**РАЗДЕЛ 5. АНАЛИЗ СЛОЖНОСТИ АЛГОРИТМОВ ПО ВРЕМЕНИ И ПРОСТРАНСТВУ/ СТАБИЛЬНОСТЬ АЛГОРИТМОВ**

**5.1. Свойства алгоритмов**

В предыдущих разделах основное внимание уделялось описанию наиболее распространенных структур данных, с одной стороны как абстрактных сущностей, а, с другой – их конкретную реализацию в языке программирования Golang. В этом разделе речь пойдет о теоретическом анализе алгоритмов обработки структур данных с позиций оценки затрат времени на выполнение программного кода и занимаемого объема компьютерной памяти.

В теории и на практике алгоритмы обработки структур данных определяется как набор последовательно выполняемых, устойчивых процедур, обеспечивающих достижение конечного вычислительного результата, исходя из фиксированного и ограниченного набора исходных данных. Большинство алгоритмов обработки данных включают в себя такие базовые процедуры обработки данных как *поиск элементов, их сортировка, добавление, обновление, удаление и т.д.*

Прежде всего следует отметить, что «хороший алгоритм», в смысле способности получения конечного результата за приемлемое время, используя приемлемый объем памяти, должен обладать следующими основными характеристиками: *корректность и конечность*,

*Корректность* означает, что если **алгоритм** создан для решения определенной задачи, то для всех исходных данных он должен всегда давать правильный результат и ни для каких исходных данных не будет получен неправильный результат.

*Конечность* алгоритма означает, что**для получения результата нужно выполнить конечное число вычислительных операций**, т. е. исполнение программного кода в некоторый момент времени должен прекратиться.

После обеспечения *корректности* и *конечности* алгоритма важнейшей его характеристикой является *эффективность*, достижение которой обеспечиваются выполнением следующих требований:

1. Алгоритм должен эффективно использовать ресурсы, доступные системе;
2. Вычислительное время (время, затрачиваемое на генерацию выхода, соответствующего конкретному входу) должно быть минимальным;
3. Память, используемой алгоритмом, также должна быть как можно меньше.

В большинстве вычислительных задач необходимо стремиться к компромиссу между вычислительным временем и занимаемой памятью. Иными словами, при выборе метода решения вычислительной задачи нужно определиться с приоритетами: что важнее – вычислительное время или объем занимаемой памяти. Может показаться, что для современных компьютеров это не так уж и важно. Однако при решении сложных практических задач, связанных с обработкой огромных массивов данных, например, для задач искусственного интеллекта или для задач биоинформатики (например, расшифровка генома) поиск оптимального сочетания вычислительного времени и компьютерной памяти становится определяющим В связи с этим будем определять термин "сложность алгоритма» как мера количества времени и/или пространства, необходимого для решения задачи в зависимости от размера данных.

Такой анализ включает в себя определение функции, которая связывает длину входных данных алгоритма с временем, которое он затрачивает (его временнáя сложность) или количеством мест хранения, которые он использует (сложность пространства). Алгоритм считается эффективным, когда значения этой функции либо ничтожны либо медленно растут относительно роста размера входных данных.

5.2. Теоретический анализ сложности алгоритмов

В конечном счете созданные алгоритмы должны быть реализованы на каком-либо вычислительном устройстве с помощью сгенерированного программного кода. Поэтому следует различать сложность самого алгоритма и сложность его компьютерной реализации. Самый быстрый алгоритм, реализованный на медленном вычислительном устройстве может оказаться менее эффективным, чем не очень удачный алгоритм, реализованный на компьютере, обладаюшим большей вычислительной мощностью или языком программирования, способным выполнять параллельные вычисления.

Таким образом при оценке сложности алгоритмов проводят на абстрактной машине с произвольным доступом к памяти, что позволяет не учитывать низкоуровневые параметры вычислительного устройства (размер памяти процессора, многозадачность и т.д.). Модель такой машины состоит из памяти и процессора, которые работают следующим образом [https://pro-prof.com/archives/1660]:

* память состоит из ячеек, каждая из которых имеет адрес и может хранить один элемент данных;
* каждое обращение к памяти занимает одну единицу времени, независимо от номера адресуемой ячейки;
* количество памяти достаточно для выполнения любого алгоритма;
* процессор выполняет любую элементарную операцию за один временной шаг;
* циклы и функции не считаются элементарными операциями.

Трудоемкость алгоритма Tn, связанной с оценками его сложности определяется подсчетом количества выполняемых операций. Для примера рассмотрим алгоритм поиска максимального элемента массива.

package main

import "fmt"

func main() {

 array:= []int{11, 9, 17, 45, 411}

N := len(array)

 maxNumber := array[0]

 for item:= 1; item < N; item++ {

    if array[item] > maxNumber {

    maxNumber = array[item]

  }

 }

 fmt.Println("Наименьшее значение элемента масссива = ", maxNumber)

}

При реализации этого алгоритма будут выполнены:

1. (N — 1) операция присваивания счетчику цикла i нового значения;
2. (N — 1) операция сравнения счетчика со значением N;
3. (N — 1) операция сравнения элемента массива со значением maxNumber;
4. от 1 до N операций присваивания значения переменной maxNumber.

Является очевидным, что такой подсчет для определения трудоемкости алгоритма и оценки его сложности является индивидуальным и зависит от размера входных данных. Поэтому совершенно естественно возникло предложение о качественной оценке сложности: в зависимости от «качества» исходных данных: *наилучшего* варианта, н*аихудщего* и *среднего*.

5.3. Качественная оценка сложности

Для качественной оценки сложности алгоритмов используются следующие асимптотические обозначения: *омега-нотация, тета-нотация*, и *нотация Big-O*,отражающие три границы времени работы алгоритма (нижнюю, среднюю и верхнюю). *Омега-нотация* представляет собой нижнюю границу времени работы алгоритма, что характеризует наилучшую сложность алгоритма в смысле достижения эффективности. Для любого значения  nисходных данных минимальное время, требуемое алгоритмом, задается обозначением Ω(f(n)).

Нотация *Big-O* представляет верхнюю границу времени выполнения алгоритма, которая характеризует наихудшую сложность алгоритма в смысле эффективности. В большинстве практических случаях именно эта нотация используется для оценки эффективности алгоритмов, поскольку разработчик программного обеспечения должен оценивать именно наихудший сценарий вычислительного процесса. Наконец, *тета-нотация (Θ-нотация)* представляет собой верхнюю и нижнюю границу времени выполнения алгоритмов и используется для их анализа средней сложности. Рассмотрим более подробно нотацию *Big O*, определяющую верхнюю границу любого алгоритма, то есть алгоритм не может занять больше времени, чем время верхней границы.

Начнем с простейшего **алгоритма с постоянным временем** — это алгоритм, который требует одинакового количества времени, независимо от его входных данных, сложность которого обозначается как O(1). Например, даны два числа, требуется вычислить сумму.

Далее следует **логарифмическая временнáя сложность** O(log n). Когда время, затрачиваемое алгоритмом на выполнение, пропорционально логарифму входного размера *n*, говорят, что он имеет логарифмическую временную сложность. В качестве примера можно привести простой подсчет числа операций печати индекса цикла:

func main() {

    n := 16

    for i:= 1; i < n; i=i\*2 {

        fmt.Println("i = ", i)

        }

    }

Напомним, что log216 = log224 = 4 log22 = 4, поскольку log22 = 1). Алгоритмы с логарифмической сложностью считаются высокоэффективными, поскольку отношение времени выполнения одной операции к размеру массива данных уменьшается с увеличением его размера. Алгоритмы, работающие за логарифмическое время, обычно встречаются при операциях с двоичными деревьями или при использовании двоичного поиска. Такие алгоритмы будут рассмотрены в следующем разделе.

**Линейная временная сложность (Оn)** характеризуеталгоритмы, время выполнения которых линейно зависит от размера входных данных (n). Простейшим примером может служить задача на вычисления суммы *n* чисел:

func main() {

    n := 16

    sum := 0

    for i:= 0; i < n; i++ {

        sum = sum + i

    }

    fmt.Println("sum =", sum)

}

Типичными примерами линейной временнóй сложности являются алгоритмы, связные с сортировкой массивов, что будет показано в следующем разделе. При этом важно заметить, что алгоритмам с линейной сложностью не требуется дополнительной памяти.

Далее следует **квадратичная сложность алгоритмов O(n2**), число операций в которых находится в квадратичной зависимости от размера входных данных. В качестве примера можно привести алгоритм умножения двух чисел в интервале [1:n;1:m]:

func main() {

    n := 3

    m := 5

    pow := 1

    for i:= 1; i < n; i++ {

        for j:= 1; j < m; j++ {

            pow = pow\*i\*j

        }

    }

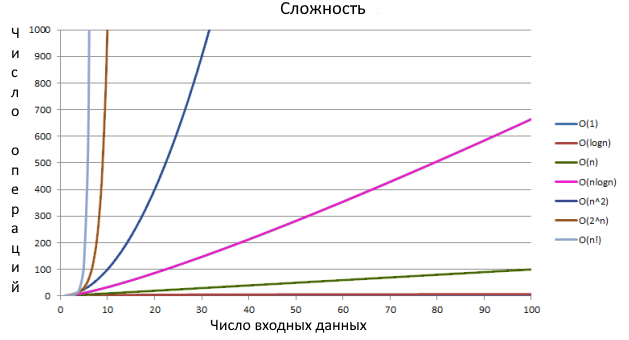
    fmt.Println("pow =", pow)

}

Сравнение рассмотренных сложностей представлено в виде таблицы. Табл. Сравнение общих сложностей

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| n | Константа О(1) | Логарифмическая  O(log n) | Линейная O(n) | Линейная логарифмическая  O(n log n) | Квадратичная O(n2) |
| 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 2 | 2 | 4 |
| 4 | 1 | 2 | 4 | 8 | 16 |
| 8 | 1 | 3 | 8 | 24 | 64 |
| 16 | 1 | 4 | 16 | 64 | 256 |

Или в виде графика (рис.5.1.).



Как видно из этой таблицы и графика, по мере увеличения сложности функции количество вычислений или времени, необходимого для выполнения функции, может значительно возрасти. Далее при описании алгоритмов будет представлены оценки их сложности.

Другая характеристика вычислительной сложности алгоритмов связана с пространственной сложностью, оцениваемой объемом памяти, необходимой алгоритму в его жизненном цикле. Это пространство состоит из фиксированной и переменной частей [Васильев]. Фиксированная часть, представляет собой пространство, необходимое для хранения простых переменных и констант, а также размера программы, которые не зависят от сложности задачи. Переменная часть — это пространство, требуемое переменными, размер которого полностью зависит от размера задачи. Например, пространство стека рекурсий, динамическое выделение памяти и т.д.

Пространственная сложность оценивается как О(1) в случае использования только входного массива данных. Если же для решения задачи потребуется вспомогательная память такого же размера как и входная память (память для хранения исходного массива), то пространственная сложность оценивается например, как О(n2).

Литература:

1. Васильев В. С. [Алгоритм. Свойства алгоритма](https://pro-prof.com/archives/578) [Электронный ресурс] – режим доступа: https://pro-prof.com/archives/578. Дата обращения: 06.01.2014.
2. Дж. Макконелл [Анализ алгоритмов](https://pro-prof.com/books-algorithms). Активный обучающий подход. — 3-е дополненное издание. М: Техносфера, 2009. -416с.