УДК 510.52
DOI: 10.21122/2309-4923-2023-4-14-19

БЕККЕР И.А., ЯКИМОВ Е.А., СКРЫЛЕВ Н.П.

ВЫЧИСЛЕНИЕ ТРУДОЕМКОСТИ АЛГОРИТМА, РЕАЛИЗОВАННОГО НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ C#

Межгосударственное образовательное учреждение высшего образования
«Белорусско-Российский университет»
г. Могилев, Республика Беларусь

Под трудоемкостью алгоритма понимают количество выполняемых алгоритмом элементарных операций (шагов) как функцию от исходных данных, авторы предлагают методику вычисления этой меры сложности алгоритма рассматривать с учетом особенностей и отличий языков программирования. В работе приводятся формулы подсчета теоретической трудоемкости и правила вычисления экспериментальной трудоемкости программного кода на языке высокого уровня C#.

Ключевые слова: *трудоемкость алгоритма, трудоемкость программы, язык программирования C#, элементарные операции, трудоемкость конструкций языка*

Введение

Трудоёмкость алгоритма можно вычислять теоретически по формулам и экспериментально при выполнении программы через счётчик трудоёмкости, который суммирует все операции, относящиеся к элементарным. Договорённость о том, какие операции считать элементарными, объективно зависит от выбора языка программирования, задающего допустимые операции с разными типами данных, организацию работы с памятью и структурами данных.

Основные элементарные операции в языке программирования C#

Основными общепринятыми элементарными операциями программного кода являются присваивание, все арифметические и логические операции языка программирования, допустимые операции сравнения и одномерная индексация, а также вызов свойства, метода, оператор перехода [1, 2].

Присваивание считается за одну элементарную операцию и при работе с типом значения, и при работе со ссылочным типом. С помощью присваивания можно дать переменной новое значение, одновременно вычисляя значение справа стоящего выражения: в языке C# предусмотрено присваивание по типу +=, -=, *=, /=, %=. В этих случаях имеет место быть всего лишь сокращенная форма записи выражения с присваиванием, и для команды sum += 2*a[i]; следует для подсчета трудоемкости рассмотреть развернутую форму присваивания: sum = sum + 2*a[i]; что даст в итоге 4 элементарных операции.

Таким образом, с учетом описанных договоренностей, трудоемкость инструкции кода

value += *Expression*; (1)

будет равна $2 + F_{Expression}$, здесь $F_{Expression}$ – трудоемкость выражения *Expression* в правой части формулы (1). Аналогично следует подходить к подсчету трудоемкости присваиваний типа -=, *=, /=, %=.

В языке C# используются стандартные арифметические операции: «сложение» +, «вычитание» -, «умножение» *, «деление» /, «нахождение остатка от деления» %, также введены «инкремент» ++ и «декремент» – префиксные и постфиксные, унарные отрицание и сложение [3]. Инкремент и декремент представляют собой сокращённую запись присваивания, но выполняются процессором довольно быстро и могут сравниваться по времени выполнения с одной элементарной операцией. Таким образом, все перечисленные арифметические операции можно считать элементарными для языка C#. Возведение в степень, извлечение корня, многие другие математические операции выполняются через вызов методов в классе Math, их трудоемкость высчитывается как вызов метода с присваиванием значений параметрам.

C# предоставляет целый набор побитовых операторов, работающих поразрядно с числами в их двоичном представлении [3]: это & (логическое умножение), | (логическое сложение), ^ (логическое исключающее ИЛИ), ~ (логическое отрицание), >> и << (операции сдвига), – эти операции также стоит отнести к элементарным.

В языке программирования C# для работы с логическими операндами предусмотрены операции логического отрицания (!), логического умножения (&) и сложения (|), исключающего сложения (^), также есть сокращенные бинарные операторы && (условное логическое И) и || (условное логическое ИЛИ), которые вычисляют второй (правый) операнд только при необходимости [3]. Имеет смысл считать каждую из них за одну элементарную операцию и учитывать ее при подсчете трудоемкости.

Все операции сравнения «равно» ==, «не равно» !=, «больше» >, «больше либо равно» >=, «меньше» <, «меньше либо равно» <= [3] согласно общему подходу относятся к элементарным.

В C# имеются операции преобразования и контроля типов (*int*), *typeof*, *is*, *as* – их трудоемкость можем считать равной единице с учетом их семантики и соответствия времени выполнения элементарному.

Общий подход к вычислению трудоемкости специфических операторов языка C#

Для создания объекта в языке C# используется оператор new, вызывающий конструктор. Его трудоемкость можно оценить с точки зрения действий в памяти, однако аналогичные действия по объявлению переменной к элементарным операциям относить не принято. В то же время вызов конструктора влечет за собой присваивание параметрам конструктора значений и другие действия, выполняемые этим методом, которые имеет смысл подсчитывать в соответствии с общей методикой подсчета трудоемкости кода, написанного на языке C#.

Все операторы перехода, имеющиеся в C# (*goto*, *break*, *continue*, *return*), будем принимать за элементарные операции трудоемкости.

Методика подсчета трудоемкости программы на языке C# и ее отдельных блоков

В анализе алгоритмов точное значение функции трудоёмкости используется не всегда [4], чаще применяют такое понятие, как степень роста функции, показывающее ее поведение при стремлении размеров n входа к бесконечности.

Степень роста функции трудоемкости математически описывается через O-нотацию [1, 2, 5] и имеет вид $O(n^2)$ для функции, которую можно мажорировать квадратичной функцией, игнорируя мультипликативную константу и пренебрегая младшими термами [6].

Однако квадратичная функция $f_1(n) = a_0n^2 + a_1n + a^2$ при некоторых значениях коэффициентов может дать значения большие, чем функция $f_2(n) = n^3$ (на каком-то интервале области определения).

И для конкретного диапазона входных данных остается актуальным выбор предпочтительного алгоритма (например, сортировки), для чего важную роль сыграют не асимптотические, а точные оценки [7, 8].

При подсчете трудоемкости $F_{Program}$ программы ее разбивают на последовательно идущие блоки и суммируют их трудоемкости [1]:

$$F_{Program}(n) = F_1(n) + F_2(n) + \ldots + F_k(n), \quad (2)$$

где $F_i(n)$ – трудоемкость *i*-го блока кода из имеющихся *k*.

Основными блоками алгоритма являются ветвление *If* и циклы *For*, *While*, причем именно циклы являются в подавляющем большинстве случаев основными источниками ресурсных затрат при выполнении алгоритмов, наряду с рекурсивными методами.

Теоретическая трудоемкость конструкции *If* без учета переходов вычисляется [1] формулой:

$$F_{If}(n) = f_{Condition} + f_{True} \cdot p_{True} + f_{False} \cdot p_{False}, \quad (3)$$

где $f_{Condition}$ – трудоемкость условия, f_{True} и f_{False} – трудоемкости ветвей, p_{True} и p_{False} – вероятности попадания на ветви блока условия, $p_{True} + p_{False} = 1$. Для конструкции *If* теоретически подсчитанное и полученное экспериментально значения трудоемкости могут отличаться [9].

В языке C# есть встроенный в конструкцию *If* дополнительный блок *Else If* проверки условия, его трудоемкость экспериментально может вычисляется, например, следующим образом (листинг 1):

Листинг 1

```
trud++;// проверка a==5 выполнится в любом случае
 if (a==5)
   { … //тело BodyThen блока Then
   trud += f_BodyThen;}
 else {trud++; if (a<5)
                {trud++;// проверка a<5
                … //тело BodyElseIf блока ElseIf
                trud += f_BodyElseIf;}
               else
                {… //тело BodyElse блока Else
                trud += f_BodyElse;
                trud++;}// проверка a<5, когда a>5
                }
 }
```

Трудоемкость конструкции *For* зависит от количества итераций *N* цикла, способа изменения счетчика и в классической формуле анализа алгоритмов [1] описывается как:

$$F_{For}(n) = (1 + 3N) + N \cdot f_{Loop}, \quad (4)$$

где f_{Loop} – трудоемкость тела цикла, что автор обосновывает сведением к конструкции *Do While* [1].

Первые два слагаемых, очевидно, считают операции со счетчиком: присваивание, сравнение, изменение значения счетчика арифметической операцией.

Трудоемкость аналогичной конструкции **for**(int i = 0;i < N;i = i + 1) в *C#* с учетом количества *N* повторов цикла и трудоемкости тела f_{Loop} опишем формулой:

$$F_{For}C\#(n) = (2 + 3N) + N \cdot f_{Loop}, \quad (5)$$

для обоснования которой предлагаем использовать изображение последовательности действий со счетчиком цикла (рисунок 1) по аналогии с таблицами прокрутки индекса [10].

Рисунок 1. Операции со счетчиком

Эффективность предлагаемого представления действий со счетчиком цикла *For* заключается в том, что такая группировка команд и операций позволяет легко посчитать их количество, разбив на два слагаемых – зависящее и независящее от числа N повторов цикла.

Формула (5) описывает трудоемкость универсального цикла *For* вне зависимости от языка программирования. Она, в отличие от формулы (4), вычисляет количество элементарных операций этой конструкции с учетом подхода к определению элементарной операции, приведенного в источнике [1], и учитывает, что цикл *For* выполняет проверку условия на вход в тело цикла до инструкций в цикле, а *Do While* после. В случае использования инкремента i++ для изменения счетчика формула (5) примет вид:

$$F_{For}C\#(n) = (2 + 2N) + N \cdot f_{Loop}. \quad (6)$$

При экспериментально вычисляемой трудоемкости целочисленный счётчик встраивается в тело цикла и условия без прямого ввода формул трудоёмкости (3) и (5) блоков в код (листинг 2).

Листинг 2

```
trud += 2;//2 операции со счетчиком следует считать вне цикла
for (int i = 1; i <= n; i++)
    {trud += 2;//2 операции со счетчиком на каждой итерации цикла
        … //тело цикла с трудоемкостью fLoop
        trud += fLoop; } //учитываются операции в теле цикла
```

Начальное и предельное значения счетчика цикла задают по-разному, а встраивание счетчика трудоемкости в цикл позволяет, в отличие от подсчета теоретической трудоемкости, не вычислять количество итераций цикла (листинг 3, рисунок 2).

Листинг 3

```
trud += 3; //3 операции не зависят от числа повторов цикла
for (int i = 0; i <= n - 1; i++)
      { trud += 3;
    … тело Loop цикла
      trud += fLoop;}
```

Рисунок 2. Выполнение операций со счетчиком цикла *For*

Теоретически трудоемкость цикла **for** (int i = 0; i <= n - 1; i++) вычисляется по формуле:

$$F_{ForC\#\_Extended1}(n) = (3 + 3N) + N \cdot f_{Loop}. \quad (7)$$

Значения счетчика цикла могут быть описаны с использованием свойства *Length* или метода *GetLength* класса *Array*, вызов свойства и метода будем считать за одну элементарную операцию, вызов метода будем оценивать вдобавок по количеству входных параметров (операции присваивания), и оператору перехода *return* (в явном/неявном виде).

Для цикла **for**(int i = 0; i < array.GetLength(0); i++) теоретическая трудоемкость будет вычисляться по формуле (8) (последовательность и количество операций со счетчиком изображены на рисунке 3):

$$F_{ForC\#\_Extended2}(n) = (5 + 5N) + N \cdot f_{Loop}. \quad (8)$$

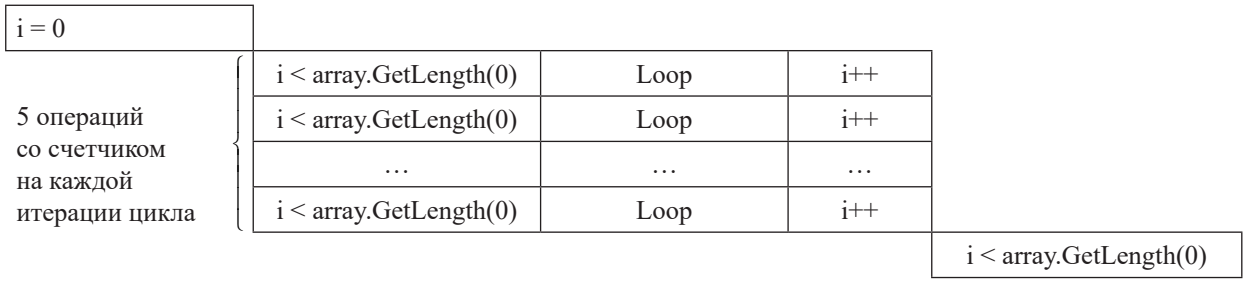

Рисунок 3. Операции со счетчиком с использованием метода

Рассмотрим конструкцию цикла *While* (листинг 4), в ней операции в выражении условия выполнятся фактически каждый раз при попадании в тело цикла и еще один, последний раз, когда выражение в условии станет равным *False*.

Трудоемкость тела встроенного метода, например, *Pow* не может быть вычислена в силу его непрозрачности, хотя играет роль в плане затрат ресурсов.

Листинг 4

```
while (Math.Pow(2, exponent) < N)
  {trud += 5;// трудоемкость условия без учёта тела Math.Pow
    … тело Loop цикла
    trud += f_Loop;}
trud += 5; // добавляем трудоемкость условия
```

Для циклов *While*, *Do While* нельзя вывести общую теоретическую формулу трудоемкости по причине невозможности описать в общем случае количество их выполнений. Но при сведении *While*, *Do While* к арифметическому циклу добавлением счетчика можно подсчитать трудоемкость теоретически за счет известного количества *N* итераций цикла:

$$F_{WhileWithCounter}(n) = (N + 1) \cdot f_{ConditionWhile} + N \cdot f_{Loop}, \quad (9)$$

где $f_{ConditionWhile}$ – трудоемкость условия цикла *While*, f_{Loop} – трудоемкость тела цикла.

$$F_{DoWhileWithCounter}(n) = N \cdot f_{ConditionDoWhile} + N \cdot f_{Loop}, \quad (10)$$

$f_{ConditionDoWhile}$ – трудоемкость условия цикла D*o While*, f_{Loop} – трудоемкость тела цикла *Do While*.

В конструкции *Do While* условие проверится каждый раз после прохождения тела цикла, в последнюю итерацию цикла выражение в условии станет равным *False* (листинг 5).

Листинг 5

```
do {trud += f_ConditionDoWhile;//учитывается трудоемкость условия цикла
        … тело Loop цикла
        trud += f_Loop;}
while (ConditionDoWhile);
```

Рассмотрим конструкцию *Do While* со счетчиком (листинг 6), ее трудоемкость отличается от трудоемкости цикла *For*, описываемой формулой (6), на одну элементарную операцию – последнюю проверку после выполнения всех итераций цикла.

Листинг 6

```
i = 1;
do { … тело Loop цикла
     i = i + 1;}
while (i <= n);//при достижении значения False выражением выполняется еще 1 операция сравнения, которую необходимо учесть:
```

$$F_{DoWhileWithCounter1}(n) = 1 + 3N + N \cdot f_{Loop}. \quad (11)$$

Следовательно, неправомерно при анализе цикла *For* сводить его к неэквивалентной по трудоемкости конструкции *Do While* со счетчиком.

Заключение

При вычислении трудоемкости алгоритма, реализованного на языке программирования высокого уровня *C#*, следует учитывать такие особенности языка, как наличие специфических операций, операторов, методов-конструкторов, блочных конструкций языка, применяя к ним обоснованные выше методики, основанные на общепризнанных для структурных языков программирования подходах учета команд языка низкого уровня типа Ассемблер.

В работе сформулирована методика нахождения трудоёмкости основных блоков программы на языке *C#*, для цикла *For* предложено табличное представление действий со счетчиком, позволяющее корректно описывать формулами их количество. Это целое семейство формул типа (5), (6), (7), (8), конкретные коэффициенты для первых двух слагаемых зависят от задания начального, предельного значения счетчика и правил его изменения. Показана неэквивалентность по трудоемкости универсальных конструкций *For* (6) и *Do While* (11), без привязки к конкретному языку программирования.

Приведены приемы нахождения экспериментальной трудоемкости основных конструкций языка *C#*.

ЛИТЕРАТУРА

1. **Ульянов, М.В.** Ресурсно-эффективные компьютерные алгоритмы. Разработка и анализ / М.В. Ульянов. – М.: ФИЗМАТЛИТ, 2008. – 304 с.

2. **Фофанов, О.Б.** Алгоритмы и структуры данных: учебное пособие / О.Б. Фофанов. – Томск: Изд-во Томского политехнического университета, 2014. – 126 с.

3. C# documentation: C# operators and expressions [Электронный ресурс]. Режим доступа: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/ Дата доступа: 08.08.2023.

4. **Kreinovich, V.** Among several successful algorithms, simpler ones usually work better: a possible explanation of an empirical observation // V. Kreinovich, O. Kosheleva // Mathematical Structures and Modeling. – 2015. – No. 1(33). – Pp. 50–55.

5. **Mala, F.A.** The Big-O of Mathematics and Computer Science / Firdous Ahmad Mala // Journal of Applied Mathematics and Computation. – 2022. – № 6 (1). – Pp. 1–3.

6. **Licht, B.** Obstacles in Learning Algorithm Run-time Complexity Analysis / Licht Bailey // University of Nebraska at Omaha. – 2022. – Theses/Capstones/Creative Projects.193

7. **Цветков, В.Я.** Сложность алгоритмов первого рода / В.Я. Цветков // Образовательные ресурсы и технологии. – 2020. – № 4 (33). – С.73–80.

8. **Самуйлов, С.В.** Методика сравнительного анализа алгоритмов на примере алгоритмов последовательного поиска / С.В. Самуйлов // Научно-методический электронный журнал «Концепт». – 2014. – № 9 (сентябрь). – С. 46–50.

9. **Выборнов, А.Н.** Операционная чувствительность алгоритмов / Выборнов А.Н., Головешкин В.А., Ульянов М.В. // Автоматизация и современные технологии. – 2015. – № 8. – С. 41–46.

10. **Rublev, V.S.** Automated System for Teaching Computational Complexity of Algorithms Course / Rublev V.S., Yusufov M.T. // Modeling and Analysis of Information Systems. – 2017. – Vol. 24, № 4. – Pp. 481–495.

REFERENCES

1. **Ulyanov, M.V.** Resource-efficient computer algorithms. Development and analysis. Moscow: Fizmatlit, 2008, 304 p.

2. **Fofanov, O.B.** Algorithms and data structures: textbook. Tomsk: Publishing house of Tomsk Polytechnic University, 2014, 126 p.

3. **C# documentation: C# operators and expressions** [Electronic resource]. Access mode: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/. – Date of access 08.08.2023.

4. **Kreinovich V., Kosheleva O.** Among several successful algorithms, simpler ones usually work better: a possible explanation of an empirical observation. Mathematical Structures and Modeling. 2015. N. 1(33). P. 50–55.

5. **Mala, F.A.** The Big-O of Mathematics and Computer Science. Journal of Applied Mathematics and Computation. 2022. № 6 (1). Pp. 1–3.

6. **Licht, B.** Obstacles in Learning Algorithm Run-time Complexity Analysis. University of Nebraska at Omaha. 2022. Theses/Capstones/Creative Projects. 193 p.

7. **Tsvetkov, V.Ya.** Complexity of algorithms of the first kind. Educational Resources and Technologies. 2020. № 4(33). Pp. 73–80.

8. **Samuilov, S.V.** Methodology of comparative analysis of algorithms by the example of sequential search algorithms. Scientific and methodical electronic journal "Concept". 2014. No. 9. Pp. 46–50.

9. **Vybornov A.N., Goloveshkin V.A., Ulyanov M.V.** Operational sensitivity of algorithms. Automation and Modern Technologies. 2015. № 8. Pp. 41–46.

10. **Rublev V.S., Yusufov M.T.** Automated System for Teaching Computational Complexity of Algorithms Course. Modeling and Analysis of Information Systems. 2017. Vol. 24, № 4. Pp. 481–495.

BECKER I.A., YAKIMOV E.A., SKRYLYOV N.P.

ALCULATING THE COMPLEXITY OF AN ALGORITHM IMPLEMENTED IN THE C# PROGRAMMING LANGUAGE

Belarusian-Russian University
Mogilev, Republic of Belarus

Time complexity of an algorithm is the number of elementary operations performed by the algorithm. Taking into account the features of programming languages, the authors propose to consider the methodology for calculating this measure of algorithm complexity in the specific language of its implementation, provide formulas for calculating theoretical complexity and the rules for calculating the experimental complexity of program in C#.

Keywords: *algorithm time complexity, program time complexity, C# programming language, elementary operations, language construction time complexity*

Беккер Инга Александровна, старший преподаватель, кафедра Автоматизированные системы управления, Белорусско-Российский университет, г. Могилев, ул. Мовчанского, д. 1, кв. 60.

Тел.: +375 447 318 464

E-mail: binga@rambler.ru

Якимов Евгений Анатольевич, к.т.н., доцент, кафедра Автоматизированные системы управления, Белорусско-Российский университет, г. Могилев, ул. 30 лет Победы, д. 22, кв. 158,

Тел.: +375 293 124 194

E-mail: e-soft@bk.ru

Скрылёв Никита Петрович, ассистент, кафедра Автоматизированные системы управления, Белорусско-Российский университет, г. Могилев, ул. Челюскинцев, д. 138, кв. 83

Тел.: +375 296 504 204

E-mail: mniccita@gmail.com