Лабораторная работа №2.

Функция и порядок сложности алгоритма.

Цель работы: изучение методов теоретической и экспериментальной оценки функции сложности алгоритма.

Теоретические сведения

Большинство вычислительных задач, для решения которых используются компьютерные программы, могут быть решены несколькими способами, то есть для их решения могут применяться разные алгоритмы. В качестве примера приведем сортировку массива – известно несколько десятков способов упорядочивания элементов в массиве, которые могут отличаться друг от друга как концептуально, так и в незначительных деталях.

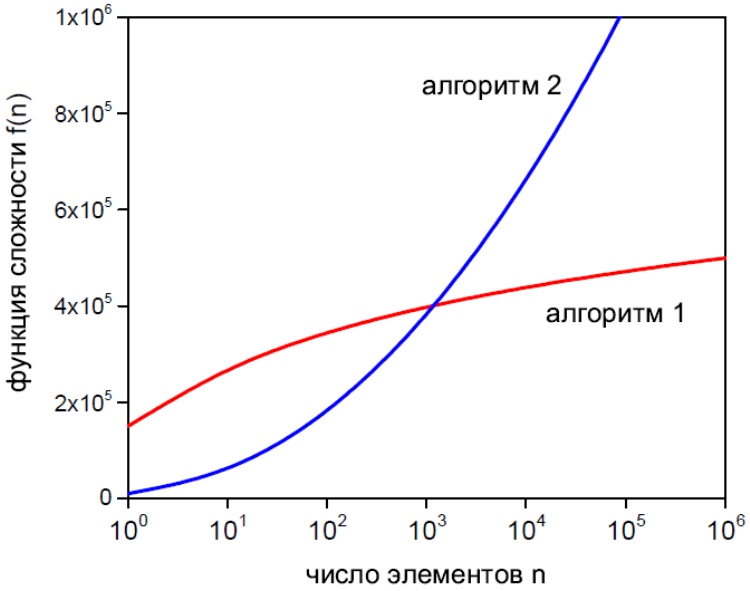
Для сравнения вычислительных алгоритмов между собой используют понятие *эффективности* алгоритма. Считается, что алгоритм 1 эффективнее алгоритма 2, если алгоритм 1 требует меньшего количества вычислительных операций (то есть, его программная реализация в общем случае работает быстрее), а также для него требуется меньший объем специфических ресурсов компьютера – оперативной памяти, дискового пространства и т.д.

Другой характеристикой алгоритма является его *сложность*. Говорят, что сложность алгоритма растет, если количество операций для решения задачи и/или объем требуемых ресурсов увеличивается. Таким образом, понятия эффективности и сложности являются связанными, но обратными по отношению друг к другу. В данной работе нас будет интересовать временна́я компонента сложности, связанная с количеством операций. Эту характеристику иногда также называют *алгоритмической* или *вычислительной сложностью*. Оценить вычислительную сложность алгоритма можно двумя способами: теоретически и экспериментально. Рассмотрим далее каждый из этих способов подробнее.

2.1. Функция сложности и порядок сложности.

Количественной характеристикой эффективности алгоритма служит **функции сложности $\boldsymbol{f(n)}$**. Эта функция связывает производительность вычислений с объемом обрабатываемых данных. Рассмотрим это понятие на следующем примере. Пусть для поиска максимального элемента в линейном целочисленном массиве, состоящем из n элементов, некоторому алгоритму требуется $3n + 8$ вычислительных операций.

Будем считать, что в это число входят операции сравнения, присвоения, увеличения счетчика цикла и т.д. Тогда функция сложности этого алгоритма может быть представлена в виде $f(n) = 3n + 8$.



Зная функции сложности двух алгоритмов, мы можем легко определить, какой из них предпочтителен с точки зрения скорости работы. Кроме того, вид функции $f(n)$ позволяет понять, как алгоритм ведет себя при различных n, например, с увеличением или уменьшением размеров массива. В качестве иллюстрации рассмотрим графики функций $f_1(n)$ и $f_2(n)$, показанные на рисунке выше. Для наглядности значения n отложены в логарифмическом масштабе. Как видно из рисунка, при небольших n (< 1000) алгоритм 2 оказывается эффективнее алгоритма 1, так как его кривая $f_2(n)$ расположена ниже кривой $f_1(n)$. Однако с увеличением объема входных данных ($n > 1000$) ситуация изменяется, и уже алгоритм 1 становится более эффективным.

Следует иметь в виду, что расчет функции сложности конкретного алгоритма в общем случае является непростой задачей, и для ее решения обычно применяются специальные методы анализа. В большинстве практических случаев для оценки используется другая (более простая) характеристика, которая называется **порядком сложности**.

Порядок сложности показывает асимптотическое поведение функции $f(n)$ при $n \to \infty$. Например, если точное выражение для функции $f(n)$ записано в виде $f(n) = 7n^2 + 3n + 8$, то при $n \to \infty$ в качестве грубой оценки сверху можно использовать выражение $g(n) = 8n^2$. Легко показать, что график функции $g(n)$ всегда будет выше графика $f(n)$ для достаточно больших n, то есть функция $g(n)$ – это действительно оценка нашего алгоритма «сверху».

Этот пример показывает основное правило вычисления порядка сложности алгоритма. Так как асимптотическое поведение функции на бесконечности определяется только слагаемыми с высшей степенью числа n, то мы можем отбросить (или с самого начала не учитывать) все остальные слагаемые в $f(n)$, как это было сделано выше. Более того, при анализе $f(n)$ не учитывается также и множитель перед высшей степенью n, в приведенном примере это число 8. То есть, считается, что функция сложности может быть оценена как $O(n^2)$, где O («о большое») – известный символ из математики. Про алгоритм $O(n^2)$ говорят, что он имеет второй порядок сложности, и называют его квадратичным.

Для вычисления $O\big(f(n)\big)$ используются следующие основные правила:

1. $O(kf) = O(f)$,
2. $O(f_1 \cdot f_2) = O(f_1) \cdot O(f_2)$,
3. $O(f_1 + f_2)$ равна доминанте $O(f_1)$ и $O(f_2)$.

Из 1-го правила следует, что константный множитель может не учитываться при определении порядка сложности, например

$$O(3.4n) = O(n), \qquad O(8n^2) = O(n^2).$$

Согласно 2-му правилу, сложность произведения двух функций равно произведению их сложностей, то есть их порядки складываются

$$O(3.4n \cdot 8n^2) = O(3.4n) \cdot O(8n^2) = O(n) \cdot O(n^2) = O(n^3).$$

И наконец, из 3-го правила следует, что сложность суммы двух функций равна сложности той функции, которая является максимальной (доминантной) из двух

$$O(n^3 + n) = O(n^3).$$

По определению, самыми эффективными алгоритмами считаются алгоритмы с **константной сложностью** или нулевым порядком, $O(1)$. Предполагается, что скорость работы такого алгоритма (практически) не зависит от объема входных данных. В качестве примера можно привести расчет среднего арифметического первого и последнего элемента массива (здесь число операций не зависит от n)

$$\tilde{A} = \frac{1}{2}(A[0] + A[n-1]).$$

Логарифмическая сложность $O\big(\log_2(n)\big)$ возникает в ситуациях, когда исходную задачу на каждом этапе удается разделить на 2 подзадачи, каждая из которых может быть решена отдельно. Бинарный поиск элемента в массиве, основанный на последовательном делении массива пополам, относится к алгоритмам такого типа.

Следующими по степени сложности идут алгоритмы 1 порядка, то есть алгоритмы с **линейной сложностью**, $O(n)$. Таким алгоритмам достаточно однократного перебора всех элементов для получения результата (например, поиск минимального элемента в массиве).

Далее выделяют **квадратичные** $O(n^2)$ алгоритмы, к которым, например, относятся многие методы сортировки, а также другие **полиномиальные** алгоритмы, $O(n^k)$. Наконец, одним из самых сложных и низкопроизводительных считается **экспоненциальный** алгоритм $O(2^n)$, который обычно возникает в случаях, когда для решения задачи используют многократный неинформативный перебор данных. Такие алгоритмы иногда называются методами «грубой силы».

2.2. Теоретическая оценка порядка сложности

В большинстве случаев порядок сложности алгоритма можно определить из анализа его программного кода. Для этого достаточно посмотреть на то, какие управляющие структуры используются в этом коде. Например, если код алгоритма не содержит рекурсивных вызовов функций и циклов, то мы сразу можем сделать вывод о том, что это алгоритм с константной сложностью, $O(1)$. Действительно, число операций в этом случае не зависит от n, а значит по свойству 1 такой алгоритм имеет нулевой порядок.

Таким образом, теоретическая оценка сложности алгоритмов в основном сводится к анализу циклов и рекурсивных вызовов. Рассмотрим несколько примеров. Пусть цикл обработки элементов некоторого массива имеет следующую структуру

```
for(int k = 0; k < n; k++)
{
    <операторы>
}
```

Независимо от того, сколько операторов находится в теле цикла, это число всегда фиксировано, поэтому блок операторов обладает константной сложностью $O(1)$. Этот блок повторяется в цикле n раз, поэтому сложность всего алгоритма можно оценить как $O(n)$, то есть линейную.

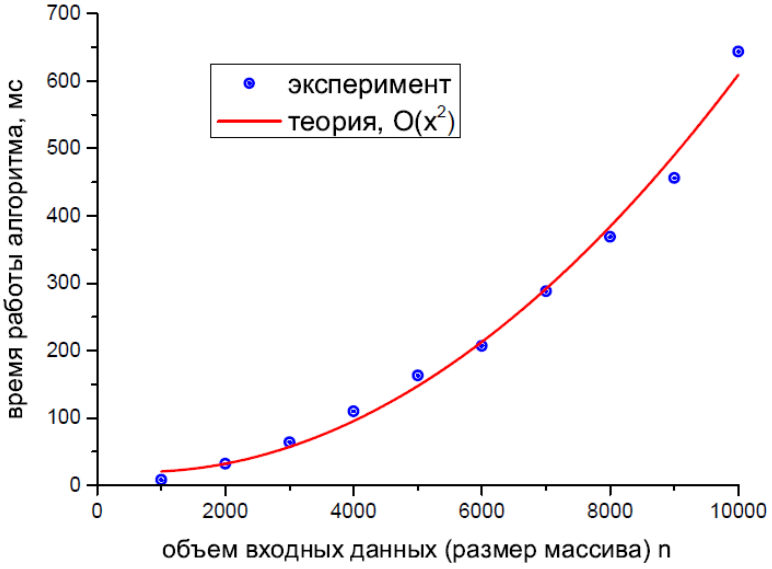
Если один из циклов вложен в другой, причем количество итераций в обоих циклах равно n, то такой алгоритм является квадратичным, $O(n^2)$

```
for(int i = 0; i < n; i++)
   for(int j = 0; j < n; j++)
   {
       <операторы>
   }
```

Аналогичным образом производится анализ и более сложных алгоритмов, в том числе, рекурсивных. Отметим, что код некоторых алгоритмов не всегда имеет такую простую структуру, как в приведенных листингах. Например, число итераций в цикле может быть меньше или больше n, а также зависеть от входных данных (как в некоторых методах сортировки). Для таких алгоритмов можно оценить усредненное количество операций, и таким образом определить порядок их сложности.

2.3. Экспериментальное измерение функции сложности.

Экспериментальный подход к анализу алгоритма основан на измерении производительности расчетов во временных единицах, например, в тактах процессора, секундах, миллисекундах и т.д. Конечно, время работы любой программы зависит от оборудования, на котором производятся расчеты; важными факторами также являются язык и среда разработки, и даже опыт и мастерство программиста. Однако, если два алгоритма реализованы в одной и той же программной среде, и далее запущены *с одинаковыми входными данными* на одном и том же оборудовании, то сравнение времен их работы может дать относительную оценку. Например, если время выполнения 1-го алгоритма в 10 раз меньше времени выполнения 2-го алгоритма (при равных прочих условиях), то мы можем говорить о том, что производительность алгоритма 1 в 10 раз выше, чем производительность алгоритма 2. Понятно, что речь в данном случае идет о *среднем времени*, так как в зависимости от входных данных полученное соотношение может отличаться, иногда значительно.



На данном рисунке показан пример экспериментального измерения функции сложности некоторого алгоритма. Точками на графике показаны результаты замеров

времени выполнения программы (например, в миллисекундах) для разных значений размера массива (параметр n). Все расчеты выполнены на одном и том же оборудовании, и с использованием одной программы. При проведении таких экспериментов рекомендуется делать несколько замеров времени для каждого значения n (например, 3-5 замеров), и откладывать на графике усредненную величину. Это поможет несколько сгладить погрешности, возникающие вследствие случайности входных данных.

Полученный график $T(n)$ является экспериментальной оценкой функции сложности алгоритма. Асимптотическое поведение $T(n)$ в области больших n также позволяет судить о порядке алгоритма. В частности, на рисунке выше, на экспериментальные данные (точки) наложен график параболы вида $f(n) = an^2 + c$ (сплошная линия) для сравнения. Удовлетворительное согласие между двумя этими кривыми позволяет нам сделать вывод о том, что исследуемый алгоритм имеет второй порядок сложности, $O(n^2)$.

Задание

1) Проведите экспериментальное исследование функции сложности алгоритма, реализованного в лабораторной работе №1. Для этого проведите серию экспериментов, в которых будет фиксироваться время работы программы для различных значений n (где n – размер используемой структуры). Постройте график полученной зависимости.
2) Оцените порядок сложности $O(f(n))$ своего алгоритма на основе анализа исходного кода (наличие циклов, рекурсивных вызовов функций и т.д.). Сравните измеренную экспериментально зависимость с теоретической.
3) Повторите измерения п. 1 для различных типов данных (int, float, double, char). Сравните эффективность работы алгоритма для различных типов данных.

Отчет по лабораторной работе №2 должен содержать

1) табличные данные о времени работы программы T для разных n

n	1000	2000	5000	10000	20000	50000	100000	200000	500000
T, мс									

2) графики зависимости f(n)