МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Брянский государственный технический университет

Утверждаю

Ректор университета

_____ О.Н. Федонин

«_____» _____ 2017г.

СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

ОЦЕНКА СЛОЖНОСТИ АЛГОРИТМА

Методические указания
к выполнению лабораторной работы №1
для студентов очной формы обучения
по направлениям подготовки
09.03.01 «Информатика и вычислительная техника»
09.03.04 «Программная инженерия»
02.03.03 «Математическое обеспечение и администрирование информационных систем»

Брянск 2017

УДК 004.01

Структуры и алгоритмы обработки данных. Оценка сложности алгоритма [Электронный ресурс]: методические указания к выполнению лабораторной работы №1 для студентов очной формы обучения по направлению подготовки 09.03.01 «Информатика и вычислительная техника»; 09.03.04 – «Программная инженерия»; 02.03.03 «Математическое обеспечение и администрирование информационных систем» – Брянск, 2017. – 31с.

Разработали
В.К. Гулаков
канд. техн. наук, проф.,
А.О. Трубаков
канд. техн. наук, доц.,
С.Н. Зимин
ст. преп.

Рекомендовано кафедрой «Информатика и программное обеспечение» БГТУ (протокол № 1 от 01.09.2017г.)

Методические издания публикуются в авторской редакции

1. ЦЕЛЬ РАБОТЫ

Цель работы – изучение и применение на практике основных методов вычислительной сложности алгоритмов. Для этого студент должен:

- изучить способы оценки вычислительной сложности алгоритма (количества операций и дополнительных затрат памяти);
- выработать навыки оценки алгоритмов, исследования предложенных идей;
- освоить способы анализа сильных и слабых сторон алгоритма.

В результате выполнения лабораторной работы студент должен приобрести углубление навыков по выбору алгоритмов для решения практических задач.

Продолжительность работы – 4 часа.

2. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Если бы компьютеры были неограниченно быстрыми, подошел бы любой корректный метод решения задачи. В этом случае выбирался бы метод, который легче всего реализовать. Конечно же, сегодня есть весьма производительные компьютеры, но их быстродействие не может быть бесконечно большим. Память тоже дешевеет, но она не может быть бесплатной. Таким образом, время вычисления – это такой же ограниченный ресурс, как и объем необходимой памяти. Этими ресурсами следует распоряжаться разумно, чему и способствует применение алгоритмов, эффективных в плане расходов времени и памяти.

Алгоритм – это точное предписание, однозначно определяющее вычислительный процесс, ведущий от варьируемых начальных данных к искомому результату.

При разработке алгоритмов очень важно иметь возможность оценить ресурсы, необходимые для проведения вычислений. Результатом оценки является функция сложности (трудоемкости). Оцениваемым ресурсом чаще всего является процессорное время (вычислительная сложность) и память (сложность алгоритма по памяти). Оценка позволяет предсказать время выполнения и сравнивать эффективность алгоритмов.

2.1. Понятие эффективности алгоритма

Алгоритмы, разработанные для решения одной и то же задачи, часто очень сильно различаются по эффективности. Эти различия могут быть намного значительнее, чем те, что вызваны применением неодинакового аппаратного и программного обеспечения.

В качестве примера можно привести два алгоритма сортировки – сортировка вставкой и сортировка слиянием. Известно, что для выполнения первого из них требуется время, которое оценивается как $c_1 \cdot n^2$, где n – количество сортируемых элементов, а c_1 – константа, не зависящая от n. Таким образом, время работы этого алгоритма приблизительно пропорционально n^2. Для выполнения второго алгоритма, сортировки слиянием, требуется время, приблизительно равное $c_2 \cdot n \cdot \log_2 n$, где c_2 – некоторая другая константа, не зависящая от n.

Можно легко убедиться, что постоянные множители намного меньше влияют на время работы алгоритма, чем множители, зависящие от n (при достаточно больших n). Для двух приведенных методов последние относятся как $\log_2 n$ к n. Когда n становится достаточно большим, все заметнее проявляется преимущество сортировки слиянием, возникающее благодаря тому, что для больших n незначительная величина $\log_2 n$ по сравнению с n полностью компенсирует разницу величин постоянных множителей.

Например, пусть каждый алгоритм для выполнения одного действия тратит всего 1 операцию, а общее количество сортируемых элементов равно 10 000. Тогда сортировка вставками потребует количество действий, пропорциональное $n^2 = 100\,000\,000$, а сортировка слиянием – $n \cdot \log_2 n \approx 132\,000$. Превосходство второго алгоритма заметны не вооруженным взглядом. При этом не имеет значения, во сколько раз константа c_1 меньше или больше, чем c_2. С ростом количества сортируемых элементов обязательно будет достигнут переломный момент, когда сортировка слиянием окажется более производительной.

2.2. Понятие сложности алгоритма

Вычислительная сложность – понятие в информатике и теории алгоритмов, обозначающее функцию зависимости объёма работы, которая выполняется некоторым алгоритмом, от размера входных

данных. Раздел, изучающий вычислительную сложность, называется теорией сложности вычислений. Объём работы обычно измеряется абстрактными понятиями времени и пространства, называемыми вычислительными ресурсами. Время определяется количеством элементарных шагов, необходимых для решения задачи и часто обозначается как функция $T(n)$, тогда как пространство определяется объёмом памяти или места на носителе данных и обозначается $V(n)$. Таким образом, в этой области предпринимается попытка ответить на центральный вопрос разработки алгоритмов: как изменится время исполнения T и объём занятой памяти V в зависимости от размера входных данных n?

Многие алгоритмы предлагают выбор между объёмом памяти и скоростью. Задачу можно решить быстро, использую большой объём памяти, или медленнее, занимая меньший объём.

Типичным примером в данном случае служит алгоритм поиска кратчайшего пути. Представив карту города в виде сети, можно написать алгоритм для определения кратчайшего расстояния между двумя любыми точками этой сети. Чтобы не вычислять эти расстояния всякий раз, когда они нам нужны, мы можем вывести кратчайшие расстояния между всеми точками и сохранить результаты в таблице. Когда нам понадобится узнать кратчайшее расстояние между двумя заданными точками, мы можем просто взять готовое расстояние из таблицы.

Результат будет получен мгновенно и практически не потребует затрат процессорного времени, но это потребует огромного объёма памяти. Карта большого города может содержать десятки тысяч точек. Тогда, описанная выше таблица, должна содержать более 10 млрд. ячеек. Т.е. для того, чтобы повысить быстродействие алгоритма, необходимо использовать дополнительные 10 Гб памяти.

Из этой зависимости проистекает идея объёмно-временной сложности. При таком подходе алгоритм оценивается, как с точки зрении скорости выполнения, так и с точки зрения потреблённой памяти.

Чаще всего основное внимание уделяется временной сложности, но, тем не менее, обязательно стоит оговаривать и объём потребляемой памяти.

2.3. Наилучший, средний, наихудший случаи

Время работы большинства алгоритмов очень сильно зависит не только от количества обрабатываемых данных, но и от самих данных и конкретного запроса. Допустим нам нужно найти заданное число в произвольном массиве целых чисел размером n элементов. Алгоритм для решения данной задачи достаточно прост. Необходимо в цикле от начала массива бежать по нему и проверять, нашли ли мы то что искали. Если элемент найден, возвращаем его, нет – бежим дальше. Код данного алгоритма показан в листинге 1.

Листинг 1

```
bool Find(int array[], int size, int n) {
    int i;
    for(i=0; i<size; i++) {
        if(array[i]==n)
            return true;
    }
    return false;
}
```

Определить необходимое количество операций (временную сложность $T(n)$) в вышеприведенном алгоритме не так просто. Дело в том, что количество операций будет зависеть от конкретных данных и запрашиваемого значения. При самом благоприятном стечении обстоятельств может оказаться так, что число, которое мы ищем, находится в самом начале массива. В этом случае нам будет достаточно выполнить всего одно сравнение, чтобы найти то, что нам нужно и сложность будет равна $T(n)=1$.

Однако может произойти и так, что нужный нам элемент находится в самом конце (или вообще отсутствует в массиве). В этом случае нам понадобится пробежать массив до конца, выполнив тем самым $T(n)=n$ операций.

Оба эти случая маловероятны. Нас больше всего интересует наиболее ожидаемый вариант. Если элементы списка изначально беспорядочно смешаны, то искомый элемент может оказаться в любом месте списка. В среднем потребуется сделать $n/2$ сравнений (пройти приблизительно до середины массива), чтобы найти требуемый элемент. Значит сложность этого алгоритма в среднем составляет $T(n)=n/2$.

В данном случае порядок средней и ожидаемой сложности совпадает, но для многих алгоритмов наихудший случай сильно

отличается от ожидаемого. Например, алгоритм быстрой сортировки в наихудшем случае имеет сложность порядка n^2, в то время как ожидаемое поведение описывается оценкой $n \cdot \log_2 n$, что на много быстрее.

Таким образом для оценки алгоритма обычно выделяют следующие типы оценок.

- *Время работы алгоритма в наихудшем случае* – это верхний предел величины сложности алгоритма для любых входных данных. Располагая этим значением, мы точно знаем, что для выполнения алгоритма не потребуется большее количество времени. Не нужно будет делать каких-то сложных предположений о времени работы и надеяться, что на самом деле эта величина не будет превышена.
- *Время работы алгоритма в наилучшем случае* – это нижний предел величины сложности алгоритма, достигаемый в идеальном для него случае. Зная эту границу можно точно оценить минимальное время, быстрее которого программа не сможет найти решение ни при каких обстоятельствах. В большинстве случаев такие идеальные условия являются маловероятными и рассчитывать на такое стечение обстоятельств не стоит.
- *Среднее время работы алгоритма* (или его математическое ожидание) – время работы алгоритма для наиболее вероятного распределения данных. Однако при анализе усредненного времени работы возникает одна проблема, которая заключается в том, что не всегда очевидно, какие входные данные для данной задачи будут «усредненными». Часто делается предположение, что все наборы входных параметров одного и того же объема встречаются с одинаковой вероятностью. На практике это предположение может не соблюдаться.

2.4. Порядок роста сложности алгоритма

Для облегчения анализа сложности алгоритма чаще всего делаются некоторые упрощающие предположения. Во-первых, игнорируется фактическое время выполнения каждой инструкции, представив эту величину в виде некоторой константы. Во-вторых, не ведется учет всех констант, которые дают излишнюю информацию.

Например, при сортировке данных из файла не учитывается время открытия и закрытия файла, другие подготовительные шаги. Это может существенно влиять при небольших размерах входных

данных, но в общем случае, когда количество обрабатываемых элементов n достаточно велико, все это не имеет значения.

Исходя из всего вышесказанного можно ввести еще одно абстрактное понятие, упрощающее анализ. Это *скорость роста*, или *порядок роста* времени работы, который и интересует нас на самом деле.

Порядок роста – это асимптотическая величина, которая показывает не точное время выполнения алгоритма, а то, как время работы алгоритма растет с увеличением размера входных данных в пределе, когда этот размер увеличивается до бесконечности. Именно эта величина отражает реальное поведение алгоритма при возрастании размера входных данных.

В многочисленных исследованиях было показано, что для порядка сложности имеет значение только главный член формулы сложности алгоритма. Допустим сложность алгоритма имеет вид:

$$T(n) = c_1 \cdot n^2 + c_2 \cdot n + c_3,$$

где c_1, c_2 и c_3 – некоторые константы.

В этом случае при больших n все слагаемые кроме первого (самого большого) вносят малый эффект и их можно отбросить. Кроме того, постоянные множители при главном члене также будут игнорироваться, так как для оценки вычислительной эффективности алгоритма с входными данными большого объема они менее важны, чем порядок роста. Таким образом, время работы алгоритма равно $\theta(n^2)$ (произносится как «тета от n в квадрате»).

Обычно один алгоритм считается эффективнее другого, если время его работы в наихудшем случае имеет более низкий порядок роста. Из-за наличия постоянных множителей и второстепенных членов эта оценка может быть ошибочной, если входные данные невелики. Однако если объем входных данных значительный, то, например, алгоритм $\Theta(n^2)$ в наихудшем случае работает быстрее, чем алгоритм $\Theta(n^3)$.

2.5. Условные обозначения асимптотических приближений

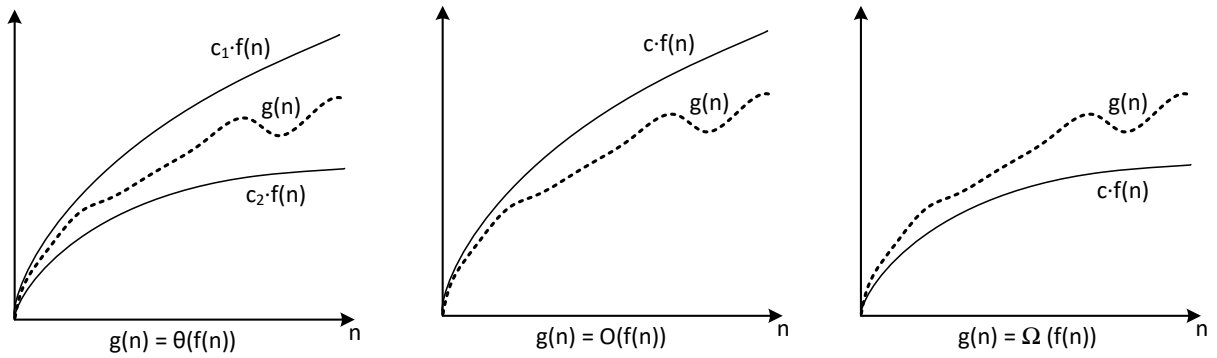
Известно, что время работы алгоритма сортировки методом вставок в наихудшем случае выражается функцией $\Theta(n^2)$. Рассмотрим подробнее смысловую нагрузку данного обозначения.

Для некоторой функции $g(n)$ запись $\Theta(f(n))$ обозначает множество функций $f(n)$, таких, что существуют некоторые константы c_1 и c_2, при которых

$$0 \le c_1 \cdot f(n) \le g(n) \le c_2 \cdot f(n),$$

для всех n больших некоторого порогового значения n_0.

Логический смысл данного высказывания говорит о том, что $\Theta(f(n))$ – это функция, которая при некоторых положительных константах c_1 и c_2, позволяет заключить $g(n)$ в рамки между функциями $c_1 \cdot f(n)$ и $c_2 \cdot f(n)$ для достаточно больших n. Графически это показано на рис. 1а



а б в

Рис. 1. Графические примеры Θ, O и Ω

Говорят, что функция $f(n)$ является асимптотически точной оценкой функции $g(n)$.

В обозначениях оценке Θ функция асимптотически ограничивается сверху и снизу (как на рис. 1а). Очень часто в оценках сложности алгоритма достаточно определить только асимптотическую верхнюю границу. В этом случае используются O-обозначения. Для данной функции $g(n)$ обозначение $O(f(n))$ (произносится «О большое от f от n») означает что $f(n)$ является асимптотической верхней границей для $g(n)$. Схематически это показан на рис. 1б.

Аналогично этому можно ввести асимптотическую нижнюю границу. Обозначается она как $\Omega(f(n))$ (произносится «омега большое от f от n»). Схематически она показана на рис. 1в.

2.6. P, NP, NPC классы сложности алгоритма

Почти все наиболее популярные и часто используемые алгоритмы имеют полиномиальное время работы (*polynomial-time algorithms*): для входных данных размера n их время работы в наихудшем случае равно $O(n^k)$, где $к$ – некоторая константа. Этот класс задач получил название *P*-класс.

Однако далеко не все задачи можно решить в течение полиномиального времени. В качестве примера можно привести знаменитую «задачу останова», предложенную Тьюрингом. Эту задачу невозможно решить ни на одном компьютере, каким бы количеством времени мы не располагали.

Существуют также задачи, которые можно решить, но не удается сделать это за время $O(n^k)$. Вообще говоря, о задачах, разрешимых с помощью алгоритмов с полиномиальным временем работы, возникает представление как о легко разрешимых или простых, а о задачах, время работы которых превосходит полиномиальное – как о трудно разрешимых или сложных.

Класс *NP* состоит из задач, которые поддаются проверке в течение полиномиального времени. Имеется в виду, что если мы каким-то образом получаем готовое решение, то в течение времени, полиномиальным образом зависящего от размера входных данных задачи, можно проверить корректность такого решения.

Например, в задаче поиска пути в графе (на карте местности, например) готовое решение имело бы вид последовательности $(v_1, v_2, \dots, v_k)$ из k вершин. В течение полиномиального времени легко проверить, что эта последовательность действительно образует замкнутый путь из начальной точки в конечную.

Любая задача класса P принадлежит классу *NP*, поскольку принадлежность задачи классу P означает, что ее решение можно получить в течение полиномиального времени, даже заранее не располагая таковым. Так что можно считать, что $P \subseteq NP$. Однако остается открытым вопрос, является ли P строгим подмножеством *NP*.

Есть еще один интересный класс задач – *NPC* (*NP*-полные задачи, *NP-complete*). Статус таких задач пока что неизвестен. Для решения *NP*-полных задач до настоящего времени не разработано алгоритмов с полиномиальным временем работы, но и не доказано, что для какой-то из них таких алгоритмов не существует.

Особо интригующим аспектом *NP*-полных задач является то, что некоторые из них на первый взгляд аналогичны задачам, для решения которых существуют алгоритмы с полиномиальным временем работы. При этом различие между задачами кажется совершенно незначительным.

Неформально задача принадлежит классу *NP*-полных, если она принадлежит классу *NP* и является такой же «сложной», как и любая

задача из класса *NP*. При этом доказано, что если любую *NP*-полную задачу можно решить в течение полиномиального времени, то для каждой задачи из класса *NP* существует алгоритм с полиномиальным временем работы. Большинство ученых, занимающихся теорией вычислительных систем, считают *NP*-полные задачи очень трудноразрешимыми, потому что при огромном разнообразии излучавшихся до настоящего времени *NP*-полных задач ни для одной из них пока так и не найдено решение в виде алгоритма с полиномиальным временем работы.

Чтобы стать квалифицированным разработчиком алгоритмов, необходимо понимать основы теории *NP*-полноты. Если установлено, что задача *NP*-полная, это служит достаточно надежным указанием на то, что она трудноразрешимая. Как инженер, вы эффективнее потратите время, если займетесь разработкой приближенного алгоритма или решением легкого особого случая, вместо того, чтобы искать быстрый алгоритм, выдающий точное решение задачи. Более того, многие естественно возникающие интересные задачи, которые на первый взгляд не сложнее задачи сортировки, поиска в графе или определения потока в сети, фактически являются *NP*-полными.

2.7. Наиболее часто встречающиеся порядки сложности алгоритмов

При сравнении различных алгоритмов важно понимать, как определенная сложность зависит от объёма входных данных. Т.е. по найденному или вычисленному значению сложности понимать применимость данного алгоритма и его эффективность и рост времени выполнения при росте объема входных данных.

В общем случае сложность алгоритма можно оценить по порядку величины. Выше было показано, что алгоритм имеет сложность O($f(n)$), если при увеличении размерности входных данных n, время выполнения алгоритма возрастает с той же скоростью, что и функция $f(n)$.

Рассмотрим некоторые функции, которые чаще всего используются для характеристик сложности алгоритма. Функции перечислены в порядке возрастания сложности. Чем выше в этом списке находится функция, тем быстрее будет выполняться алгоритм с такой оценкой.

1. $O(1)$ – *константная сложность*. Время работы программы не зависит от объема входных данных и является константой. Самый хороший вариант из возможных, т. к. время работы известно заранее.
2. $O(\log_2 n)$ – *логарифмическая сложность*. Очень хорошая сложность, т. к. увеличение объема входных данных слабо влияет на время выполнения программы. Например, увеличение входных данных в 500 раз приведет к увеличения времени мене чем в 9 раз.
3. $O(n)$ – *линейная сложность*. Данная сложность является предсказуемой и линейной. Во сколько раз увеличиваются входные данные, во столько же раз увеличивается и время выполнения программы.
4. $O(n \cdot \log_2 n)$. Данный вариант сложности является достаточно хорошим для единовременных задач. Например, сортировка данных, которые потом будут использоваться многократно в уже отсортированном порядке.
5. $O(n^2)$ – *квадратичная сложность*. Алгоритм, обладающий такой сложностью, является достаточно трудоемким. Но его выполнение еще возможно при приемлемых объемах входных данных.
6. $O(n^a)$, где a>2. Время выполнения программы будет достаточно большим. В подобных случаях требуется дополнительный анализ условий решения задачи, т.к. время выполнения может стать неприемлемо большим.
7. $O(a^n)$. Решение возможно только для небольших объемов входных данных.
8. $O(n!)$. Наиболее неприятный случай, т.к. время работы растет с очень большой скоростью. Добавление всего одного элемента к входным данным может привести к тому, что задача станет нерешенной за приемлемое время.

Если алгоритм вызывается редко и для небольших объёмов данных, то приемлемой можно считать сложность $O(n^2)$. Если же алгоритм работает в реальном времени, то не всегда достаточно производительности $O(n)$.

Обычно алгоритмы со сложностью $O(n \cdot \log_2 n)$ работают с хорошей скоростью. Алгоритмы со сложностью $O(a^n)$ можно использовать только при небольших значениях a. Вычислительная сложность алгоритмов, порядок которых определяется функциями

$O(a^n)$ и $O(n!)$ очень велика, поэтому такие алгоритмы могут использоваться только для обработки ограниченного числа элементов.

В заключение приведём таблицу, которая показывает, как долго компьютер, осуществляющий миллион операций в секунду, будет выполнять некоторые медленные алгоритмы.

Таблица 1

Примеры выполнения алгоритмов

	n=10	**n=20**	**n=30**	**n=40**	**n=50**
$\mathbf{n^3}$	0,001 с	0,008 с	0,027 с	0,064 с	0,125 с
$\mathbf{2^n}$	0,001 с	1,05 с	17,9 мин	1,29 дней	35,7 лет
$\mathbf{3^n}$	0,059 с	58,1 мин	6,53 лет	$3{,}86*10^5$ лет	$2{,}28*10^{10}$ лет
n!	3,63 с	$7{,}71*10^4$ лет	$8{,}41*10^{10}$ лет	$2{,}59*10^{34}$ лет	$9{,}64*10^{50}$ лет

3. ПРИМЕРЫ ГРУБОЙ ОЦЕНКИ СЛОЖНОСТИ АЛГОРИТМА

3.1. Пример оценки

Для примера рассмотрим код, который для матрицы $A[n*n]$ находит максимальный элемент в каждой строке. Код данной программы показа в листинге 2

Листинг 2

```
int i, j;
for(i=0; i<n; i++) {
     max = A[i][0];
     for(j=0;j<n;j++) {
          if(A[i][j]>max)
               max = A[i][j];
     }
     printf("Макс. элемент %d-й строки %d", i, max);
}
```

В этом алгоритме используется несколько вложенных друг в друга циклов. В первом цикле переменная *i* меняется от 1 до *n*. При каждом изменении *i*, переменная вложенного цикла *j* тоже меняется от 1 до *n*. Во время каждой из *n* итераций внешнего цикла, внутренний цикл тоже выполняется *n* раз. Общее количество итераций внутреннего цикла равно $T(n) = n \cdot n = n^2$. Это определяет сложность алгоритма $O(n^2)$.

Оценивая порядок сложности алгоритма, необходимо использовать только ту часть, которая возрастает быстрее всего. Предположим, что рабочий цикл описывается выражением $T(n) = n^3 + n$. В таком случае его сложность будет равна $O(n^3)$. Рассмотрение быстро растущей части функции позволяет оценить поведение алгоритма при увеличении n. Например, при n=100, разница между $n^3 + n = 1000100$ и $n = 1000000$ равна всего лишь 100, что составляет 0,01%.

Так же при вычислении O можно не учитывать постоянные множители в выражениях. Алгоритм с рабочим шагом $T(n) = 3n^3$ рассматривается, как $O(n^3)$. Это делает зависимость отношения $O(n)$ от изменения размера задачи более очевидной.

3.2. Общая методика грубой оценки

Наиболее сложными частями программы обычно является выполнение циклов и вызов процедур. В предыдущем примере весь алгоритм выполнен с помощью двух циклов.

Если одна процедура вызывает другую, то необходимо более тщательно оценить сложность последней. Если в вызываемой процедуре выполняется $O(n)$ шагов, то функция может значительно усложнить алгоритм. Если же процедура вызывается внутри цикла, то влияние может быть намного больше.

В качестве примера рассмотрим две процедуры: *Slow* со сложностью $O(n^3)$ и *Slow2* со сложностью $O(n^2)$ (см. листинг 3).

Листинг 3

```
void Slow() {
     for(i=0; i<n; i++) {
          for(j=0; j<n; j++) {
               for(k=0; k<n; k++) {
                    [КАКОЕ-ТО ДЕЙСТВИЕ]
               }
          }
     }
}
void Slow2() {
     for(i=0; i<n; i++) {
          for(j=0; j<n; j++) {
               Slow();
          }
     }
}
```

```
void main() {
     Slow2();
}
```

Если во внутренних циклах процедуры *Slow2* происходит вызов процедуры *Slow*, то сложности процедур перемножаются. В данном случае сложность алгоритма составляет $O(n^2 \cdot n^3) = O(n^5)$.

Если же основная программа вызывает процедуры по очереди, то их сложности складываются: $O(n^2 + n^3) = O(n^3)$. Следующий фрагмент имеет именно такую сложность (см. листинг 4).

Листинг 4

```
void Slow() {
     for(i=0; i<n; i++) {
          for(j=0; j<n; j++) {
               for(k=0; k<n; k++) {
                    [КАКОЕ-ТО ДЕЙСТВИЕ]
               }
          }
     }
}

void Slow2() {
     for(i=0; i<n; i++) {
          for(j=0; j<n; j++) {
               [КАКОЕ-ТО ДЕЙСТВИЕ]
          }
     }
}

void main() {
     Slow();
     Slow2();
}
```

3.3. Сложность рекурсивных алгоритмов

Напомним, что рекурсивными процедурами называются процедуры, которые вызывают сами себя. Их сложность определить довольно тяжело. Сложность этих алгоритмов зависит не только от сложности внутренних циклов, но и от количества итераций рекурсии. Рекурсивная процедура может выглядеть достаточно

простой, но она может серьёзно усложнить программу, многократно вызывая себя.

Рассмотрим рекурсивную реализацию вычисления факториала (см. листинг 5).

Листинг 5

```
void Factorial(int n) {
    if(n>1)
        return n*Factorial(n-1);
    else
        return 1;
}
```

Эта процедура выполняется n раз, таким образом, вычислительная сложность этого алгоритма равна $O(n)$.

Рекурсивный алгоритм, который вызывает себя несколько раз, называется многократной рекурсией. Такие процедуры гораздо сложнее анализировать, кроме того, они могут сделать алгоритм гораздо сложнее.

Рассмотрим такую процедуру (см. листинг 6).

Листинг 6

```
void Recursive2(int n) {
    if(n>2)
        return Recursive2 (n-1)* Recursive2 (n-2);
    else
        return 1;
}
```

Поскольку процедура вызывается дважды, можно было бы предположить, что её рабочий цикл будет равен $O(2n) = O(n)$. Но на самом деле ситуация гораздо сложнее. Если внимательно исследовать этот алгоритм, то станет очевидно, что его сложность равна $O\left(2^{(n+1)-1}\right) = O(2^n)$. Т.е. на самом деле задача, не смотря на внешнее сходство кода с предыдущим, является трудноразрешимой в вычислительном плане. В этом не трудно убедиться запустив данный код на выполнения для какого-то n.

Всегда надо помнить, что анализ сложности рекурсивных алгоритмов весьма нетривиальная задача.

Так же для всех рекурсивных алгоритмов очень важно понятие объёмной сложности. При каждом вызове процедура запрашивает небольшой объём памяти, но этот объём может значительно

увеличиваться в процессе рекурсивных вызовов. По этой причине всегда необходимо проводить хотя бы поверхностный анализ объёмной сложности рекурсивных процедур.

4. ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОГО РЕШЕНИЯ

Для выполнения данной лабораторной работы студенту необходимо сделать следующую последовательность шагов:

- изучить методические указания по выполнению лабораторной работы и получить индивидуальное задание;
- проанализировать поставленную задачу, изучить алгоритмы решения подобных задач;
- произвести математическую оценку изученных и придуманных самостоятельно алгоритмов, рассчитать количество операций и затраты памяти, необходимые в лучшем, худшем и среднем случае (число алгоритмов должно быть не менее 2-х, желательно 3);
- выбрать наилучший алгоритм с точки зрения вычислительной сложности;
- составить программу на алгоритмическом языке Паскаль или C/C++;
- составить контрольный пример;
- отладить программу;
- проверить свои предположения, сделанные в отношении количества операций;
- пройти собеседование с преподавателем на тему математической сложности алгоритма и реализованной программы.

Варианты индивидуальных заданий выбираются из списка ниже согласно номеру студента в журнале.

1. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Вдоль линии расположены лунки. В каждой лунке лежит красный, белый или синий шар. Одним ходом разрешается менять местами два любых шара. Добиться того, чтобы все красные шары шли первыми, все синие – последними, а белые – посередине.

Комментарий 1: известно, что хороший алгоритм будет находить решение данной задачи не более чем за *(n-1)* ход (где *n* – число лунок).

Комментарий 2: для удобства программирования, можно прибегнуть к числовому кодированию. Например, можно закодировать красные шары – 0, белые – 1, синие – 2.

Пример входных данных	*Результат*
0 1 2 0 1 2 0 1 2	0 0 0 1 1 1 2 2 2
2 2 1 1 0 0 0	0 0 0 1 1 2 2
1 1 1 1	1 1 1 1

2. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Имеется массив целых чисел. Каждое число может принимать значение от 0 до 10^{10}. Числа в исходном массиве могут повторяться. Необходимо узнать, сколько чисел в этом массиве являются уникальными (число называется уникальным, если оно встречается в массиве ровно один раз).

Комментарий: известно, что данную задачу можно решить за $n*log_2(n)$ шагов.

Пример входных данных	*Результат*
1 2 6 2 1	1
10 25 38 43 58 62	6
8 12 10 6 6 8 10 12	0

3. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Имеется массив целых чисел. Каждое число может принимать значение от -10^{10} до 10^{10}. Необходимо преобразовать массив так, чтобы все отрицательные значения предшествовали положительным.

Комментарий 1: элементы не обязательно сортировать, достаточно просто отделить отрицательные числа от положительных.

Комментарий 2: известно, что данную задачу можно решить за *n* шагов.

Пример входных данных	*Результат*
100 -121 200 -357	-121 -357 100 200
-456 298 4324 -8456 37298 -80024	-456 -8456 -80024 298 4324 37298
-342 -455 653 234 523	-342 -455 653 234 523

4. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Имеются две последовательности целых чисел упорядоченных по возрастанию. Числа внутри каждой последовательности не повторяются. Необходимо произвести слияние этих последовательностей в одну с учетом повторений.

Комментарий: известно, что данную задачу можно решить за n шагов.

Пример входных данных	*Результат*
1 2 3 4 6 1 2 4 7 8	1 2 3 4 6 7 8
1 3 8 23 67 10 39 543	1 3 8 10 23 39 67 543
12 23 34 45 56 67 78 89 23 34 56 67 89	12 23 34 45 56 67 78 89

5. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Дан массив целых чисел $a_1,a_2,\ldots a_n$, заполненный случайно. Необходимо переставить элементы массива так, чтобы вначале в массиве шла группа элементов, больших a_1, затем сам этот элемент a_1, потом группа элементов, меньших или равных ему.

Комментарий: известно, что данную задачу можно решить таким способом, что число сравнений и перемещений, каждое в отдельности не будет превышать n-1.

Пример входных данных	*Результат*
12 4 5 53 75 21	53 75 21 12 4 5
54 31 67 54	67 54 54 31
1 2 3 4 5 6	2 3 4 5 6 1

6. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Дан набор чисел $a_1,a_2,\ldots,a_n$ и последовательность перестановок $p_1,p_2,\ldots,p_n$. Необходимо расположить исходный набор чисел в заданном порядке $a_{p1},a_{p2},\ldots,a_{pn}$.

Комментарий: данную задачу можно решить без использования дополнительного массива.

Пример входных данных	*Результат*
241 423 765 543 4 3 2 1	543 765 423 241
432 56465 6564 743 5634 1 3 5 2 4	432 6564 5634 56465 743
5435 8967 6765 7876 95534 1 2 3 4 5	5435 8967 6765 7876 95534

7. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

В памяти ЭВМ хранятся списки номеров телефонов и фамилий абонентов, упорядоченных в алфавитном порядке. Необходимо разработать алгоритм, который обеспечит быстрый поиск телефона абонента по фамилии.

Пример входных данных	*Результат*
Иванов И.И. 12-45-67 Пертов П.П. 43-76-23 Сидоров С.С. 76-23-98 Федоров Ф.Ф. 43-76-23 Кемеров К.К. 54-23-12 > Сидоров	76-23-98

8. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

В лотереи вытаскивают два бочонка с цифрами. На каждом бочонке может быть цифра от 0 до n. Сколько существует возможных комбинаций, при которых сумма чисел на этих бочонках окажется равна m.

Комментарий: для улучшения быстродействия необходимо подумать в сторону различных вариантов отсечения.

Пример входных данных	*Результат*
n=6 m=12	Комбинаций – 1 (6;6)
n=7 m=12	Комбинаций – 3 (6;6), (5;7), (7;5)
n=10 *m=25*	Комбинаций – 0

9. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Дан набор из *n* цифр. Необходимо сформировать все различные четные двухзначные числа из этого набора.

Комментарий: данную задачу можно решить без полного перебора всех возможных комбинаций и проверки их на четность.

Пример входных данных	*Результат*
1 2 3 5	Наборов – 3 12 32 52
1 2 3 4	Наборов – 6 12 32 42 14 24 34
2 4 3 2 1 2 3 4 1	Наборов – 8 12 32 42 22 14 24 34 44

10. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Дан упорядоченный по возрастанию набор из *n* чисел. Необходимо выбрать из этого набора только простые числа.

Комментарий: достаточной быстротой обладает алгоритм «решето Эротосфена».

Пример входных данных	*Результат*
1 2 3 4 5 6 7 8 9 10	1 2 3 5 7
5 18 19 28	5 19
7 11 13	7 11 13

11. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Даны *m* наборов, каждый из которых состоит из *n* чисел. В каждом наборе можно переставлять цифры произвольным образом, получая при этом некоторое число. Необходимо проверить, можно ли сформировать из этих наборов возрастающую последовательность чисел.

Комментарий 1: для улучшения быстродействия необходимо подумать в сторону различных вариантов отсечения.

Комментарий 2: так же стоит заметить, что перебирать все комбинации не обязательно, достаточно на каждом этапе формировать минимальное из возможных чисел, превосходящее предыдущее в последовательности.

Пример входных данных	*Результат*
{8,1} {8,2} {5,6} {4,7}	18 28 56 74
{9,3,8} {1,3,2} {3,7,2}	Сформировать последовательность нельзя

12. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Дана упорядоченная последовательность целых чисел. Необходимо определить, какие из этих чисел не являются точными квадратами.

Комментарий 1: операция извлечения корня является очень трудоемкой.

Комментарий 2: использование квадратного корня можно заменить последовательностью делений.

Пример входных данных	*Результат*
4 9 16 20 25	20
1 100 1000 10000	1 1000
2 4 8 16 32 64	2 8 32

13. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Дана последовательность n натуральных чисел $a_1, a_2, \ldots, a_n$, и некоторое нечетное число m. Необходимо переставить числа так, чтобы сначала располагались все нечетные числа, кратные m, затем все остальные нечетные числа, а потом все четные числа. При этом каждая из 3-х частей должна быть отсортирована по возрастанию.

Пример входных данных	*Результат*
n=9, m=5 10 15 16 12 25 9 7 35 13	15 25 35 7 9 13 10 12 16
n=8, m=3 11 10 15 21 17 25 20 39	15 21 39 11 17 25 10 20
n=6, m=9 11 15 21 17 25 39	11 15 17 21 25 39

14. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Игроки в кости бросают n кубиков. На каждом кубике может выпасть число от 1 до 6. Сколько существует вариантов получить сумму, равную m.

Комментарий 1: число кубиков n и нужная сумма m должны задаваться с клавиатуры.

Комментарий 2: для улучшения эффективности можно использовать различные варианты отсечения.

Пример входных данных	*Результат*
n=2, *m*=10	Вариантов – 3 (4;6), (5;5), (6;4)
n=2, *m*=2	Вариантов – 1 (1;1)
n=3, *m*=4	Вариантов – 3 (1;1;2), (1;2;1), (2;1;1)

15. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Дана последовательность целых положительных чисел. Каждое число может принимать значение от 100 до 10^9. Необходимо посчитать, сколько чисел в данной последовательности имеют уникальные две последние цифры.

Комментарий 1: число считается уникальным и его нужно учитывать при подсчете только в том случае, если в массиве больше нет чисел с такими же двумя последними цифрами.

Комментарий 2: известен алгоритм, решающий эту задачу со сложностью *O(n)*.

Пример входных данных	*Результат*
100 121 200 357	2
456 298 4324 8456 37298 80024	0
342 455 653 234 523	5

16. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Дана таблица клиентов фирмы. В каждой строке таблицы указаны ФИО и тип клиента (целое число от 1 до 5). Необходимо переставить записи так, чтобы все клиенты были сгруппированы по типу клиента (причем группы шли в порядке возрастания номера).

Пример входных данных	*Результат*
Иванов Иван Иванович 1 Петров Петр Петрович 4 Сидоров Иван Васильевич 1 Кривин Емельян Игнатьевич 2 Простаков Петр Сергеевич 5	Иванов Иван Иванович 1 Сидоров Иван Васильевич 1 Кривин Емельян Игнатьевич 2 Петров Петр Петрович 4 Простаков Петр Сергеевич 5

17. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Даны три массива. Необходимо переставить элементы так, чтобы в первом массиве были самые маленькие элементы, в третьем — самые большие — в среднем — все остальные.

Пример входных данных	*Результат*
1 40 80 90 100 60 5 7 60	1 5 7 40 60 60 80 90 100
40 5 8 6 1 2 100 25 3	1 2 3 5 8 6 25 40 100

18. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Дан список студентов и их оценок за прошлую сессию. Необходимо отсортировать список так, что бы вначале были отличники, далее студенты с оценкой "4" и т. д.

Пример входных данных	*Результат*
Иванов Иван Иванович 5 4 3 5 Петров Петр Петрович 5 5 5 5 Сидоров Иван Васильевич 3 4 3 3 Кривин Емельян Игнатьевич 4 5 4 4 Простаков Петр Сергеевич 5 5 5 5	Петров Петр Петрович 5 5 5 5 Простаков Петр Сергеевич 5 5 5 5 Кривин Емельян Игнатьевич 4 5 4 4 Иванов Иван Иванович 5 4 3 5 Сидоров Иван Васильевич 3 4 3 3

19. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Дан двухмерный массив *n*m*. Найти два столбца, в которых находятся самые большие элементы.

Пример входных данных	*Результат*
n=3, *m*=3 1 2 3 4 5 6 7 8 9	2, 3
n=5, *m*=2 1 2 3 4 10 6 7 8 7 9	3, 5

20. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Дан массив целы чисел и три числа a,b,c. Необходимо расставить элементы в массиве так, чтобы сначала шли все

элементы, которые больше a+b и a-c, в конце — элементы меньше чем a+b, a-c, а в середине — все остальные элементы.

Комментарий 1: упорядочивать элементы при этом не обязательно.

Комментарий 2: известен алгоритм, решающий эту задачу со сложностью *O(n)*.

Пример входных данных	*Результат*
a=5, *b*=1, c=2 1 2 3 4 5 6 7 8 9	7 8 9 3 4 5 6 1 2
a=2, *b*=10, c=2 10 25 48 6 12 10 15 18 79 6	25 48 15 18 79 10 6 12 10 6

21. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Дана таблица обращений в службу поддержки сотового оператора . В каждой строке таблицы указан текст обращения и тип клиента (физическое лицо, бюджетная организация, коммерческая компания). Необходимо переставить записи так, чтобы все обращения были сгруппированы по типу клиента.

Пример входных данных	*Результат*
Бюджетная организация Сколько стоит ваш продукт? Коммерческая компания Есть ли скидки на объем? Бюджетная организация Какой срок гарантии? Физическое лицо Вы работаете с физ.лицами? Коммерческая компания Как с вами связаться?	Бюджетная организация Сколько стоит ваш продукт? Бюджетная организация Какой срок гарантии? Коммерческая компания Есть ли скидки на объем? Коммерческая компания Как с вами связаться? Физическое лицо Вы работаете с физ.лицами?

22. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Даны три массива. Необходимо в первом массиве заменить на -1 все элементы, которые есть либо во втором, либо в третьем массиве.

Пример входных данных	*Результат*
1 40 80	-1 -1 -1

90 40 60 80 1 60	
40 5 8 6 1 2 100 25 3	40 5 8 6

23. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Даны два числа a и b. Найти все простые числа в диапазоне от a до b.

Пример входных данных	*Результат*
a=1, b=10	1 2 3 5 7
a=7, b=20	7 11 13 17 19

24. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Дана таблица записей сведений о том, как студенты сдали зимнюю сессию. В каждой строке таблицы записано ФИО и пять оценок по разным предметам. Необходимо переставить записи в таблице так, чтобы они были сгруппированы по сумме баллов по всем экзаменам (сначала студенты с максимальной суммой и т. д.).

Пример входных данных	*Результат*
Иванов Иван Иванович 5 4 3 5 5 Петров Петр Петрович 5 5 5 5 5 Сидоров Иван Васильевич 3 4 3 3 3 Кривин Емельян Игнатьевич 4 5 4 4 5 Простаков Петр Сергеевич 5 5 5 5 5	Петров Петр Петрович 5 5 5 5 5 Простаков Петр Сергеевич 5 5 5 5 5 Иванов Иван Иванович 5 4 3 5 5 Кривин Емельян Игнатьевич 4 5 4 4 5 Сидоров Иван Васильевич 3 4 3 3 3

25. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Дана последовательность целых положительных чисел. Каждое число может принимать значение от 0 до 100 и встречаться в последовательности много раз. Необходимо вывести на экран только те числа, которые встречаются более 5 раз.

Комментарий: известен алгоритм, решающий эту задачу со сложностью *O(n)*.

Пример входных данных	*Результат*
1 2 3 4 5 6 7 8 9 1 1 1 2 1	1
10 25 48 6 12 10 15 18 79 6 8 6 49 6 8 6 8 8 8	6 8

26. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Дана последовательность n натуральных чисел $a_1, a_2, \ldots, a_n$, и некоторое нечетное число m. Необходимо переставить числа так, чтобы сначала располагались все четные числа, затем все нечетные числа кратные m, затем все остальные числа.

Комментарий 1: сортировать числа не обязательно.

Комментарий 2: данную задачу можно решить без использования дополнительного массива.

Комментарий 3: известен алгоритм, решающий эту задачу со сложностью $O(n)$.

Пример входных данных	*Результат*
n=9, m=5 10 15 16 12 25 9 7 35 13	10 16 12 15 25 35 9 7 13
n=8, m=3 11 10 15 21 17 25 20 39	10 20 15 21 39 11 17 25
n=6, m=9 11 15 21 17 25 39	11 15 21 17 25 39

27. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

В музее регистрируется в течение дня время прихода и ухода каждого посетителя. Таким образом за день получены n пар значений, где первое значение в паре показывает время прихода посетителя и второе значения – время его ухода. Найти промежуток времени, в течение которого в музее одновременно находилось максимальное число посетителей.

Комментарий: для решение задачи можно применить стек или сортировку.

Пример входных данных	*Результат*
n=3 11.30 – 12.00 11.45 – 13.00 11.50 – 15.00	Максимальное число посетителе было в период с 11.50 до 12.00 (3 человека)
n=5 15.30 – 17.00 16.45 – 17.00 17.20 – 18.00 17.20 – 17.50 17.30 – 18.00	Максимальное число посетителе было в период с 17.30 до 17.50 (3 человека)

28. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

В памяти ЭВМ хранятся англо-русский словарь. В этом словаре каждому английскому слову сопоставлен русский перевод. Словарь отсортирован по алфавиту английских слов. Необходимо разработать алгоритм, который обеспечит быстрый поиск перевода для заданного английского слова.

Пример входных данных	*Результат*
group – группа subject – предмет speciality – специальность university – университет > university	университет

29. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Даны два файла. В каждом файле находится список слов через пробелы. Необходимо сформировать третий файл, в котором будут только те слова, которые встречаются и в первом и во втором файлах.

Пример входных данных	*Результат*
слово1 слово2 слово3 слово4 слово1 слово3 слово5	слово1 слово3
слово1 слово2 слово3 слово4 слово1 слово2 слово3 слово4	слово1 слово2 слово3 слово4
слово1 слово2 слово3 слово4 слово5 слово6 слово7	

30. Придумать несколько алгоритмов и сравнить их порядок сложности в лучшем, среднем и худшем случаях для решения следующей задачи.

Имеется массив целых чисел. Каждое число может принимать значение от 0 до 100 000. Числа в исходном массиве могут повторяться. Необходимо сформировать массив, в котором будут находиться все исходные числа без повторений.

Комментарий 1: сохранения порядка следования чисел необязательно (в результирующем массиве числа могут следовать в любом порядке).

Комментарий 2: известно, что данную задачу можно решить за $n*log_2(n)$ шагов.

Пример входных данных	*Результат*
1 2 3 2 1	1 2 3
10 25 38 43 58 62	10 25 38 43 58 62
1 2 3 3 2 1 4 5	1 2 3 4 5

5. КОНТРОЛЬНЫЕ ВОПРОСЫ

Для проверки своих знаний при подготовке к сдаче лабораторной работы можно использовать следующие контрольные вопросы.

1. Что такое вычислительная сложность и какие параметры на неё влияют?
2. Что называется сложностью алгоритма по памяти? На что влияет данный показатель?
3. Какова будет сложно алгоритма, если он предполагает два вложенных друг в друга цикла?
4. Что в общем случае быстрее — алгоритм с линейной или логарифмической сложностью?
5. Может ли при каких-то условиях алгоритм с квадратичной сложностью обогнать алгоритм с линейной сложностью?
6. Как определить сложность алгоритма с рекурсией?
7. Важен ли постоянный константный коэффициент в слагаемом при определении порядка сложности?
8. Может ли рекурсивный алгоритм быть быстрее итерационного?

СПИСОК РЕКОМЕНДОВАННОЙ ЛИТЕРАТУРЫ

1. Кормен, Т. Алгоритмы. Построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. – М.: Вильямс, 2013. – 1324 с.

2. Гирш, Э.А. Сложность вычислений и основы криптографии. Курс лекций описывающий основы сложности вычислений и криптографии / Э.А. Гирш.

3. Лифшиц Ю. Современные задачи теоретической информатики. Курс лекций по алгоритмам для NP-трудных задач / Ю. Лифшиц.

4. Разборов, А.А. Theoretical Computer Science: взгляд математика / А.А. Разборов // Компьютерра. — 2001. — № 2.

5. Разборов, А.А. О сложности вычислений / А.А. Разборов // Математическое просвещение. — МЦНМО, 1999. — № 3. — С. 127-141.

6. Альфред, А.В. Структуры данных и алгоритмы / А.В. Альфред, Д.Э. Хопкрофт, У.Д. Джеффри. – М.: Издательский дом "Вильямс", 2000. – 384 с.

7. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт. – М.: МИР, 1989. – 368 с.

8. Кнут, Д. Искусство программирования для ЭВМ т.1, Основные алгоритмы / Д. Кнут. – М.: МИР, 1976. – 986 с.

9. Кнут, Д. Искусство программирования для ЭВМ Т. 3. Сортировка и поиск / Д. Кнут. – М.: МИР, 1978. – 932 с.

Структуры и алгоритмы обработки данных. Оценка сложности алгоритма [Электронный ресурс]: методические указания к выполнению лабораторной работы №1 для студентов очной формы обучения по направлению подготовки 09.03.01 «Информатика и вычислительная техника»; 09.03.04 – «Программная инженерия»; 02.03.03 «Математическое обеспечение и администрирование информационных систем» – Брянск, 2017. – 35с.

ГУЛАКОВ ВАСИЛИЙ КОНСТАНТИНОВИЧ
ТРУБАКОВ АНДРЕЙ ОЛЕГОВИЧ
ЗИМИН СЕРГЕЙ НИКОЛАЕВИЧ

Научный редактор Д.А. Коростелев
Компьютерный набор А.О. Трубаков
Иллюстрации А.О. Трубаков

Подписано в печать 01.09.2017г. Формат 60x84 1/16 Бумага офсетная. Офсетная печать. Усл.печ.л. 1,93 Уч.-изд.л. 1,93 Тираж 1 экз.

Брянский государственный технический университет
Кафедра «Информатика и программное обеспечение», тел. 56-09-84
241035, Брянск, бульвар 50 лет Октября, 7 БГТУ