(1) + O(1) + O(1) + O(1) + O(N) + O(N) + O(1) + O(M) + O(M) + O(M) + O(1) =$$

Значит, сложение двух строк длины N и M занимает A = O(N + M) времени.

Пример 5. Сравнение строк

Сравнение строк так же не является элементарной операцией.

```
fn is_equal(a: String, b: String) -> bool {
    N = a.length(); // O(1)
    M = b.length(); // O(1)
    if (N != M){ // O(1)
        return false; // O(1)
    }

    for i in range(0, N) { // O(1) * m, где m может быть от 0 до N
        if a[i] != b[i] { // O(1) * m, где m -//-
            return false; // O(1) * m, где m -//-
    }
    }

    return true; // O(1)
}
```

В данном случае время выполнения функции зависит от входных данных. Если строки изначально разной длины, то функция выполнится за $O(1) + O(1) + O(1) + O(1) = O(1)$ времени.

Если строки где-то различаются, то выполнение может занять разное количество времени, если они различаются во втором символе - то все равно $O(1)$, а если в середине - то $O(N/2) = O(N)$ времени. Аналогично, если строки совпадают, то функция отработает $O(N)$ времени.

Обычно в таких случаях могут рассматривать три O-сложности:

1. Сложность в лучшем случае
2. Сложность в среднем случае
3. Сложность в худшем случае

Сложность в лучшем случае

В данном случае O-сложность в лучшем случае равна O(1) - когда наши строки имеют разную длину.

Сложность в среднем случае

В случае сложность в среднем случае нам спрашивают о том, как устроено матожидание числа операций, которые выполняет наш алгоритм.

Разбор среднего случая требует дополнительного знания о природе данных. Например, в случае сравнения строк можно рассуждать вот так: В случае наличия k символов в алфавите, вероятность их совпадения у двух случайных строк в i-й позиции

равна 1/k. При этом первый же несовпавший символ влечет за собой выход из функции. Таким образом мы проверим s позиций до первого несовпадения. Число s (номер первого несовпадения) распределено геометрически. Матожидание s равно $1/(1-1/k) = k/(k-1)$.
В таком случае, если мы предполагаем фиксированный небольшой алфавит, то k * O(1) = O(1)

В то же время в силу разной встречаемости разных символов алфавита и прочих факторов, может вполне оказаться, что в реальности средний случай имеет O(N) сложность.

По причине того, что для ответа на вопрос о сложности в среднем случае необходимо знать что-то о реальных данных, ответ на вопрос о среднем случае чаще всего затруднителен (хотя и представляет наибольший практический интерес)

Сложность в худшем случае

Обычно же когда спрашивают о O-сложности алгоритма имеют ввиду именно сложность в худшем случае. В худшем случае (строки равны), мы потратим O(N) времени на их сравнение.

Пример 6.

```
N: int = input(); // O(1)
k = 0; // O(1)

while k < N { // O(1) * N = O(N) на сравнение
    k += 1; // O(1) * N = O(N)
}
```

Аналогично со случаем с циклом for в данном случае мы потратим O(N) действий

Пример 7

```
N: uint = input(); // O(1), uint - от 0 до 2 ^ 31 - 1

while N != 0 { (k + 1)* O(1), k - надо найти, число проверок условия на 1
больше, чем исполнений тела цикла
    N //= 2; // k * O(1)
}
```

Нам надо найти значение k - сколько раз будет выполнена операция сравнения. Попробуем представить, что мы знаем не N, а k и хотим оценить N. Так как на каждом

этапе N уменьшалось минимум в 2 раза (если оно на этом этапе было четное, иначе дополнительно откидывалась 1).

Тогда мы можем утверждать, что $2^k \leq N < 2^{(k+1)}$ (легко можно проверить на 4 и 9)

В таком случае $k > \log N - 1$ и $k <= \log N$. Отсюда получаем, что k есть O(\log N) сложность фрагмента кода - O(\log N)

Пример 8

```
N: uint = input(); // O(1), uint - от 0 до 2 ^ 31 - 1

while N != 0 { (k + 1)* O(1), k - надо найти, число проверок условия на 1
больше, чем исполнений тела цикла
    N //= 7; // k * O(1)
}
```

Рассуждая аналогично предыдущему примеру, получаем ответ

$$O(logN)$$

В этом и прошлом примере при записи сложности мы опустили основание логарифма. Это допустимо, так как все логарифмы связаны между собой следующим соотношением:

$$\log_a^b = \frac{\log_d b}{\log_d a}$$

Отсюда следует, что

$$O(\log_2 N) = O(C \cdot \log_7 N) = O(\log_7 N)$$

Для простоты основание просто опускается

$$O(\log_2 N) = O(\log_7 N) = O(\log N)$$

Пример 9

```
N: uint = input(); // O(1)

s = 0; // O(1)
for i in range(0, N) { // O(N)
    for j in range(0, N){ // O(N) * N = O(N^2)
        s += 1; // O(1) * N * N = O(N^2), N итераций внешнего цикла по N
итераций внутреннего цикла
    }
}
```

В данном случае строка с увеличением s будет выполняться N раз для каждого запуска внутреннего цикла, который будет запущен N раз. Потому сложность данного фрагмента кода - $O(N^2)$

Пример 10

```
N: uint = input(); // O(1)

s = 0; // O(1)
for i in range(0, N) { // O(N)
    for j in range(0, i){ // для i = 0 не зайдем в цикл
        s += 1; // сумма от i = 1 до i = N ( O(1) * i )
    }
}
```

Мы можем прикинуть оценку времени работы алгоритма - в среднем на каждую итерацию внешнего цикла внутренний цикл делает O(N/2) = O(N) итераций. Строка в теле цикла так же выполняется O(N) раз. Внешний цикл выполняется N раз. Таким образом сложность этого фрагмента кода

$$O(1) + O(1) + O(N) * N = O(N^2)$$

Можно посчитать и точно. За каждую i итерацию внешнего цикла выполняется i итераций внутреннего цикла. Тогда всего внутренний цикл выполнится

$$\sum_{i=0}^{N-1} O(i) = O((N-1) * (N-2)/2) = O(N^2)$$

В последнем равенстве мы использовали формулу суммы арифметической прогрессии. Таким образом, наша точная оценка совпала с прикидкой.

Пример 11

```
N: uint = input(); // O(1)

for i in range(0, N){ // (O(N))
    s = N; // O(1) * N = O(N)
    while s != 0 { // O(1) * N * O(logN)
        s = s // 2; // O(1) * N * O(logN) = O(NlogN)
    }
}
```

Для каждой итерации внешнего цикла строка во внутреннем цикле будет выполняться O(\log N) раз (как было показано в примере 7). Всего будет N итераций внешнего цикла, потому в результате получаем сложность

$$O(N) + O(1) * O(\log N) * N + O(1) * O(\log N) * N + O(N) = O(N \log N)$$

Пример 12

```
N: uint = input(); // O(1)

for i in range(0, N){ // O(N)
    s = i; // O(1) * N = O(N)
    while s != 0 { сумма от i = 1 до i = N O(1) * O(logi) и O(1) для i = 0
        s = s / 2; // сумма от i = 1 до i = N  O(1) * O(logi)
    // считаем, что int делится на 2 целочисленно. То есть 5 / 2 = 2
    }
}
```

Здесь мы имеем две возможности оценить время работы приведенного фрагмента кода. Грубым приближением будет считать, что в среднем все части внутреннего цикла выполняются

$$O(\log(N/2)) = O(\log N)$$

раз. Тогда наша оценка времени работы -

$$O(\log N) * N + O(N) = O(N \log N)$$

Можно попытаться оценить и более точно. Для каждой i итерации внешнего цикла, части, связанные с внутренним циклом будет исполняться O(\log i) раз (кроме i = 0, когда выполнится только проверка на условие). Тогда всего эти части будет исполнены

$$O(1) + O(N) + O(N) + \sum_{i=1}^{i=N} O(\log i) = O(\log N!)$$

В последнем равенстве мы воспользовались тем, что

$$\log a + \log b = \log(ab)$$

Далее можно вспомнить формулу Стирлинга:

$$N! \sim \sqrt{2\pi N}(\frac{n}{e})^n$$

Тогда для логарифма факториала

$$logN! \sim \frac{1}{2} \log 2\pi N + N \log \frac{N}{e} = O(\logN) + O(N\logN) = O(N\logN)$$

$$O(O(N\logN)) = O(\logN)$$

То есть получаем такую же оценку, которую и получили с грубой прикидкой.

Пример 13

```
N: uint = input(); // O(1)

flag = false; // O(1)
for i in range(1, N + 1){ // O(N)
    s = 0; // O(N)
    while s != i and not flag { // O(1) * 2 + ... O(1) * 7 + O(1) * 8 + O(1) *
(N - 7) = O(N)
        // пока flag = false цикл крутится пока s не станет равным N
    // последняя проверка условия говорит, что s == N и в цикл мы не заходим
    // (отсюда проверок условия на 1 больше, чем число исполнений тела цикла)
    // после того как flag стал равным true происходит только одна проверка
    s += 1; // O(1) * 1 + ... O(1) * 6 + 7 * O(1) = O(1)
        if s % 7 == 0 { // O(1) * 1 + ... O(1) * 6 + 7 * O(1) = O(1)
        flag = true; // O(1), как только flag становится равным true, больше в
цикл мы не заходим
    }
    }
}
```

В данном случае внутренний цикл выполняется только до тех пор, пока flag не примет значение true. Как только это происходит (на 7 итерации, когда i = 7), в дальнейшем на каждой итерации внешнего цикла при входе в внутренний цикл мы только один раз проверяем условие. Потому суммарное время работы кода получается равным

$$O(1) + O(N) + O(N) + O(1) + O(1) + O(1) = O(N)$$

Пример 14

```
N: uint = input(); // O(1)

for i in range(1, N + 1){ // O(N)
    flag = false; // O(N)
    s = 0; // O(N)
    while s != i and not flag { // O(1) * 2 + ... O(1) * 7 + O(1) * (N - 6) * 8
= O(N)
        // пока flag = false цикл крутится пока s не станет равным N
    // после того как flag стал равным true происходит только одна проверка
    s += 1; // O(1) * 1 + ... O(1) * 6 + O(N - 6) * 7 = O(N)
        if s % 7 == 0 { // O(1) * 1 + ... O(1) * 6 + 7 * O(N - 6) = O(N)
        flag = true; // O(N) //как только flag становится равным true, НА ЭТОЙ
ИТЕРАЦИИ
        // больше в цикл мы не заходим
    }
    }
}
```

Теперь значение flag обновляется на каждой итерации внешнего цикла. Поэтому тело внутреннего цикла выполняется на каждой итерации до тех пор, пока s не станет равным 7. Потому за все время работы каждой из строк внутреннего тела выполнится $7 * O(N) = O(N)$ раз. Суммарно представленный фрагмент кода будет работать за

$$O(1) + O(N) + O(N) + O(N) + O(N) + O(N) + O(N) = O(N)$$

Пример 15

```
N: uint = input(); // O(1)

for i in range(1, N + 1){ // O(N)
    flag = false; // O(N)
    s = 0; // O(N)
    while s != i and not flag { // O(1) * 2 + ... O(1) * 7 + O(1) * (N - 6) * 8
= O(N)
        // пока flag = false цикл крутится пока s не станет равным N
    // после того как flag стал равным true происходит только одна проверка
    s += 1; // O(1) * 1 + ... O(1) * 6 + O(N - 6) * 7 = O(N)
        if s % 7 == 0 { // O(1) * 1 + ... O(1) * 6 + 7 * O(N - 6) = O(N)
        flag = true; // O(N) //как только flag становится равным true, НА ЭТОЙ
ИТЕРАЦИИ
        // больше в цикл мы не заходим
    }

    if s == 20 { // O(1) * 1 + ... O(1) * 6 + O(N - 6) * 7 = O(N)
        // сравнение будет выполняться только по flag не станет равным true
        // дальше в цикл не заходим => и сравнение не выполняется
        flag = false; // 0, мы сюда никогда не придем, эта строчка - мертвый код
    }
    }
}
```

Данный пример отличается от предыдущего тем, что мы добавили условие установки переменной flag в прежнее значение (false). Однако из-за того, что мы не заходим во внутренний цикл после s = 20, этот код никогда не выполнится. Такой код называется **мертвым кодом**. В первом приближении он никак не сказывается на времени работы программы.

$$O(1) + O(N) + O(N) + O(N) + O(N) + O(N) + O(N) = O(N)$$

При этом во втором приближении большое количество мертвого кода может сказаться на времени работы программы, потому в компилируемых языках существуют методы нахождения участков мертвого кода и их удаления.

Пример 16

```
N: uint = input(); // O(1)

s = 0;
for i in range(0, N) { // O(N)
    for j in range(0, N) { // O(N) * N = O(N^2)
        for k in range(0, N) { // O(N) * N * N = O(N^3)
        s += 1 // O(1) * N * N * N = O(N ^ 3)
    }
    }
}
```

Аналогично примеру 9, в данном случае тело самого внутреннего цикла будет исполнено N * N * N раз, итоговое время работы цикла:

$$O(1) + O(N) + O(N^2) + O(N^3) + O(N^3) = O(N^3)$$

Пример 17

```
N: uint = input(); // O(1)

s = 0; // O(1)
for i in range(0, N) { // O(N)
    for j in range(0, i) { // O(N) * N = O(i^2)
        for k in range(0, i) { // O(N) * N * N = O(i^3)
            s += 1 // от i=0 до N сумма O(i^2)
        }
    }
}
```

В этом случае мы можем знаем, что два внутренних цикла исполняются для каждого i за $O(i^2)$ Можно прикинуть, что в среднем на каждой итерации внешнего цикла внутренние исполняется за $O((N/2)^2) = O(N^2)$. Тогда в целом получим время

$$O(N^2) \cdot N = O(N^3)$$