Лабораторная работа № 7

Тема: *Оценка сложности алгоритмов.*

Цель: *ознакомиться с понятием «сложность алгоритмов», изучить распространенные сложности алгоритмов, научиться применять полученные знания на практике.*

1 Краткая теория

1.1 ОБЩИЕ ПОНЯТИЯ.

Сложность алгоритмов. Основы.

Сложность алгоритма – это количественная характеристика, которая говорит о том, сколько времени, либо какой объём памяти потребуется для выполнения алгоритма.

Развитие технологий привело к тому, что память перестала быть критическим ресурсом. Поэтому, когда говорят об анализе сложности алгоритма, обычно подразумевают то, насколько быстро он работает.

Но ведь время выполнения алгоритма зависит от того, на каком устройстве его запустить. Один и тот же алгоритм, запущенный на разных устройствах, выполняется за разное время.

Тогда было предложено измерять сложность алгоритмов в элементарных шагах – то, сколько действий необходимо совершить для его выполнения. Любой алгоритм включает в себя определённое количество шагов и не важно на каком устройстве он будет запущен, количество шагов останется неизменным. Эту идею принято представлять в виде Big O (или O-нотации).

Big O показывает то, как сложность алгоритма растёт с увеличением входных данных. При этом она всегда показывает худший вариант развития событий - верхнюю границу.

Распространённые сложности алгоритмов.

Рассмотрим некоторые распространённые виды алгоритмов. Всё зависит от алгоритма, который вы оцениваете. Всегда может появится какая-то дополнительная переменная (не константа), которую необходимо будет учесть в функции Big O.

Константная - O(1).

Означает, что вычислительная сложность алгоритма не зависит от входных данных. Однако, это не значит, что алгоритм выполняется за одну операцию или требует очень мало времени. Это означает, что время не зависит от входных данных.

Пример № 1.

У нас есть массив из 5 чисел и нам надо получить первый элемент.

```
val nums = intArrayOf(1, 2, 3, 4, 5)
val firstNumber = nums[0]
```

Насколько возрастет количество операций при увеличении размера входных параметров?
Ни на сколько. Даже если массив будет состоять из 100, 1000 или 10 000 элементов нам все равно потребуется одна операция.

Пример № 2.

Сложение двух чисел. Функция всегда выполняет константное количество операций.

```
fun pairSum(a: Int, b: Int) = a + b
```

Пример № 3.

Размер массива. Опять же, функция всегда выполняет константной количество операций.

```
fun getSize(nums: List<Int>): Int = nums.size
```

Линейная - O(n).

Означает, что сложность алгоритма линейно растёт с увеличением входных данных. Другими словами, удвоение размера входных данных удвоит и необходимое время для выполнения алгоритма.

Такие алгоритмы легко узнать по наличию цикла по каждому элементу массива.

Пример № 1 - Рекурсивная функция.

```
fun sum(n: Int): Int {
  if (n == 1) return 1
  return n + sum(n - 1)
}
```

Пример № 2 - Линейная функция.

```
fun pairSumSequence(n: Int): Int {
  var sum = 0
  for (i in 0 until n) {
    sum += pairSum(i, i + 1)
  }
  return sum
}

fun pairSum(a: Int, b: Int) = a + b
```

Функция pairSumSequence() в цикле складывает какие-либо пары чисел, вызывая функцию pairSum(). Цикл выполняется от 0 до n. Т.е. чем больше n, тем

больше раз выполнится цикл. Поэтому сложность функции pairSumSequence() - O(n).

Пример № 3.

```
fun sum(nums: List<Int>) {
  var sum = 0
  var product = 1

  for(num in nums) {
    sum += num
  }

  for(num in nums) {
    product *= num
  }

  println(sum, product)
}
```

В функции два последовательных цикла for, каждый из которых проходит массив длиной n, следовательно сложность будет: O(n + n) = O(n)

Логарифмическая - O(log n).

Означает, что сложность алгоритма растёт логарифмически с увеличением входных данных. Другими словами это такой алгоритм, где на каждой итерации берётся половина элементов.

К алгоритмам с такой сложностью относятся алгоритмы типа “Разделяй и Властвуй” (Divide and Conquer), например бинарный поиск.

Линеарифметическая или линеаризованная - O(n * log n).

Означает, что удвоение размера входных данных увеличит время выполнения чуть более, чем вдвое.

Примеры алгоритмов с такой сложностью: Сортировка слиянием или множеством n элементов.

Квадратичная - $O(n^2)$, O(n^2).

Означает, что удвоение размера входных данных увеличивает время выполнения в 4 раза. Например, при увеличении данных в 10 раз, количество операций (и время выполнения) увеличится примерно в 100 раз. Если алгоритм имеет квадратичную сложность, то это повод пересмотреть необходимость использования данного алгоритма. Но иногда этого не избежать.

Такие алгоритмы легко узнать по вложенным циклам.

Пример № 1.

```
fun printPairs(nums: List<Int) {
  for (i in nums) {
    for (j in nums) {
      println(nums[i], nums[j])
    }
  }
}
```

В функции есть цикл в цикле, каждый из них проходит массив длиной n, следовательно сложность будет: $O(n * n) = O(n^2)$

Теория алгоритмов.

В теории алгоритмов НЕ рассматривают характеристики окружения, на котором выполняется программа. Учитывают только **входные данные**! От них будет зависеть количество элементарных действий. Важна тенденция на большом количестве входных данных.

При анализе алгоритма следует понимать, как меняется количество действий при увеличении количества входных данных.

Почему нам важно для больших входных данных? А все потому, что на маленьких данных большинство алгоритмов работают достаточно быстро, а вот с увеличением входных данных уже нужно смотреть и анализировать. Все в этом мире растет и приумножается: машины, железные дороги, книги, поэтому мы и исследуем то, насколько наш алгоритм сможет выдержать при потенциальном росте данных.

На самом деле расчет временной сложности можно рассматривать как некоторую **функцию**, которая принимаем **входные данные** и выдает **количество** элементарных операций, потраченный на обработку этих данных.

Рассмотрим следующую программу:

C#

```
static void Sum(int a, int b)
{
        int sum = a + b;
        Console.WriteLine(sum);
}
```

Python

```
def Sum (a, b):
        sum = a + b
        print(sum)
```

Посчитаем сколько здесь элементарных действий. В данной программе три элементарных действия: сложение, присваивание и вывод на консоль. Говорят, что данная программа занимает **константное время**, так как от увеличения входных данных *a* и *b*, количество элементарных действий не поменяется. И обозначают как $O(1)$ - говорят о большое от единицы. Почему от единицы? Потому как бы мы не увеличили входные данные, действий останется столько же. Эти действия **ничтожно малы и равны единице по сравнению с входными данными**.

Рассмотрим программу и посчитаем для него временную сложность:

C#

```
static void Print(int n)
{
        for (int i = 1; i <= n; i++)
        {
                Console.WriteLine(i);
        }
}
```

Python

```
def Print (n):
        for i in range(1, n + 1):
                print(i)
```

При n=5 алгоритм сделает 55 действий вывода на экран. Если мы n увеличим в 1010 раз, то количество действий также увеличится не более чем в 1010 раз.

Так вот, сложность алгоритма называется **линейной**, если при увеличении входных данных в n раз, количество элементарных действий алгоритма увеличится не более чем в nраз. Говорят, что алгоритм имеет **линейную сложность**, либо алгоритм работает за линию. Еще говорят, что алгоритм делает **не более** чем n-операций и записывает как $O(n)$ - говорят O большое от n.

Давайте все-таки считать точнее количество элементарных операций:

Задача 1.

C#

```
static void Print(int n)
{
        for (int i = 1; i <= n; i++)
        {
                Console.WriteLine(i);
        }
}
```

Python

```
def Print (n):
        for i in range(1, n + 1):
                print(i)
```

- 11 операция для **int i = 1**
- n+1 операций сравнения при **i <= n**
- 2n операций для **i++** (эквивалентно i = i + 1, а это две операции: **присваивание** и **сложение**)
- n операций для **Console.WriteLine(i)**

Итого: временная сложность равна $O(1+(n+1)+2n+n)=O(4n+2)$.

Задача 2.

C#

```
int n = Convert.ToInt32(Console.ReadLine());
int i = 0;
int count = 0;
while (i < n)
{
        for (int j = 0; j < n; j++)
        {
                count++;
        }
        i++;
}
```

Python

```
n = int(input())
i = 0
count = 0
while i < n:
        for j in range(n):
                count += 1
        i += 1
```

- 11 операция для **int i = 0**
- 11 операция для **int count = 0**
- n+1 операций сравнения при **i < n**
- 2n операций для **i++**
- n итераций цикла **for** и каждый раз:
- 11 операция для **int j = 0**
- n+1 операций сравнения при **j < n**
- 2n операций для **j++**
- 2n операций для **count++**

Итого: временная сложность равна $O(1+1+(n+1)+2n+n(1+(n+1)+2n+2n))=$ $=O(5n2+5n+3)$.

Согласитесь, что **даже для простых алгоритмов сложно посчитать точное количество элементарных операций.** Из-за этого ввели такое понятие как асимптотическая сложность с помощью которого вычисляется сложность алгоритмов. Если говорить по-простому, то она получается следующим образом:

1. отбросим в функции сложности все слагаемые, **кроме одного с самой быстрой скоростью роста;**
2. отбросим все константы.

Это и будет **асимптотической оценкой сложности.**

Зная теперь, что такое асимптотическая сложность, давайте посчитаем для примеров, которые были приведены выше:

1. для первой задачи временная сложность равна $O(4n+2)$. Оставим только слагаемое 4n, так как оно является с самой быстрой скоростью роста. А потом уберем константу 44, так как при увеличении n, константа 44 большой роли не играет. То есть, асимптотическая сложность будет равна $O(n)$.
2. для второй задачи временная сложность равна $O(5n2+5n+3)$. Оставим только слагаемое 5n2, так как оно является с самой быстрой скоростью роста. А потом уберем константу 55, так как при увеличении n, константа 55 большой роли не играет. То есть асимптотическая сложность будет равна $O(n2)$.

2.1 ШПАРГАЛКА ПО АСИМПТОТИЧЕСКОЙ СЛОЖНОСТИ АЛГОРИТМОВ.

Хорошо	Приемлемо	Плохо

Поиск

Алгоритм	Структура данных	Временная сложность		Сложность по памяти
		В среднем	В худшем	В худшем
Поиск в глубину (DFS)	Граф с |V| вершинами и |E| ребрами	-	O(|E| + |V|)	O(|V|)
Поиск в ширину (BFS)	Граф с |V| вершинами и |E| ребрами	-	O(|E| + |V|)	O(|V|)
Бинарный поиск	Отсортированный массив из n элементов	O(log(n))	O(log(n))	O(1)
Линейный поиск	Массив	O(n)	O(n)	O(1)
Кратчайшее расстояние по алгоритму Дейкстры используя двоичную кучу как очередь с приоритетом	Граф с |V| вершинами и |E| ребрами	O((|V| + |E|) log |V|)	O((|V| + |E|) log |V|)	O(|V|)
Кратчайшее расстояние по алгоритму Дейкстры используя массив как очередь с приоритетом	Граф с |V| вершинами и |E| ребрами	O(|V|^2)	O(|V|^2)	O(|V|)
Кратчайшее расстояние используя алгоритм Беллмана–Форда	Граф с |V| вершинами и |E| ребрами	O(|V||E|)	O(|V||E|)	O(|V|)

Сортировка

Алгоритм	Структура данных	Временная сложность			Вспомогательные данные
		Лучшее	В среднем	В худшем	В худшем
Быстрая сортировка	Массив	O(n log(n))	O(n log(n))	O(n^2)	O(n)
Сортировка слиянием	Массив	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Пирамидальная сортировка	Массив	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Пузырьковая сортировка	Массив	O(n)	O(n^2)	O(n^2)	O(1)
Сортировка вставками	Массив	O(n)	O(n^2)	O(n^2)	O(1)
Сортировка выбором	Массив	O(n^2)	O(n^2)	O(n^2)	O(1)
Блочная сортировка	Массив	O(n+k)	O(n+k)	O(n^2)	O(nk)
Поразрядная сортировка	Массив	O(nk)	O(nk)	O(nk)	O(n+k)

Структуры данных

Структура данных	Временная сложность								Сложность по памяти
	В среднем				В худшем				В худшем
	Индексация	Поиск	Вставка	Удаление	Индексация	Поиск	Вставка	Удаление	
Обычный массив	O(1)	O(n)	-	-	O(1)	O(n)	-	-	O(n)
Динамический массив	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Односвязный список	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Двусвязный список	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Список с пропусками	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Хеш таблица	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Бинарное дерево поиска	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Декартово дерево	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
Б-дерево	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Красно-черное дерево	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Расширяющееся дерево	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
АВЛ-дерево	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

Кучи

Куча	Временная сложность						
	Преобразование к куче	Поиск максимума	Извлечение максимума	Увеличить ключ	Вставить	Удалить	Слияние
Связный список (отсортированный)	-	O(1)	O(1)	O(n)	O(n)	O(1)	O(m+n)
Связный список (не отсортированный)	-	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)
Бинарная куча	O(n)	O(1)	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(m+n)
Биномиальная куча	-	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))
Фибоначчева куча	-	O(1)	O(log(n))	O(1)*	O(1)	O(log(n))*	O(1)

Представление данных

Пусть дан граф с |V| вершинами и |E| ребрами, тогда

Способ представления	Память	Добавление вершины	Добавление ребра	Удаление вершины	Удаление ребра	Проверка смежности вершин
Список смежности	O(|E| + |V|)	O(1)	O(1)	O(|E| + |V|)	O(|E|)	O(|V|)
Список инцидентности	O(|E| + |V|)	O(1)	O(1)	O(|E|)	O(|E|)	O(|E|)
Матрица смежности	O(|V|^2)	O(|V|^2)	O(1)	O(|V|^2)	O(1)	O(1)
Матрица инцидентности	O(|V||E|)	O(|V||E|)	O(|V||E|)	O(|V||E|)	O(|V||E|)	O(|E|)

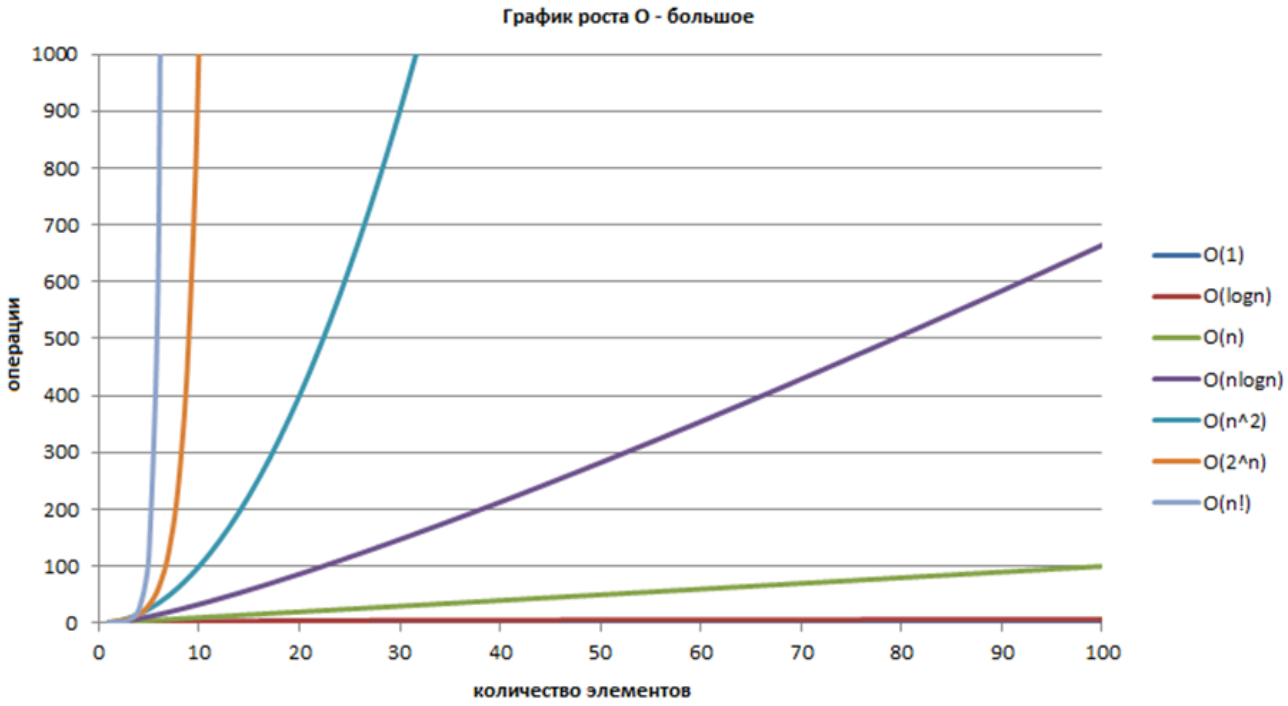
Нотация асимптотического роста

Обозначение	**Граница**	**Рост**
(Тета) Θ	Нижняя и верхняя границы, точная оценка	Равно
(O - большое) O	Верхняя граница, точная оценка неизвестна	Меньше или равно
(o - малое) o	Верхняя граница, не точная оценка	Меньше
(Омега - большое) Ω	Нижняя граница, точная оценка неизвестна	Больше или равно
(Омега - малое) ω	Нижняя граница, не точная оценка	Больше

1. (O — большое) — верхняя граница, в то время как (Омега — большое) — нижняя граница. Тета требует как (O — большое), так и (Омега — большое), поэтому она является точной оценкой (она должна быть ограничена как сверху, так и снизу). К примеру, алгоритм требующий Ω (n logn) требует не менее n logn времени, но верхняя граница не известна. Алгоритм требующий Θ (n logn) предпочтительнее потому, что он требует не менее n logn (Ω (n logn)) и не более чем n logn (O(n logn)).
2. f(x)=Θ(g(n)) означает, что f растет так же как и g когда n стремится к бесконечности. Другими словами, скорость роста f(x) асимптотически пропорциональна скорости роста g(n).
3. f(x)=O(g(n)). Здесь темпы роста не быстрее, чем g (n). O большое является наиболее полезной, поскольку представляет наихудший случай.

Алгоритм	**Эффективность**
o(n)	< n
O(n)	≤ n
Θ(n)	= n
Ω(n)	≥ n
ω(n)	> n

<u>График роста O – большое</u>



! Дополнительный материал !

1. Оценка сложности алгоритмов.
2. Анализ сложности алгоритмов, примеры.
3. Структуры данных, алгоритмы и сложность.
4. Оценка сложности алгоритмов. Сложность алгоритмов.

2 Задания.

Обязательное 1: *Оценить временную сложность алгоритма для следующей программы:*

C#

```
int n = Convert.ToInt32(Console.ReadLine());
int a = n / 100;
int b = n / 10 % 10;
int c = n % 10;
int revert = c * 100 + b * 10 + a;
Console.WriteLine(revert);
```

Python

```
n = int(input())
a = n // 100
b = n // 10 % 10
c = n % 10
revert = c * 100 + b * 10 + a
print(revert)
```

Обязательное 2: *Оценить временную сложность алгоритма для следующей программы:*

C#

```
int n = Convert.ToInt32(Console.ReadLine());
int count = 0;
for (int i = 0; i < n; i++)
{
        int number = Convert.ToInt32(Console.ReadLine());

        if (number % 10 == 0)
        {
                count = count + 1;
        }
}
Console.WriteLine(count);
```

Python

```
n = int(input())

count = 0
for i in range(n):
    number = int(input())
    if number % 10 == 0:
        count = count + 1

print(count)
```

Обязательное 3: *Оценить временную сложность алгоритма вычисления* m^m.

! Контрольные вопросы !

1. Определение понятия сложности алгоритма.
2. Описание константной сложности алгоритма.
3. Описание линейной сложности алгоритма.
4. Описание квадратичной сложности алгоритма.

Содержание отчёта

1. Ф.И.О., группа, название лабораторной работы.
2. Цель работы.
3. Описание проделанной работы.
4. Результаты выполнения лабораторной работы.
5. Ответы на контрольные вопросы.
6. Выводы.